1995

IPCC++: A concurrentC++ for Centralized and Distributed Memory Models.

Shelly S. Stubbs
Louisiana State University and Agricultural & Mechanical College

IPCC++: A CONCURRENT C++ FOR CENTRALIZED AND DISTRIBUTED MEMORY MODELS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agriculture and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Shelly S. Stubbs
B.S., Southeastern Louisiana University, 1985
December 1995

Acknowledgments

I would like to express my sincere thanks to my committee chair, *Dr. Doris L. Carver,* for her inspiration and guidance through out my college career. She has served as a valuable resource of information and direction with my research as well as every other aspect of my education.

I would like to extend a thank you to my minor professor *Dean Bert Boyce, Ph.D.*, and committee members: *Drs. Bush Jones, Donald Kraft,* and *Brian Marx* for their support and guidance.

I would like to extend a special thank you to *Dr. Andrew Hoppe*, for his insight into distributed computing and knowledge of the UNIX operating system. He served as a valuable source of information and inspiration.

Last, but not least, I would like to thank my family for their endless support and patience which always provides me with the strength to attain my goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

InterProcess Communication with C++, (IPCC++), is a concurrent object-oriented programming language that supports concurrency for centralized and distributed memory models while maintaining the high level of abstraction associated with object-oriented languages. The IPCC++ language model is a natural extension of the C++ programming language which introduces and supports the following features of concurrency: process concept, mechanism for process instantiation, static and dynamic process declaration, inter-object concurrency, monitor structure, condition variable, socket structure, typed message passing interprocess communication, synchronous and asynchronous communication, client/server paradigm, and run-time communication error detection. Features of concurrency are introduced as complete objects using the primitives of object-oriented programming languages as the vehicle for introduction. The underlying implementation of the components utilizes Parallel Virtual Machine (PVM), a software system that provides an abstraction of the UNIX operating system.

A description of the object-oriented and concurrency paradigms are presented. The IPCC++ language model, which represents both paradigms, is defined and an overview of the language and the features it supports is provided. The environment of execution of the IPCC++ language model is described, along with the components of the model used to establish the IPCC++ environment.

IPCC++ supports both the centralized and distributed memory models. Each memory model is defined along with the IPCC++ components necessary to support interprocess communication for its corresponding memory model. The centralized memory model uses the monitor structure and the condition variable of concurrency to facilitate centralized interprocess communication. In addition, the distributed memory model uses the socket structure along with a message passing protocol to support distributed interprocess communication. The producer consumer concurrency problem is presented with the corresponding IPCC++ solution designed for a centralized memory model. The dining philosopher concurrency problem is presented with the corresponding IPCC++ solution designed for a distributed memory model.

The language design and concurrency features of IPCC++ are discussed and compared with current research efforts that introduce concurrency to C++ supporting centralized and distributed memory models. A description of the IPCC++ implementation model, preprocessing design, and research contributions of the IPCC++ language is provided.

1. INTRODUCTION

The increasing demand for reliable software applications that support cost efficient computer systems has caused two different programming language paradigms to collide. On one side, the demanding need for modularization, code sharing, and code reuse which supports reliable software development has been a factor in making the object-oriented paradigm a paradigm of choice among many software engineers. On the other side, the growing trend toward distributed computer systems has resulted in the concurrency paradigm becoming an important research area. The combination of these two paradigms is of current research interest due to the potential benefits for reliability and efficiency.

The object-oriented paradigm supports reliable software development while the concurrent programming paradigm supports the use of efficient computer systems. An object-oriented language is an ideal vehicle for the development of concurrent languages [Chan93]. No previous language has successfully extended this paradigm to handle both concurrency and distribution without sacrificing much of its support for reuse. Object-oriented languages that handle concurrency or distribution, such as POOL[Amer87] and Emerald[Blac87], do not support an inheritance mechanism permitting the 'reuse' of implementations, while those designed to enhance the reuse advantages of the object-oriented approach, such as Eiffel[Meye88] and C++[Stro86], do not tackle concurrency or distribution [Atki91]. Therefore, there exists a need for a

language model that fully supports both the concurrency and object-oriented paradigms in order to achieve benefits offered by both paradigms.

This research encapsulates the object-oriented and concurrency paradigms into the language model, InterProcess Communication with C++ (IPCC++). IPCC++ utilizes the inheritance feature of C++ and supports both centralized and distributed processing.

1.1 Object-Oriented Language Model

The rapidly expanding discipline of object-oriented programming is the most promising vehicle for reuse that has emerged in recent years. Object-oriented languages allow objects to be treated as 'first-class citizens'. Although objects are structuring units, encapsulating states and operations, they are also treated as basic data items which can be instantiated dynamically and passed as parameters to operations. Therefore, objects serve both as the operands acted upon by operations, and as the operators performing operations on others. This dual role is one of the most confusing aspects of object-oriented programming for newcomers, but is the basis of much of its power [Atki91].

The object-oriented programming paradigm includes three fundamental characteristics. One characteristic is data abstraction (an object) which focuses on data structures and then adds functionality or processing capability to those structures. A data structure definition and its defined methods are packaged together in some syntactic structure, in

which the structural definition and process implementation are hidden from the program units that use it. The second characteristic is dynamic type binding which allows abstract data types to be generic; therefore, code is not rewritten solely due to the types of objects which they operate on. The third characteristic is inheritance, which is a method of sharing code among users.

Inheritance is an idea first introduced in Simula and is generally regarded as one of the most important features of object-oriented programming. Essentially it is a mechanism for sharing knowledge in systems by enabling new classes to reuse part of the declarations of others. Inheritance is a relationship only between classes. When using inheritance, a new class (subclass or derived class) may be defined as a specialization of another class (super class or base class) by inheriting its methods and instance variables and adding to them. A derived class usually conforms to its base class since it normally inherits all of the methods in its interface [Atki91]. Two different types of inheritance are single inheritance and multiple inheritance (both supported by C++). Single inheritance occurs when a class has only one base class. Multiple inheritance occurs when a class has more than one base class which is a quick means of combining the functionality of several classes. Inheritance can also be static or dynamic. Static inheritance means all information sharing is fixed at compile time (C++ supports static inheritance). Dynamic inheritance means all objects are accessible at run-time and

method selection in response to a method invocation is located dynamically by the run-time system [Atki91]. The IPCC++ model utilizes the inheritance feature of C++ and provides base class definitions representing the primitives of concurrency which are used to derive and modify the concurrency primitives for a particular application.

Language features necessary to develop object-oriented programs which support reliable software applications include modularity, abstraction, encapsulation, inheritance, and polymorphism. Modularity refers to splitting a design into smaller, more manageable components that can be tackled separately. Abstraction refers to separating the concerns for what a particular component does from how it does it. Encapsulation or information hiding refers to concealing details irrelevant at a particular level of abstraction [Atki91]. Inheritance and polymorphism are language features that support reduction in design and code reuse. An object-oriented language is viewed as a collection of modular units, that is, objects that encapsulate their data by offering methods to access the data. The primitives of the object-oriented language C++ that support the object-oriented programming paradigm include: class construct, objects, inheritance, and polymorphism.

The class construct is an extension of the idea of struct in the C language. A class provides a means for implementing a user-defined data type and associated functions and operators, that is, an abstract data type [Pohl89]. The class

construct supports encapsulation in that each class represents a unique set of objects and the methods available to create, manipulate, and destroy such objects [Borl90]. Objects refer to instances of a class. A class defines a data type and an object is a variable of that data type. Inheritance refers to the deriving of a new class from one or more existing classes called base classes. The base class can be added to or altered to define the derived class. This forms a hierarchy of related data types that can be defined to share code. [Pohl89] Access privileges of public, private, and protected are used to define visibility. Polymorphism refers to the ability to sort out inherited types [Davi90]. It is a powerful feature of C++ that supports run-time binding when used in conjunction with the key word virtual.

1.2 Concurrent Programming Language Model

Language features necessary to develop parallel software applications include the concept of process, specification of concurrent execution, synchronization, mutual exclusion, and interprocess communication. The concept of process refers to the existence of an executing program [Havi87]. A sequential program represents a single thread of control (the program counter) which begins at the first statement in the process and moves throughout the process as statements are executed. Concurrent programs results in multiple threads of control (processes), one for each constituent process with the processes executing simultaneously within one program [Andr91]. The specification of concurrent execution is the primitive of

concurrency that indicates multiple processes are to execute a specific code block and the construct used to create processes. For example, the UNIX fork() system call splits the execution of a program into two concurrent executions. Establishing constituent processes is normally performed by associating a process type with a code block. When multiple threads of control exist in a program, it creates the need for a synchronization mechanism as well a mechanism to provide mutual exclusion.

A synchronization mechanism is a primitive of concurrency used to suspend and activate the execution of processes. It is used to control access to shared data. Synchronization mechanisms include semaphores, monitors, critical regions, and conditional critical regions. The sending and receiving of messages are primitives of concurrency which represent a synchronization mechanism that supports distributed computing. Mutual exclusion is a concept of concurrent programming which guarantees that only one process is executing a code block (often referred to as a critical section) at a time. A critical section of a process is a code segment in which some shared resource is accessed; therefore, during the execution of a critical section, mutual exclusion must be ensured. There exist numerous primitives to support mutual exclusion, including the monitor structure of concurrent programming languages. Interprocess communication mechanisms facilitate process communication in the absence of shared memory where processes exchange information and synchronize by using

communication commands. Synchronization occurs because a time ordering of events is imposed by the fact that the receipt of a message must be preceded by its sending. Synchronization mechanisms used to support interprocess communication in the absence of shared memory include: signals, pipes, FIFOs (First In First Out named pipes), and sockets. Interprocess communication issues include the naming of sender/receiver processes (direct naming or indirect naming), the buffering of messages (synchronous or asynchronous communication), message length (fixed or variable), and message type (based on programming languages typing system).

In Chapter 2, we present the IPCC++ language model, define its components, and discuss the environment of execution. Chapter 3 presents features of the model that support the centralized memory model. Chapter 4 defines the model components which support the distributed memory model. Chapter 5 contains a discussion on related research and performs an in-depth comparison to languages supporting either the centralized or distributed memory model. Chapter 6 presents the implementation design. Finally, Chapter 7 summarizes the language and states the research contribution as well as the direction of future research.

2. IPCC++ LANGUAGE MODEL

A concurrent object-oriented language facilitates the use of today's technology by supporting object re-use, object sharing, and parallel object execution. In general, two different methods are used to define a concurrent object-oriented language. One method is to introduce concurrency as an integral part of the design, and the other method is to extend a pre-existing language with concurrency features.

The popularity of the object-oriented language C++ and of the UNIX environment has led this research in the direction of extending C++ with concurrency features designed for a UNIX environment. The goal of this work was to develop a concurrent object-oriented programming model that supports concurrency for centralized and distributed memory models while maintaining the high level of abstraction associated with object-oriented languages. The IPCC++ model is an extension of C++ that introduces the concurrency features as base class definitions. The IPCC++ model adheres to the principle of orthogonality and introduces the primitives of concurrency using the primitives of object-oriented languages. The concurrency features, which are encapsulated in objects, support information hiding, polymorphism, and inheritance.

IPCC++ is a concurrent object-oriented language designed for centralized and distributed memory models. It is a natural extension of the C++ that supports concurrent object-oriented programming paradigm. It introduces the following to C++:

- process object,
- mechanism for process instantiation,

* Portions reprinted by copyright permission of Journal of Object-Oriented Programming [Stub95a].

- static and dynamic process declaration,
- monitor object,
- condition object,
- socket object,
- typed message passing communication protocol,
- synchronous and asynchronous communication,
- selective waiting, and
- run-time communication error detection.

The model extends C++ with the power of parallel programming while maintaining the integrity of the C++ design. One advantages is the learning curve associated with grasping the features of concurrent programming for programmers familiar with an object-oriented language is reduced. C++ programmers are familiar with the class related features and can focus their attention on the semantics of concurrent programming, not the syntax. By design, we restricted IPCC++ to inter-object concurrency on the premise that having only one concurrent action within an object reflects encapsulation and the true spirit of C++.

The features of the IPCC++ language model can be categorized into concurrency, process management, and communication protocol. Concurrency refers to the features of concurrency supported by the primitive of concurrency introduced to facilitate mutual exclusion and synchronization of processes for centralized memory models. Process management refers to the features of concurrency supported by the primitives of concurrency introduced to facilitate the specification of concurrent execution and the declaration, activation, and termination of processes within an IPCC++ application. Communication protocol refers to the features of

message passing supported by the primitives of concurrency introduced to facilitate interprocess communication for distributed memory models.

Of the following sections, Section 2.1 gives an overview of the IPCC++ language. In Section 2.2, the environment of execution for the IPCC++ language model is discussed. Section 2.3 defines the process class definition, illustrates its use, and presents IPCC++ support of the client/server paradigm.

2.1 Overview of IPCC++

IPCC++ supports the interprocess communication within the two memory models by providing class definitions that represent and perform the functionality of the supported concurrency primitives. For centralized memory models, the monitor structure and condition variables are introduced as objects and are used as the synchronization mechanism to enforce mutual exclusion. The monitor structure is used because the syntax of a monitor is based on encapsulating data items and the procedures that operate upon them in a single module. Therefore, the monitor is an abstract data type that can be implemented as an IPCC++ class definition. The difference between a monitor class and an ordinary class is that a monitor guarantees mutual exclusion and synchronizes calls to its methods [BenA90]. IPCC++ supports the mutual exclusion and synchronization within a monitor with objects of a condition class. The functions of signal(), wait(), and empty() associated with a monitor structure are supported as methods of the condition class definition. An IPCC++ monitor object

declares a condition object to facilitate the suspension conditions of the shared resource. The special relationship between monitor objects and condition objects is discussed in Section 3.1.

For distributed memory models, the socket object is the language construct used to provide synchronization. Coupling the communication methods with the socket object rather than the process object supports indirect naming. The C++ inheritance features make a socket object declared in a superclass visible to process objects derived from the superclass, thereby supporting indirect naming of the communicating processes. Introducing the low-level UNIX system calls necessary for interprocess communication as objects produces an elegant message passing protocol.

The concept of process is introduced as an object. It is important that a user of IPCC++ understand the concept of a process and the relationship that exists between multiple processes contained within one program. A C++ program has one single sequential process (the program control) that dictates the course of action. However, within an IPCC++ program, a set of sequential processes can be created to operate in parallel. Initially, an IPCC++ program consists of one sequential control that has the capability to create multiple sequential process objects. Each process object created by the initial program control is a descendent of the program control and can create other process objects. Therefore, the relationship of the program control and the newly created processes is a tree

structure which supports object inheritance. The creation of multiple process objects within a program implies the use of a particular environment. The process objects of IPCC++ and the interprocess communication objects are designed for the UNIX operation system.

2.2 Environment

The ongoing popularity of the UNIX environment led this research in the direction of utilizing the interprocess communication mechanisms of the UNIX environment for the implementation of the IPCC++ language model. The UNIX environment is rich in interprocess communication mechanisms for both centralized and distributed memory models. The IPCC++ language model utilizes Parallel Virtual Machine (PVM), as an abstraction of the UNIX interprocess communication mechanisms. PVM is a software system that encapsulates heterogeneous networked parallel and serial computers as one concurrent computational resource [Begu94]. PVM is designed to work with the C and Fortran programming languages, thereby supporting C++. PVM is popular in the distributed computing environments because it represents a high level abstraction of UNIX system calls and it is availability as public domain software. PVM offers tools necessary to create processes, assign processes to specific nodes (processors), facilitate interprocess communication, as well as many other features used by the IPCC++ environment.

2.3 Environment Class Definition

A class IPCC_ENVIRONMENT is used to establish the IPCC++ execution time environment. The data consists of the integer field ParentTID which specifies the PVM task identification number associated with the parent of this process. It is used to distinguish the initial program control from spawned process object controls. The class constructor enrolls the executing program into the PVM environment and establishes an IPCC++ server process (master server). The master server executes the method Server() of the IPCC_ENVIRONMENT which represents the actions of all process objects within the program. The implementation of the IPCC_ENVIRONMENT::Server() method utilizes features of the PVM system for supporting interprocess communication. The class destructor is responsible for removing process from the PVM environment and killing the master server process. The components of the IPCC_ENVIRONMENT class are depicted in Figure 2.1.

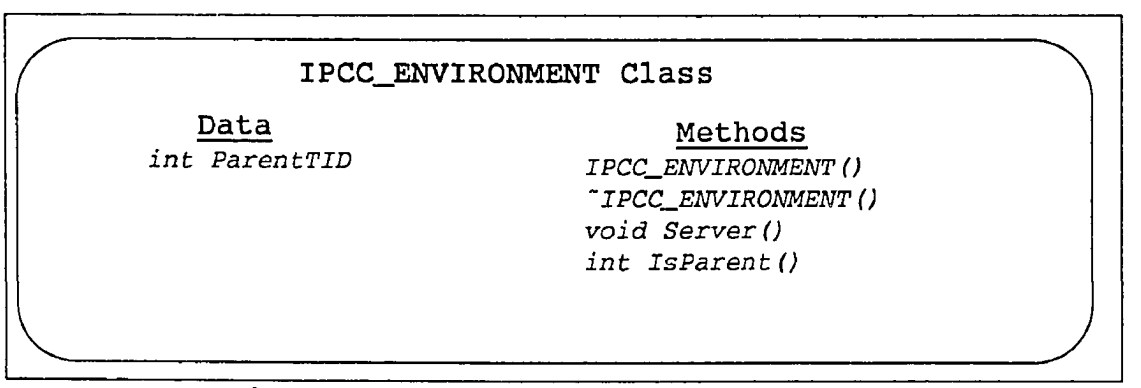


Figure 2.1: IPCC_ENVIRONMENT Class

2.4 Process Manager Class Definition

IPCC_PROCESS_MANAGER is a class definition that provides the process objects with the necessary information to implement

creation and activation of processes. As depicted in Figure 2.2, the IPCC_PROCESS_MANAGER consists of two dynamic arrays

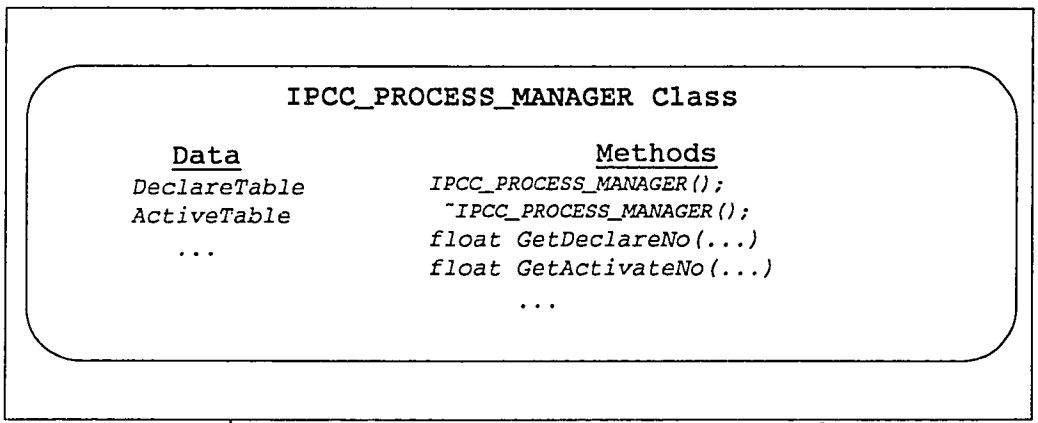


Figure 2.2: IPCC_PROCESS_MANAGER Class

structures, DeclareTable and ActivateTable, which containing information necessary to declare and activate a process object, respectively. The class also offers two methods, GetDeclareNo(...) and GetActivateNo(...), which return information to the invoking process directing the declaration and activation of processes, respectively. The class constructor creates the tables and loads them with the necessary information from a file created during the preprocessing of IPCC++. The class destructor deletes the dynamically allocated memory used by the tables.

2.5 Process Class Definition

The IPCC_PROCESS class definition is designed for a UNIX environment. As depicted in Figure 2.3, the IPCC_PROCESS data structures consists of int ChildTaskID which is the process task identification number (tid) assigned to the child by the PVM environment. The class definition provides methods which

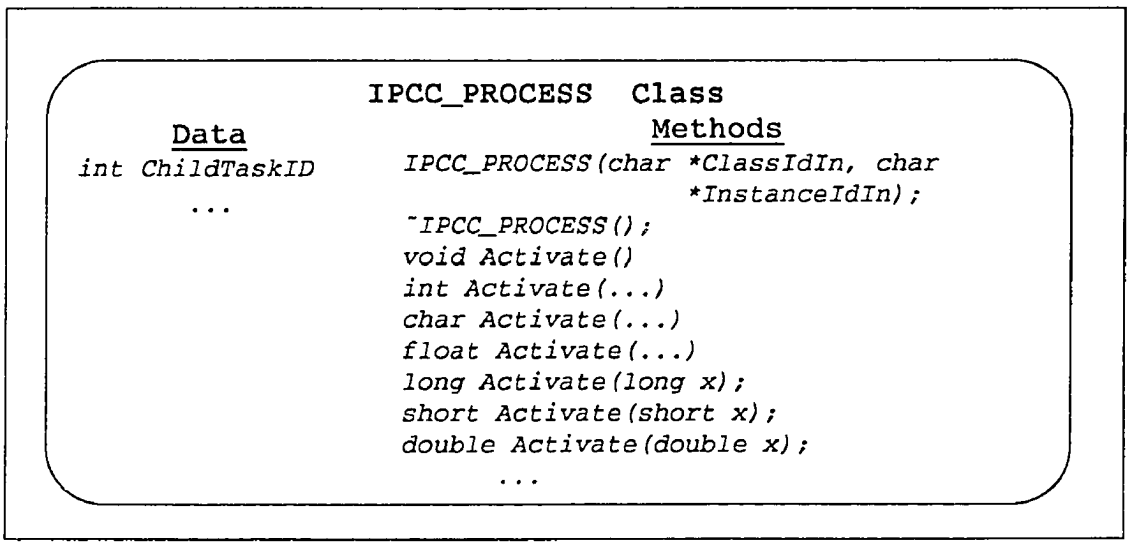
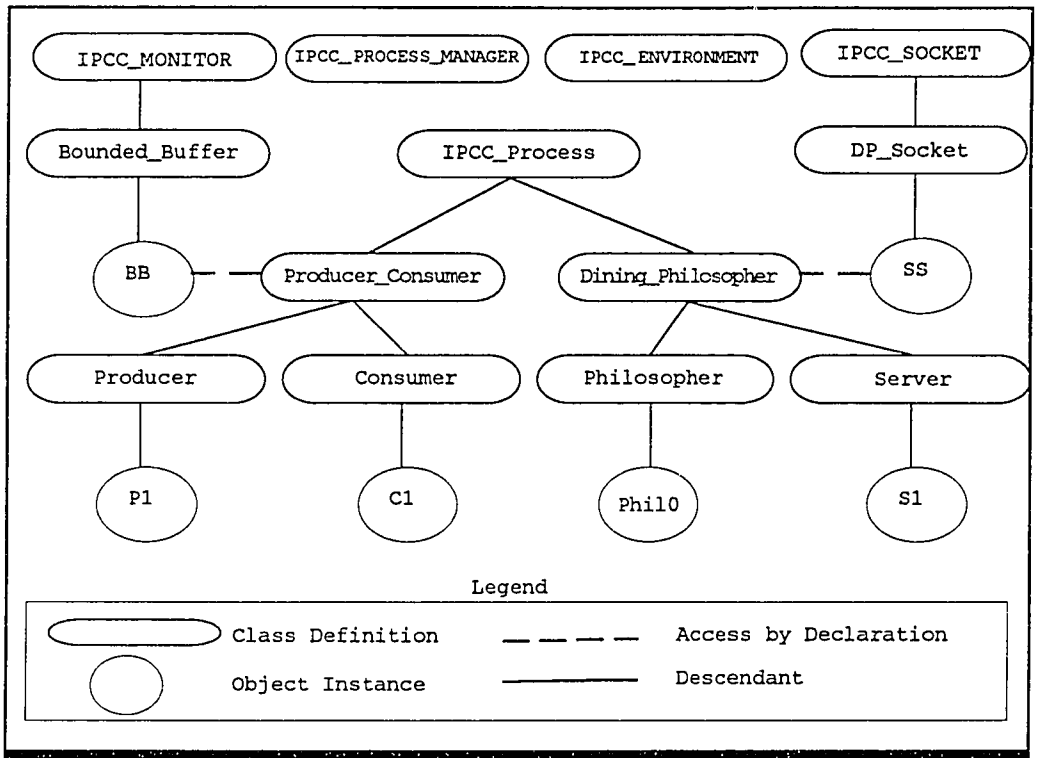


Figure 2.3: IPCC_PROCESS Class

activate the suspended process object by invoking PVM communication commands. If the method to be executed by the process object is to return a value, then the caller of Activate(...) is suspended until the process object completes the method and returns a value. There exist an activation method for each of the standard C++ types. The class constructor accesses the GetDeclareNo() method and invokes a PVM command to spawn a child process. This results in the creation of a suspended process, possibly on a different node in the network.

As illustrated in Figure 2.4, the C++ inheritance feature is used to make a synchronization object, either a monitor or socket object, accessible to a process objects. P1 and C1 are objects declared of the type Producer and Consumer, respectively. The Producer and Consumer class definitions are derived public from the class Producer_Consumer. The Producer_Consumer class declares BB, a monitor object of the



Centralized Memory Model:

```
Class Producer_Consumer : public  IPCC_PROCESS
       {  public:
        Bounded_Buffer BB;
              ...};
Class Producer : public Producer_Consumer { ... };
Class Consumer : public Producer_Consumer { ... };
```

Distributed Memory Model:

```
Class Dining_Philosopher : public  IPCC_PROCESS
       {  public:
        DP_Socket SS;
              ...};
Class Philosopher : public Dining_Philosopher { ... };
Class Server : public Dining_Philosopher  { ... };
```

Figure 2.4: Process Hierarchical Structure & Inheritance

type Bounded_Buffer which is derived public from the IPCC_MONITOR class. Therefore the objects P1 and C1 have access to the monitor object BB. The Producer_Consumer class is derived public from the class IPCC_PROCESS and uses the data and methods of IPCC_PROCESS_MANAGER to create, activate, and terminate process objects.

Phil0 and S1 are objects declared of the type PHILOSOPHER and Server, respectively. The PHILOSOPHER and SERVER class definitions are derived public from the class Dining_Philosopher. The Dining_Philosopher class declares SS, a socket object of the type DP_Socket which is derived public from the IPCC_SOCKET class. Therefore the objects Phil0 and S1 have access to the monitor object SS. The Dining_Philosopher class is derived public from the class IPCC_PROCESS and uses the data and methods of IPCC_PROCESS_MANAGER to create, activate, and terminate process objects.

A process in IPCC++ is used to identify an independent sequential control whose execution may occur in parallel with other processes. A process derived class definition is used to specify different courses of action for different derived process classes. Process objects can access shared resources or work in conjunction by having access to a monitor or socket object or can perform an independent task. All process objects have the ability to create and terminate processes.

A process class definition is static; however, instances of a process class can be created dynamically. The declaration of a process object causes the system to create a suspended

process of the specified process class; the class constructor is responsible for creating the process object. Static or dynamic process declarations exist, and the only distinction to the programmer is the declaration syntax. The ability of IPCC++ to dynamically create processes is necessary to support concurrent servers as well as other parallel programming needs.

Static process declaration causes the constructor associated with the object to invoke a method to create a process which returns a pid. At compile time, the system binds the pid to the object variable. The object name then becomes the handle of the process and is used to access any methods. This results in the creation of a suspended process. Figure 2.5 represent the code segment that creates three suspended processes at compile time. P1 and P2 are handles to PHILOSOPHER process objects while S1 is a handle to a SERVER process object. The handles are used to access the methods of each process.

```
...
PHILOSOPHER  P1(1), P2(100);
SERVER  S1;
...
```

Figure 2.5: Static Process Declaration

Dynamic process declaration allows the user to create process objects as needed during execution. A handle to the dynamically created process is returned and stored in a pointer type variable. This method also results in the creation of a suspended process. Figure 2.6 represent the code segment that causes one suspended process to be created during program execution. The Philosopher process passes the parameter to its

constructor. The handle, Phil, is a pointer to an object of the type Philosopher.

```
...
Philosopher* Phil = new  Philosopher(10);
...
```

Figure 2.6: Dynamic Process Declaration

All process object declarations are preprocessed to invoke the IPCC_Process class constructor with the appropriate arguments to causes the invocation of a PVM command to create a suspended child process on a particular node. All processes created are children of the process that created them, typically main() referred to as P_0. As illustrated in Figure 2.7, the activation statement or process instantiation statement of a process is simply a call to one of its methods. All process object method invocations are preprocessed into

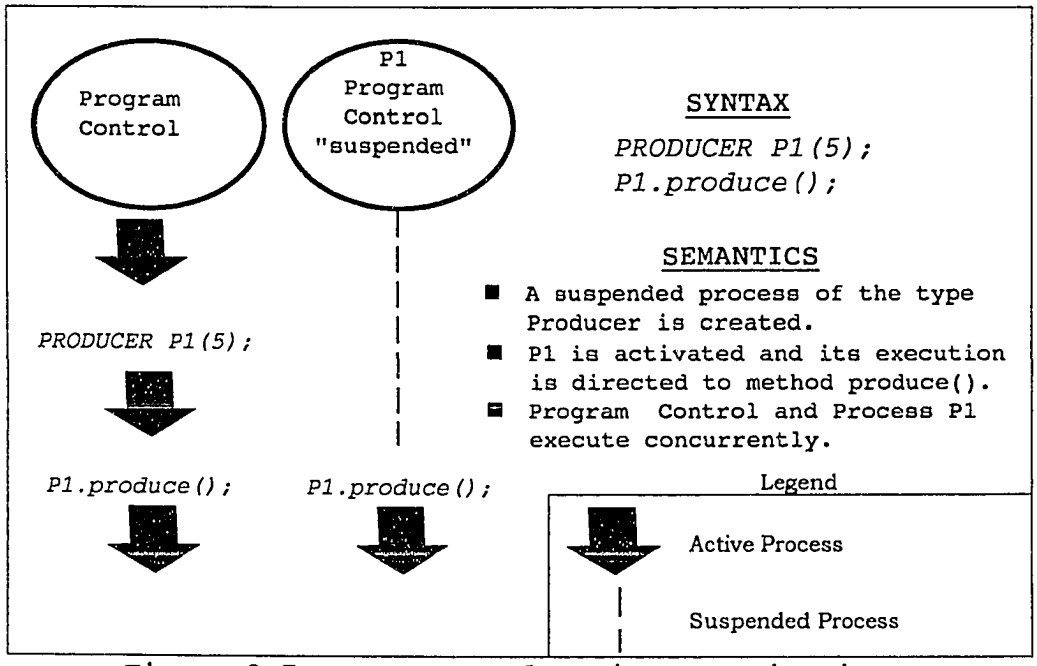


Figure 2.7: Process Declaration & Activation

calls to the activation method of IPCC_PROCESS which invoke PVM commands to activate a suspended child process on a particular node and direct its action to the method expressed in the statement. The preprocessing of IPCC++ to C++ implicitly controls suspension based on whether the child, P_i, is to return a value to P_0 or not. The preprocessing of IPCC++ to C++ determines the necessary communication between P_0 and P_i and achieves synchronization if necessary.

A parent process will not terminate until all of its children have terminated. It is possible for children of one process to become the parent of other processes. The process activation technique described above results in the creation of a tree structure of processes, thus supporting the inheritance feature of C++. A process object cannot execute independently within the same scope level as its parent process. Its execution must be directed to one of its methods; this feature supports inter-object concurrency. When a process completes the execution of the activating method, it will either terminate immediately or wait for all of its children processes to terminate and then subsequently terminate. Figure 2.8 formally expresses the termination condition for active processes. This condition eliminates the appearance of (orphans) process objects whose parents in the hierarchical structure have terminated.

$$\forall P_0 : P_0 \text{ terminates} \Longleftrightarrow \nexists P_i \, [P_i \text{ child of } P_0 \textit{ and } P_i \text{ active}]$$

Figure 2.8: Process Termination Condition

Process objects can be classified as clients, servers, or inactive for a socket object. The programmer can define a select_wait() method in the process class definition designed to be a server process, actually any method name can be employed. A server process is viewed as a process issuing a communication response (receive) from a common socket object. The select_wait() method supports the client/server paradigm. The process invoking the select_wait() method becomes the server process of a visible socket object. Its life is spent serving clients. A client process is viewed as a process issuing a communication request (send) to a common socket object. When a process object invokes its select_wait() method, its execution is directed to the select_wait() method. Using the PVM system, the server will check for pending communication requests directed to the visible socket object.

The PVM interface is transparent to the IPCC++ programmer. The IPCC++ language model supports these functions implicitly. The encapsulation of PVM into IPCC++ objects supports the principle of orthogonality and simplifies the coding effort of the programmer. The communication requests directed to a socket object can appear in a conditional statement; thus, selective waiting is achieved. The first condition that evaluates to true is executed. If a conditional statement evaluates to true, then the communication occurs and the body of the conditional statement is executed. IPCC++ employs a deterministic method for evaluating the condition statements. This is not as powerful or as fair as the original

nondeterministic notation of Dijkstra [Dijk76]; however, it gives the IPCC++ programmer power of expression by ordering the condition statements.

The model represents a natural extension of C++. It is unique in that it introduces concurrency to C++ utilizing the primitives of object-oriented programming. The class definition encapsulates the primitives of concurrency and builds a concurrent object-oriented language based on complete objects while adhering to the principle of orthogonality, and exploitation of inheritance features of C++. The term 'complete objects' refers to the encapsulation of concurrency features into objects by hiding the implementation from the programmer and extending the language solely with objects. The 'principle of orthogonality' refers to the using of a set of primitive constructs that are combined in a relatively limited number of ways to build the control and data structures of the language [Sebe89]. IPCC++ uses the object construct to realize concurrency in C++, thereby adhering to the principle of orthogonality. The 'exploitation of inheritance features' refers to the utilization of the hierarchical structure associated with C++ classes to reduce replication in design and code. IPCC++ also supports desirable features such as explicit concurrency, inter-object concurrency, static and dynamic process creation, synchronous and asynchronous communications, uncoupled process declaration and activation, typed message passing system, selective waiting and run-time error communication detection. The IPCC++ extension of C++ suggests

a simple and natural merge which supports reliable software development for efficient computer systems.

In summary, the language model and environment support centralized and distributed memory models. The process object is used as the vehicle to support the specification of concurrent execution. This component supports multiple program controls executing in either a single or distributed address space. Process objects utilize the monitor and socket objects to facilitate mutual exclusion, synchronization, and interprocess communication for the centralized and distributed memory models, respectively. Chapter 3 defines the basic meaning of the synchronization objects used to support the centralized memory model, the concurrency features introduced, and any special relationships that exist among the objects.

3. CENTRALIZED MEMORY MODEL

A centralized memory model constitutes a single serial machine where concurrency is achieve through multitasking. In IPCC++, interprocess communication and synchronization are achieved by IPCC++ class definitions that support the monitor structure and condition variable primitives of concurrency.

3.1 Monitor Class Definition

IPCC_Monitor is a class definition that offers concurrent access to its private data via a set of methods. The monitor object is essentially an entry queue. The processes waiting for initial entry have a priority of 0 and are scheduled FIFO. Waiting signalers are also placed in the entry queue and have a priority of 1 and are scheduled FIFO. Therefore, waiting signalers have priority over entry processes. IPCC++ programs use the IPCC_Monitor class to derive monitor class definitions for specific applications.

A monitor derived class definition is used to specifically define the monitor class to protect shared resources. It is required that all monitor derived class definitions contain at least one condition object. This requirement is necessary to provide for process synchronization. Figure 3.1 creates a derived class of monitor, Bounded_Buffer. The condition object requirement is satisfied with the declaration of notempty and

```
class Bounded_Buffer : public IPCC_MONITOR
    {  IPCC_Condition notempty, notfull;
                ...                    };
```

Figure 3.1: Derived Monitor Class and Condition Object

* Portions reprinted by copyright permission of Journal of Object-Oriented Programming [Stub95a].

notfull as part of its private data. As previously illustrated in Figure 2.4, the inheritance feature of C++ is used to derive public the subclass Bounded_Buffer from IPCC_Monitor. Figure 3.2 illustrates the special relationship between the monitor object and its condition objects. Figure 3.2 is a conceptual image of the IPCC++ code sample of Figure 3.1. It illustrates the special relationship between a monitor object and condition objects. The condition objects, notfull and notempty, are viewed as part of the monitor object because they have access to the monitors private data, that is the pid's suspended in the entry queue. The private data of the monitor object represents the shared resource and is accessible through methods defined within the derived monitor class definition Bounded_Buffer.

A monitor is an object that remains active throughout the execution of a program until the destructor of the monitor is invoked by exiting a scope level. A single monitor object can be shared by multiple processes by using the inheritance feature of C++. The utilization of the C++ inheritance feature to allow processes to access a common monitor object is also depicted in Figure 2.4. Monitor objects rely upon a condition object for process synchronization and mutual exclusion.

3.2 Condition Class Definition

The IPCC++ environment provides the IPCC_CONDITION class definition that offers methods necessary to perform synchronization of processes and guarantee mutual exclusion within a monitor object. An object declared of the type IPCC_CONDITION

is referred to as a condition object. A condition object has access to the private entry queue of a monitor as shown in Figure 3.2 and is used within the monitor to control process synchronization. A condition object is declared of the class

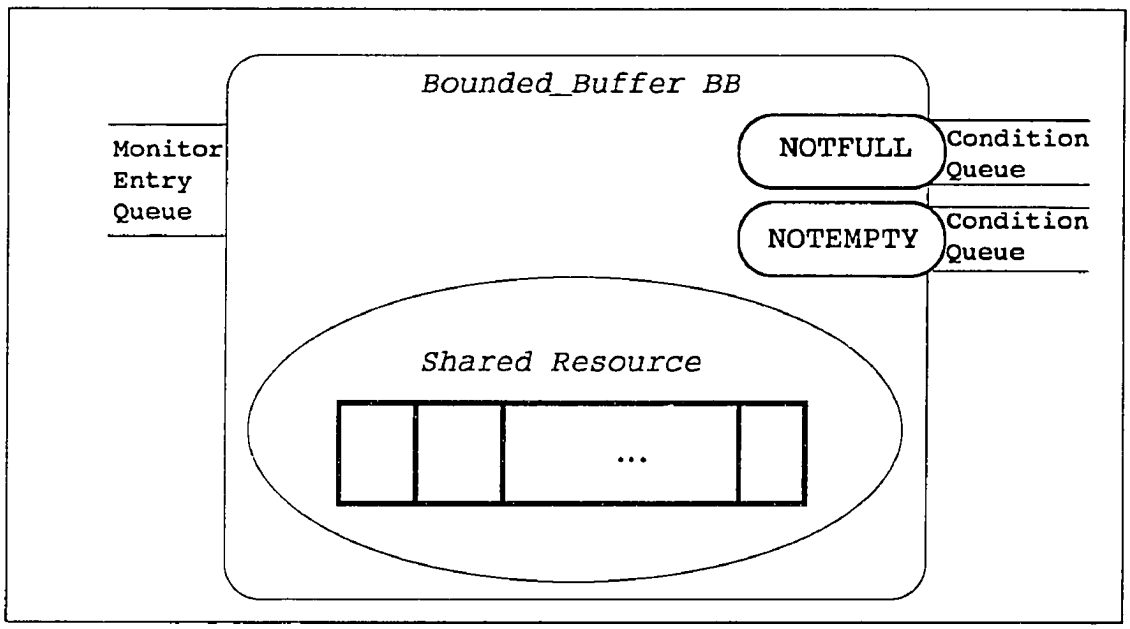


Figure 3.2: Monitor and Condition Object Relationship

type IPCC_CONDITION and consists of a queue of processor id's, pid, as its private data. The IPCC_CONDITION class definition has two constructors; one constructor is for creating a simple condition object, and the other constructor is for creating an array structure of condition objects, as depicted in Figure 3.3. All condition objects have access to two queues: Monitor Entry Queue and Condition Queue. Access to the Monitor Entry Queue is provided through the C++ friend declaration. This queue contains the pids of processes suspended and waiting for access to the monitor. Access to the Condition Queue is provided through the declaration of the condition object.

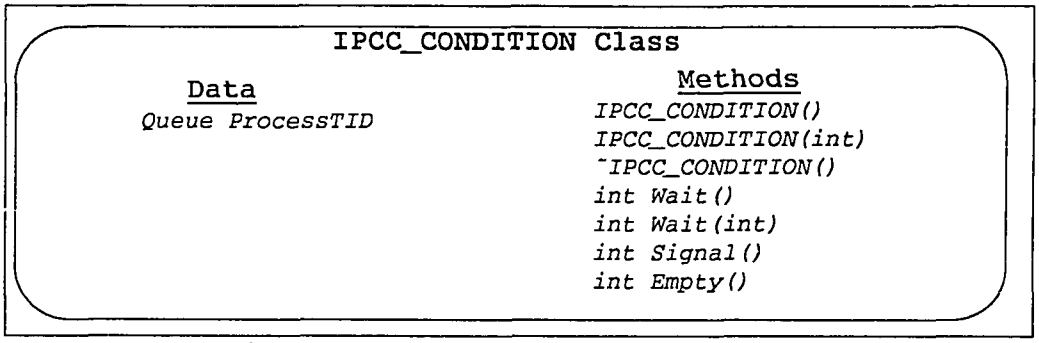


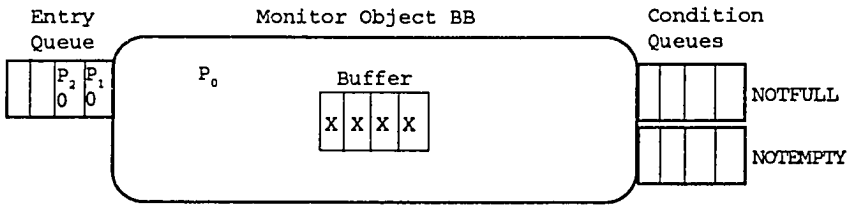
Figure 3.3: IPCC_CONDITION Class

This queue contains the pids of processes suspended by the wait() method of the condition object.

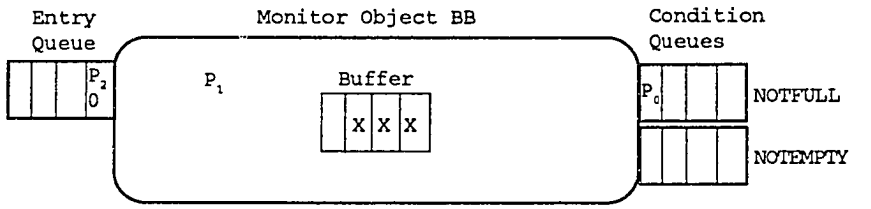
The IPCC_CONDITION class definition provides methods of wait(), signal(), and empty() to access the Monitor Entry Queue and Condition Queue. The functionality of these methods is depicted in Figure 3.4 and is as follows:

- Wait(Void) - The wait() method suspends the caller and places it at the end of the Condition Queue.
- Wait(int) - The wait(int) method suspends the calling process and places it in the Condition Queue at the position specified by the integer parameter. This supports user specified priority in the Condition Queue.
- Signal(Void) - The signal() method performs actions based on the status of the Condition Queue and Monitor Queue and is discussed below.
- Int Empty(Void) - The empty() method is used to see if the Condition Queue is empty. If it is empty then 1 is returned, else 0.

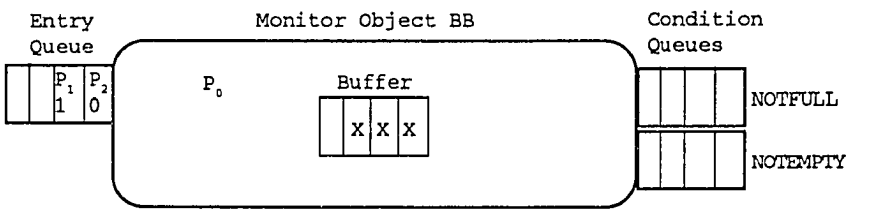
The methods are implemented using the PVM software system.



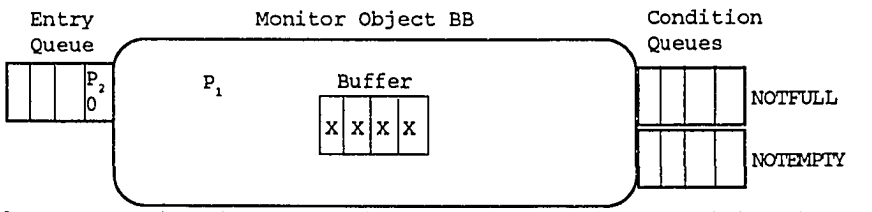
P_0 : Process Allocated the Monitor Object by calling BB.Deposit()
It executes NOTFULL.wait()
P_1,P_2 : Processes wanting access to the the Monitor Object by calling BB.Remove()



P_0 : Suspended to NOTFULL QUEUE,
waiting for the Buffer to be not full
P_1 : Process Allocated the Monitor Object by calling BB.Remove()
It executes NOTFULL.signal()



P_0 : Removed from the Condition Queue by the NOTFULL.signal()method.
P_1 : Placed in the Entry Queue with a priority of 1 by the NOTFULL.signal() method.



P_0 : Completes executing the BB.Deposit() method and prior to exiting the Monitor, executes a NOTEMPTY.signal().
P_1 : Process Allocated the Monitor Object because it has the highest priority of processes in the Entry Queue.

Figure 3.4: Process Suspension & Priority of the Monitor and Condition Object Queues

A condition derived class definition is rarely ever used; however, the ability to derive classes from the IPCC_CONDITION class still exists. It is possible to redefine the synchronization methods of the IPCC_CONDITION class definition, but it requires knowledge of both the PVM software system and the internal design of the IPCC++ language model. Figure 3.5 illustrates the ability to derived classes from the IPCC_CONDITION class definition.

```
Class NewCondition : public IPCC_CONDITION
        { ... };
```

Figure 3.5: Derived Condition Class

As previously stated, the IPCC_CONDITION class definition consists of two constructors; one to define simple condition objects, and one to define an array structure of condition objects. Figure 3.6 illustrates the use of the two constructors. Two simple condition objects named *notfull* and *notempty* and one array of condition objects named *OktoEat are declared*. Each of the condition objects declared, has its own

```
Class Bounded_Buffer : IPCC_MONITOR
          {  Condition notfull, notempty;
             ...  };
Class Fork_Resource : IPCC_MONITOR
          { Condition OktoEat(F);
             ...  };
```

Figure 3.6: Condition Class Constructors

queue for suspended processes. The dimension of the array, F, is passed as a parameter and causes the correct constructor to be invoked. Therefore, F condition objects are created and are accessed as array elements, i.e. OktoEat[i] for 0<=i<=F-1. The advantage of introducing the condition variables as objects refers to the elimination of unnecessary process state transitions by using separate condition queues.

The monitor structure of IPCC++ is a natural starting point used to extend C++ with the capability for interprocess communication and synchronization. It is an abstract data type and readily adaptable to the class construct of C++. The class definitions of IPCC_Monitor and IPCC_CONDITION introduce features of concurrency necessary to support interprocess communication for the centralized memory model. With these two class definitions and IPCC_Process objects, IPCC++ supports concurrent object-oriented programming for centralized memory models while maintaining a high level of abstraction common to object-oriented programming languages.

3.3 Producer Consumer Problem & IPCC++ Solution

The producer consumer problem represents a classic concurrent programming problem. Two types of processes coordinate their actions of producing and consuming by accessing a shared buffer. A producer is a process that spends its life producing products and depositing them in a shared buffer. A consumer is a process that spends its life removing items from the shared buffer and consuming them. Therefore, the buffer is a shared resource of the producer and consumer.

The solution described below uses the monitor object and condition object to synchronize the actions of the producer and consumer process objects. The monitor and condition objects are the interprocess communication and synchronization objects used by the producer and consumer process objects. This solution, which assumes shared-memory, is designed for a centralized memory model. The suspension rules are defined as follows:

- Producers must suspend if the buffer is full,
- Consumers must suspend if the buffer is empty [BenA90].

The IPCC++ code in Section 3.4 solves the producer consumer problem defined above. The class Bounded_Buffer is derived public from IPCC_Monitor. It defines a shared buffer B, methods of Deposit and Remove used to access the shared buffer, and condition objects of notfull and notempty to implement the suspension rules. The class Producer_Consumer is derived public from IPCC_Process.

It declares a monitor object BB of the type Bounded_Buffer. The class Producer and Consumer are derived public from Producer_Consumer. Utilizing C++ inheritance features, process objects of the type Producer or Consumer have access to the monitor object BB. The Producer class definition defines a method of produce() along with data that represents the product being produced. The produce() method is an infinite loop that results in the production of an item and the depositing of the item into the shared buffer, Buf. The item is placed in the Buf by accessing the Deposit method of the

monitor object BB. The Consumer class definition defines a method of consume() along with data that represents the product being consumed. The consume() method is an infinite loop that results in the removal of the item from Buf and the consumption the item. The item is removed from Buf by accessing the Remove method of the monitor object BB.

The main() procedure statically declares and activates two producer process objects: p1 and p2 and two consumer objects c1 and c2. The producer process objects are activated by each accessing their method produce() and the consumer process objects are activated by each accessing their method consume().

3.4 IPCC++ Code Solution to Producer Consumer Problem

This section is the complete IPCC++ solution to the producer consumer problem with a conceptual illustration of the object relationships depicted in Figure 3.7.

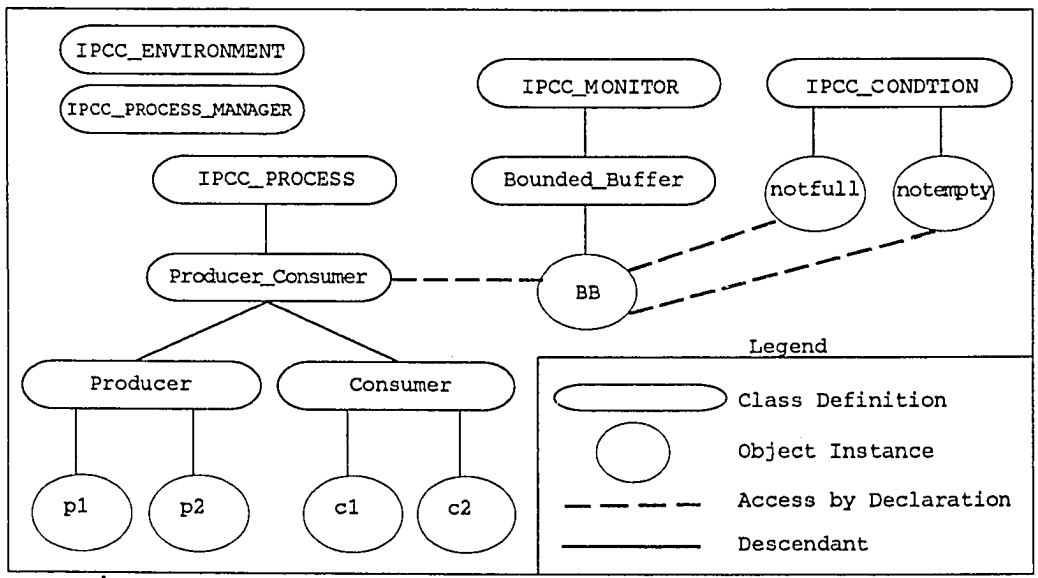


Figure 3.7: Producer Consumer Object Relationships

```
#define BNUM 50
class Bounded_Buffer : public IPCC_MONTIOR
  {  Condition notempty, notfull;
     int* Buf[];
     int in, out, count, N;
     public:
      Bounded_Buffer(int num)
       {  N=num;
          Buf = new int[N];
          out = 0; in = 0; count = 0;}
      Bounded_Buffer()
       {  N=BNUM;
          Buf = new int[N];
          out = 0; in = 0; count = 0;}
      ~Bounded_Buffer
       {  delete Buf;}
      void Deposit(int item)
       {  if (count==N) notfull.wait();
          Buf[in] = item;
          in = (in+1)%N;
          count++;
          notempty.signal();}
      int Remove(void)
       {  if (count==0) notempty.wait();
          int item = Buf[out];
          out = (out+1)%N;
          count--;
```

```
            notfull.signal();
            return(item);}
 };
class Producer_Consumer : IPCC_PROCESS
 {   public:
       Bounded_Buffer BB();
      ~Producer_Consumer()
       { delete BB; }
 };
class Producer : Producer_Consumer
 {   int product;
     public:
      Producer(int start) {   product = start;}
      void produce(void);
 };
     void Producer::produce()
       { while (1)
         { BB.Deposit(product++);}}
class Consumer : Producer_Consumer
 {   int item;
     public:
      void consume(void);
 };
     void Consumer::consume()
       { while (1)
         { item = BB.Remove();}}
```

```
main()
 {   Producer p1(1), p2(20);
     Consumer c1, c2;
     p1.produce();
     p2.produce();
     c1.consume();
     c2.consume();
}
```

This solution displays the ease of using IPCC++ to solve concurrency problems designed for the centralized memory model. The following contains a summary of the features of IPCC++ which support the centralized memory model.

3.5 Summary of IPCC++ Centralized Memory Model

In summary, the components defined above extend C++ with the ability to perform UNIX interprocess communication based on shared memory. A process class definition is provided to support the creation, activation, and termination of process objects. It is used to create multiple program controls (processes) within a single address space. The process class supports C++ inheritance and process objects form a hierarchical structure. Either static or dynamic declaration of a process object is supported and results in the creation of a suspended process of the declared type. An invocation of a process method results in the activation of the suspended process. Therefore, IPCC++ supports separate declaration and activation of process objects. Inter-object concurrency is

provided because each new process is encapsulated in a process object and only one control can exist within an object.

The monitor class definition is used to provide C++ with a language construct that guarantees mutual exclusion and protection of a shared resource. Monitor objects form a hierarchical structure and support inheritance. Synchronization within the monitor objects is performed by a condition object. Each monitor object declares condition objects to perform synchronization which results in condition objects being encapsulated within monitor objects. The condition object provides methods of *wait()*, *signal()* and *empty()* which perform the process synchronization and supports synchronous and asynchronous communication. *IPCC++* adheres to the principle of orthogonality and extends C++ solely with objects. *IPCC++* represents a natural extension of C++ which supports concurrency by utilizing the object-oriented paradigm.

In Chapter 4, we describe the IPCC++ features specific to the distributed memory model, namely the socket object and its support of communication configuration and the client/server paradigm with selective waiting.

4. DISTRIBUTED MEMORY MODEL

A distributed memory model constitutes a collection of independent computers interconnected with a network protocol. A distributed computation is a collection of processes on two or more independent computers (nodes in the network), which need to communicate to achieve some objective [Maek87]. In IPCC++, interprocess communication and synchronization for distributed memory models are achieved with the socket structure which has been introduced in IPCC++ as a class definition.

4.1 Socket Class Definition

IPCC_SOCKET is the class definition that offers an application program interface necessary to support message passing interprocess communication. The IPCC_SOCKET class does not represent the communication endpoint that one usually associates with the term socket, but rather an abstraction of the underlying PVM software system which supports socket communication endpoints.

As shown in Figure 4.1, the IPCC_SOCKET class consists of the following data: GroupID, SocketMode, SocketType, and SocketDirection. The GroupID is used to support the PVM named group facility. It represents the instance of the socket object which is used to form a group of communicating processes. The integer SocketMode is used to determine whether a socket object supports synchronous or asynchronous communication using asynchronous communication as the default. The integer SocketType is used to determine whether a process object

* Portions reprinted by copyright permission of Proceedings of Nineteenth Annual International Computer Software and Applications Conference [Stub95b] and Journal of Object-Oriented Programming [Stub95c].

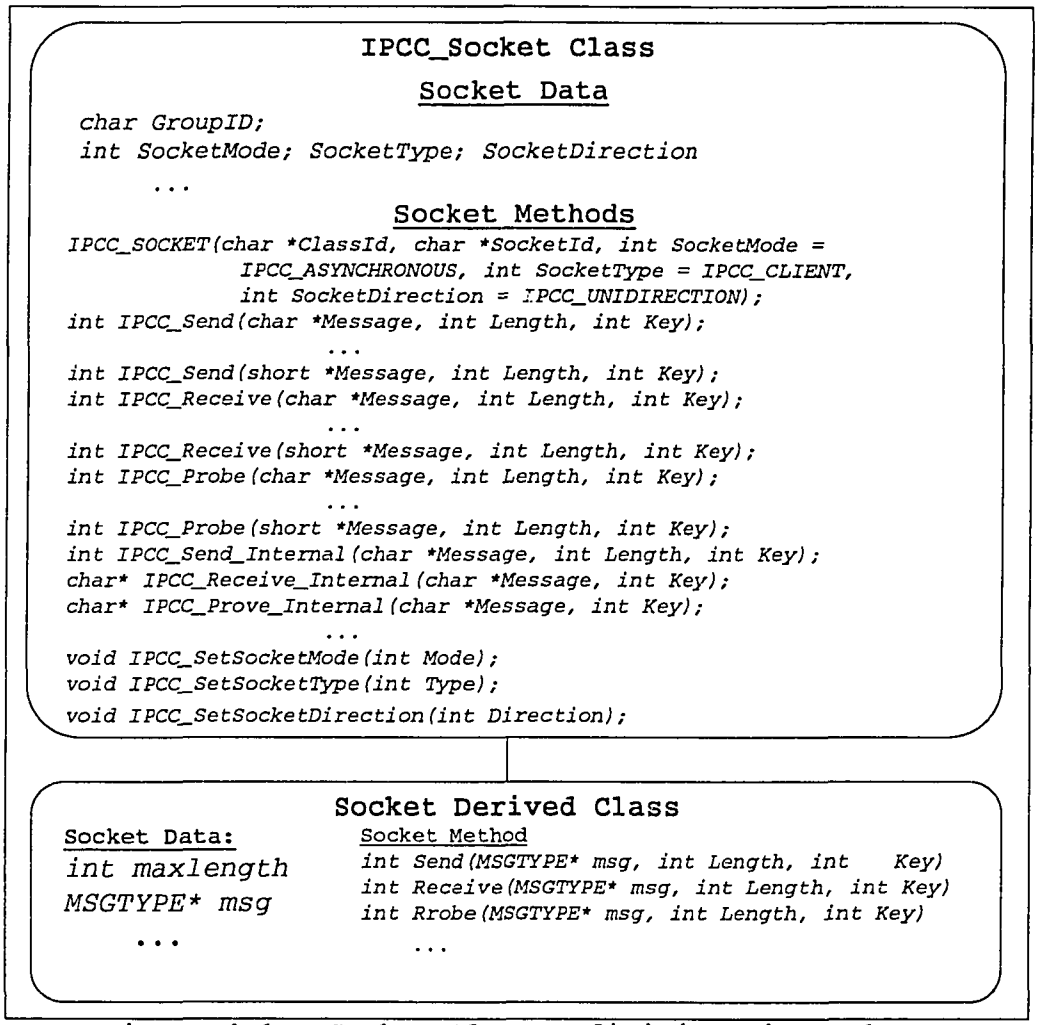


Figure 4.1: Socket Class Definition Hierarchy

which uses the socket is a client, server, or inactive, defaulting to inactive. The integer SocketDirection is used to determine whether a socket object supports unidirectional or bidirectional flow of data using unidirectional as the default.

Socket methods support the sending and receiving of messages between process objects and are referred to as communication methods. The communication methods include

Socket::Send(...), Socket::Receive(...), and Socket::Probe(...). All communication methods return integer values based on the status of communication. This design allows Socket::Send() and Socket::Receive() methods to be components of a conditional expression. If the conditional needs to express local constraints based on the contents of the message received, then a Socket::Probe() can be employed. The Socket::Probe() methods peeks into the contents of the message, prior to actually receiving it (removing it from the receive buffer). This results in a pseudo receive and gives the programmer the capability to test local constraints relevant to the data.

The socket methods facilitate selective waiting and run-time communication error checking. Figure 4.2 defines the

IPCC++ Code Segment	*Evaluation of Expression*
int i=SS.Send(...);	*i=0: communication not pending*
...	*i=1: communication successful*
	i=-1: communication error
if (i=SS.Receive(...)==1)	*i=0: communication not pending*
...	*i=1: communication successful*
	i=-1: communication error
if (i=SS.Probe...)==1)	*i=0: communication not pending*
...	*i=1: communication successful*
	i=-1: communication error

Figure 4.2: Socket Send(), Receive(), and Probe() Semantics

values returned from the Socket::Send(), Socket::Receive() and Socket::Probe() method calls. When evaluating the Socket::Send() statement, the value of i evaluates to 1 upon successful completion of the communication, or i evaluates to -1 indicating a communication error. The evaluation of i to 0 which indicating that communication is not pending is not significant to the sender. However, when a Socket::Receive() or Socket::Probe() method is part of a conditional expression, the evaluation of i to 0 indicating no pending communication becomes very significant to the receiver and can be is used to facilitate selective waiting. The ability to perform selective waiting and detect run-time errors is an important feature of IPCC++.

The int Key field associated with communication methods is used to restrict communication based on matching key values. The default value of the key field (-1) is a wildcard match with any other key field. The key is used to support a form of direct naming by allowing a process issuing a send or receive with a particular key value to only be relieved by a receive or send message with a matching key value. Processes form communication groups based on the visibility of a socket object. This features gives flexibility by allowing subrgoups of processes to communicate within a group.

The following discussion describes the Socket::Send(), Socket:: Receive() and Socket::Probe methods of Figure 4.2 in terms of process suspension.

- Socket::Send(MSGTYPE *msg, int msglen, int Key)
  If the SocketMode of the socket object is configured for synchronous communication, then the process invoking the Socket::Send() method is blocked until some other process invokes the Socket::Receive() method. If the SocketMode of the socket object is configured for asynchronous communication, then the process invoking the Socket::Send() method is blocked only if the system buffer associated with the socket is full.
- Socket::Receive(MSGTYPE *msg, int msglen, int Key)
  If the SocketMode of the socket object is configured for synchronous communication, then the process invoking the Socket::Receive() method is blocked until some other process invokes the Socket::Send() method. If the SocketMode of the socket object is configured for asynchronous communication, then the process invoking the Socket::Receive() method is blocked until a message is available.
- Socket::Probe(MSGTYPE *msg, int msglen, int Key)
  The Socket::Probe() method has no affect on blocking of processes regardless of using synchronous or asynchronous communication. It simply peeks at the contents of the message without actually receiving it.

Associating the send, receive, and probe communication commands with the socket object rather than the process object

facilitates indirect naming interprocess communication. Indirect naming is a desirable feature and enhances the expressiveness of IPCC++.

The IPCC_SetSocketMode(), IPCC_SetSocketType(), and IPCC_SetSocketDirection() methods are used to by process objects to configure the socket for a particular communication task.

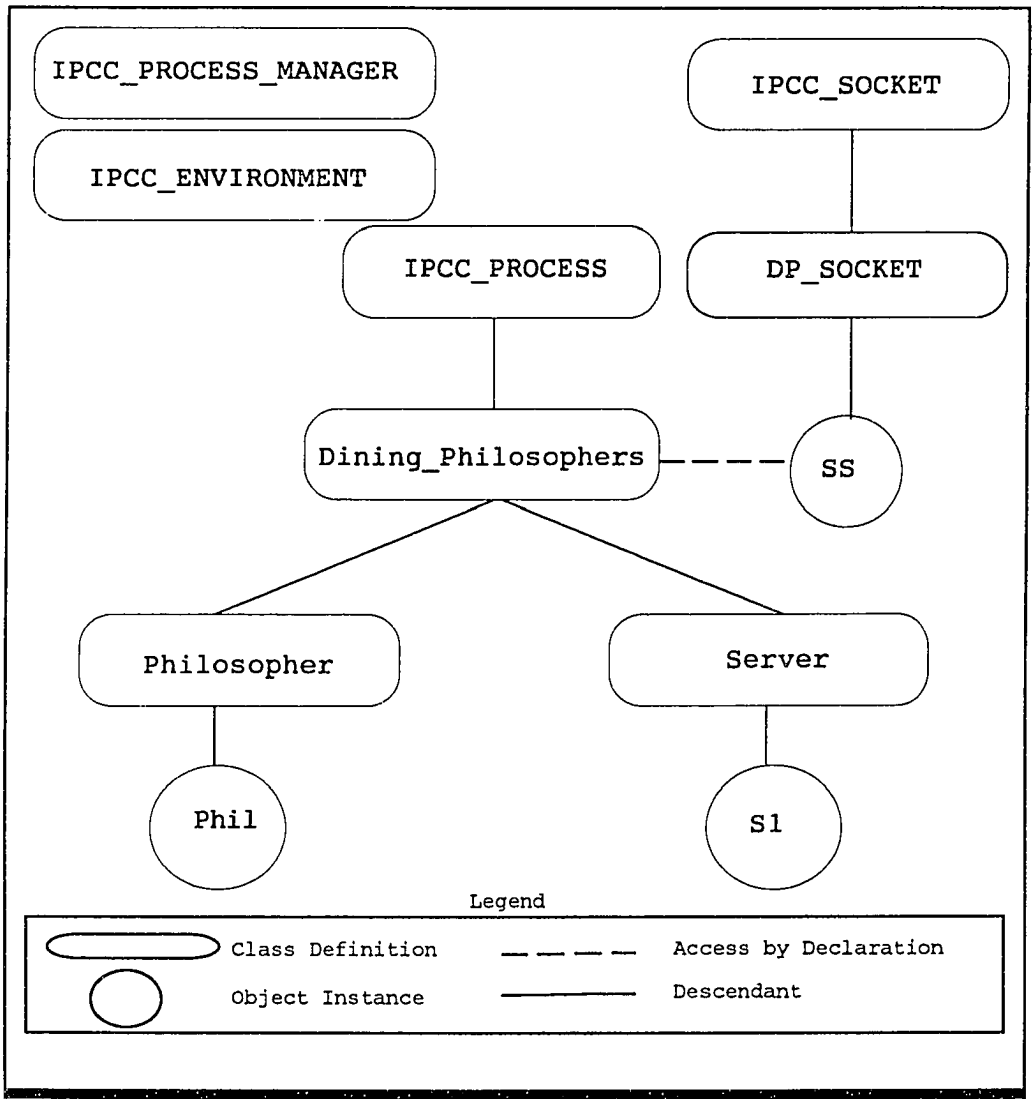
IPCC++ programs use the IPCC_SOCKET class definition to create derived socket class definitions which configure a socket for a specific application. The socket data is used to define a valid message and the type of communication supported by the socket. First, the message type and maximum size simply define the type of message and place a constraint on the

SockeMode=0	*synchronous communication*
SockeMode=1	*asynchronous communication*

Figure 4.3: Synchronous / Asynchronous Configuration

allowable size of a message. Second, the default value of async can be changed in the derived class. Figure 4.3 defines the interpretation of the value of async by the system.

As illustrated in Figure 4.4, the inheritance feature of C++ is used to derive public the subclass DP_Socket from IPCC_SOCKET. The inheritance feature of C++ is used to make the socket object, SS, accessible to the process objects Phil and S1. All objects declared of the type PHILOSOPHER or SERVER have access to the socket object SS. The hierarchical structure of the process objects in Figure 4.4 corresponds directly with the previous discussion in Section 2.5.




```
Class Dining_Philosophers : public IPCC_PROCESS
   {  public:
         DP_Socket SS();
         ...};
Class Philosopher : public Dining_Philosophers { ... };
Class Consumer : public Dining_Philosophers { ... };
main()
{ ...
Philosopher* Phil1 = new Philosopher;
Server* S1 = new Server;
  ... }
```

Figure 4.4: Socket Hierarchical Structure & Inheritance

The declaration of a socket causes the system to create a socket object as specified in the derived class definition. The handle of the socket is visible to communicating processes through the inheritance feature of C++. That is, processes can communicate only if they share a common ancestor object and the ancestor object declared a socket object. The object name then becomes the handle of the socket which is used to access socket methods. Figure 4.5 represents the IPCC++ code segment that declares two socket objects, S1 and S2 which are instances of the DP_Socket class. The parameters of 50, and 0 cause S1 to be configured with a maximum message length of 50 and for synchronous communication. The parameters of 5, and 1 cause S2 to be configured with a maximum message length of 5 and for asynchronous communication.

```
DP_Socket S1(50,0), S2(5,1);
```

Figure 4.5: Socket Declaration

4.2 Selective Waiting

Process objects designed for the client/server paradigm utilize the unique design of the Socket::Send(), Socket::Receive(), and Socket::Probe() methods to perform selective waiting. Figure 4.6 illustrates the use of process objects and socket objects to facilitate selective waiting. In the code segment of Figure 4.6, one process is statically created and 5 are dynamically created. The declaration of SERVER S1 process object results in the creation of one suspended process. The S1 process object is activated by the

```
Class SERVER : DP_Process
   {      ...
     void select_wait();
          ...   };
void SERVER::select_wait()
   {  int i;
      while (1)
     { if (f[I]>=2 &&
          (i=SS.Probe(int *I, int len, "T")!=0))
               if (i==-1)
                { printf("Communication Error");
                  exit(); }
            { i=SS.Receive(int *I, int len, "T");
               if (i==-1)
                { printf("Communication Error");
                  exit(); }
              f[(I+1)%5] = f[(I+1)%5] - 1;
              f[(I-1)%5] = f[(I-1)%5] - 1;
              break; }
       if (i=SS.Receive(int *I, int len, "R")!=0)
         { if (i==-1)
                { printf("Communication Error");
                  exit(); }
           f[(I+1)%5] = f[(I+1)%5] + 1;
           f[(I-1)%5] = f[(I-1)%5] + 1;
           break; }}}

Main()
 {   SERVER S1;
     PHILOSOPHER* Phil0 = new  PHILOSOPHER;
     PHILOSOPHER* Phil1 = new  PHILOSOPHER;
     PHILOSOPHER* Phil2 = new  PHILOSOPHER;
     PHILOSOPHER* Phil3 = new  PHILOSOPHER;
     PHILOSOPHER* Phil4 = new  PHILOSOPHER;
     S1.select_wait();
     Phil0->lifecycle(0);
     Phil1->lifecycle(1);
     Phil2->lifecycle(2);
     Phil3->lifecycle(3);
     Phil4->lifecycle(4);
}
```

Figure 4.6: Selective Waiting

invocation of the select_wait() method and is used to coordinate communication between the five PHILOSOPHER process objects using the socket handle SS declared in super class DP_Process, as previously illustrated in Figure 4.4. The PHILOSOPHER objects are dynamically created and are accessed in the pointer to PHILOSOPHER variables: Phil0, Phil1, ..., Phil4. Each PHILOSOPHER process object is subsequently activated with the call to lifecycle(...).

The SERVER S1 cycles infinitely evaluating the If statements expressed in the select_wait() method. The communication method in the first conditional statement that evaluates to true is executed and the server process is directed to the body of the conditional statement. Thus, conditional statements with communication methods operate just like conditional statements without communication methods, except for the communication occurring. This supports the principle of orthogonality. This example illustrates the use of Socket::Probe() to check the local conditions in regards to the contents of a message prior to an actual Socket::Receive(). The ability to peek into the contents of a message without actually receiving it is an expressive feature of IPCC++.

Within the definition of the select_wait() method, it is possible to declare and use SERVER children to perform the task being requested of the SERVER S1. Therefore, IPCC++ supports a concurrent client server paradigm, and the inheritance structure of C++ is used to support the concurrent server.

The socket object of IPCC++ supports interprocess communication for distributed memory models. The IPCC_SOCKET class definition of IPCC++ introduces interprocess communication as complete objects. It supports the popular client/server paradigm with selective waiting and the power of concurrent servers. The IPCC++ language model maintains the reliable software development environment common to object-oriented programming by supporting type checking and communication error detection within its message passing protocol. The use of the inheritance structure of C++ supports indirect naming by socket handle and demonstrates the expressiveness of IPCC++. The components of the IPCC++ language model are class definitions of objects which is orthogonal to C++ and maintains a high level of abstraction while realizing the power of concurrent programming. The implementation of the IPCC++ language model integrates the class definitions of the IPCC++ language model with the PVM software system to provide an efficient yet reliable concurrent object-oriented programming environment.

4.3 Dining Philosophers Problem & IPCC++ Solution

The dining philosopher problem represents a classic concurrent programming problem. The IPCC++ solution is designed as a client/server application where a SERVER process is used to coordinate the actions of the dining philosophers. A PHILOSOPHER is a client process that spends its life thinking about life, requesting to take a fork resource, eating spaghetti, and requesting to release a fork resource. The

SERVER process is a server that spends its life servicing the requests of take and release a fork resource received from PHILOSOPHERS. The solution described below uses the socket object to synchronize the PHILOSOPHERS. This solution, which assumes no shared-memory, is designed for a distributed memory model. The suspension rules are defined as follows:

- A PHILOSOPHER eats only if he has two forks.
- No two PHILOSOPHERS may hold the same fork.
- No deadlock.
- No individual starvation!
- Efficient behavior under the absence of process contention [BenA90].

Figure 4.7 is a picturesque image of the table layout that depicts the position of the forks and the philosophers as they dine on spaghetti.

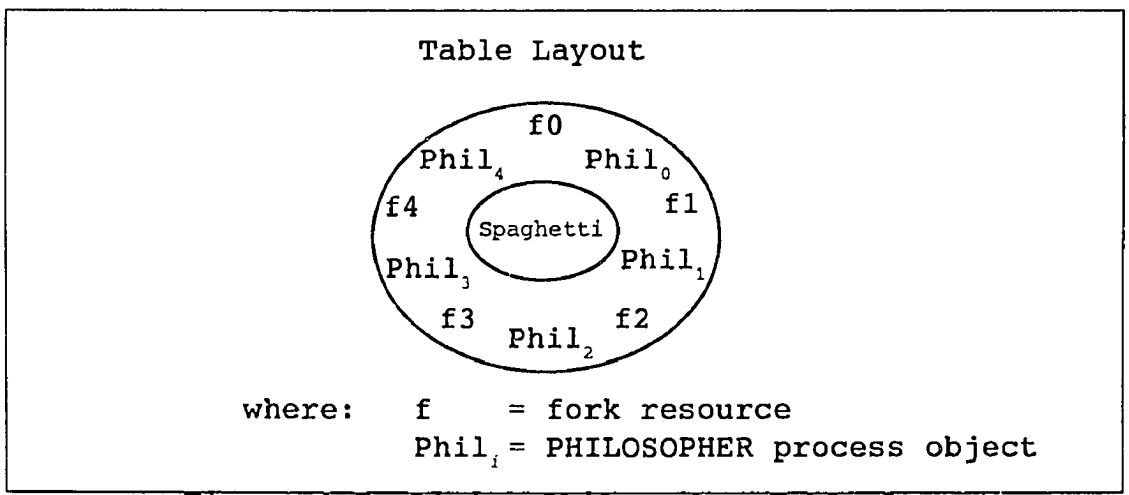


Figure 4.7: Dining Philosopher Table Layout

The IPCC++ code in Section 4.4 solves the Dining PHILOSOPHER problem defined above. The class DP_Socket is derived public from IPCC_SOCKET. It configures the socket for synchronous communication and to support messages of integer variables, with a maximum message length of 2. It uses the

matching key communication pair for process communication. It defines methods of Send(), Receive() and Probe() which accept a parameter of the message type. These methods in turn invoke Socket::Send(...), Socket::Receive(...), and Socket::Probe() methods, respectively, to carry out the communication.

The class DP_Process is derived public from IPCC_PROCESS. It declares the a socket object SS of the type DP_Socket. The class SERVER and PHILOSOPHER are derived public from DP_Process. Through C++ inheritance feature, process objects of the type SERVER or PHILOSOPHER have access to a socket object SS.

The SERVER class definition defines a constructor to initialize its private data, a destructor to delete its private data, and a method of select_wait to service clients. The select_wait method is an infinite loop that accepts messages from the socket object SS. The receive method calls are placed in conditional statements and are evaluated only if the conditional has not yet failed. This demonstrates IPCC++ ability to perform selective waiting.

The PHILOSOPHER class definition defines a method of lifecycle. The lifecycle method is an infinite loop that represents the life of a PHILOSOPHER. That is, the PHILOSOPHER thinks about life, sends a message to the SERVER using the matching key code, "T", to indicate a requests to Take a fork, eats spaghetti, and sends a message to the SERVER using the matching key code "R", to indicate a requests to Release a fork.

The main() procedure statically declares the SERVER process object S1 and dynamically declares the PHILOSOPHER objects Phil0, Phil1, ..., Phil4. The SERVER process object is activated by accessing the method select_wait and the PHILOSOPHER process objects, Phil0, Phil1, ..., Phil4, are activated by each accessing their method lifecycle.

4.4 IPCC++ Code Solution to Dining Philosopher Problem

This Section is the complete IPCC++ solution to the dining philosopher problem with a conceptual illustration of the object relationships depicted in Figure 4.8.

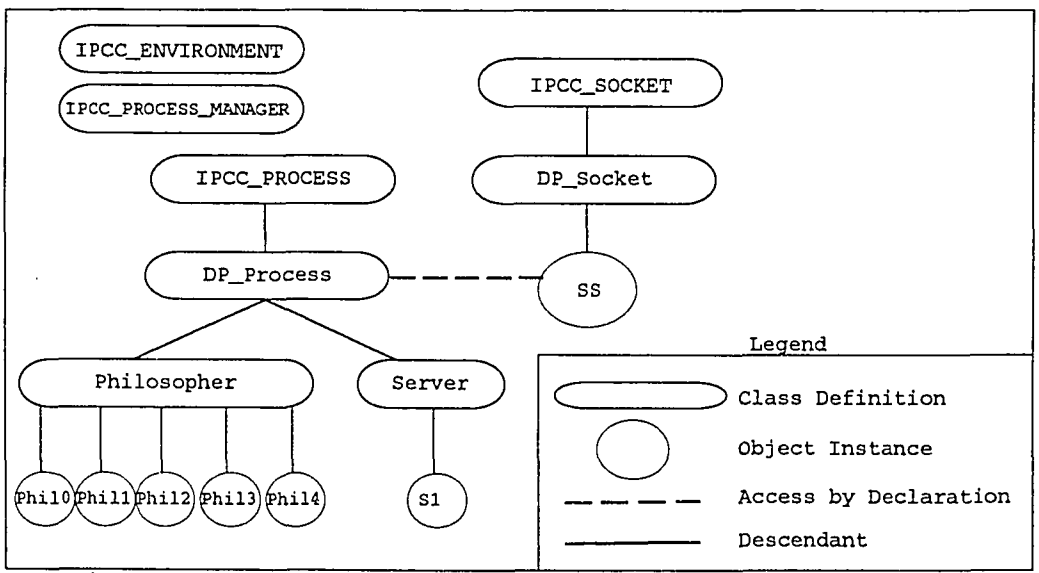


Figure 4.8: Dining Philosopher Object Relationships

```
#define MAXLEN 2
// Class DP_Socket Definition
class DP_Socket : public IPCC_SOCKET
  {  int maxmsgsize;
     int *msg;
```

```
    int msglen;
    int async=0;
    char key;
    public:
     DP_Socket()
       {  maxmsgsize=MAXLEN;}
     DP_Socket(int num)
       {  maxmsgsize=num;}
    int Send(int *msg, int msglen, char key);
    int Receive(int *msg, int msglen, char key);
    int Probe(int *msg, int msglen, char key);
 };
int DP_Socket::Send(int *msg, int msglen, char key)
 { return (IPCC_SOCKET::Send(*msg, msglen, key));}
int DP_Socket::Receive(int *msg, int msglen, char key)
 { return (IPCC_SOCKET::Receive(*msg, msglen, key));}
int DP_Socket::Probe(int *msg, int msglen, char key)
 { return (IPCC_SOCKET::Probe(*msg, msglen, key));}
//Class DP_Process definition
class DP_Process : public IPCC_PROCESS
 {   public:
      DP_Socket SS(2); }
 };
//Class SERVER definition
class SERVER : public DP_Process
 {   int *f;
     SERVER(int fnum);
```

```
      ~SERVER()
       {   delete f; }
      void select_wait();
 };
void SERVER::SERVER(int fnum)
 {   int i;
      f = new int[fnum];
      while (i<fnum)
       { f[i]=2; }
 }

void SERVER::select_wait()
 {  int i;
    while (1)
      { if (f[I]>=2 && (i=SS.probe(int *I, int len, 'T')!=0))
            { SS.receive(int *I, int len, "T")
              f[(I+1)%5] = f[(I+1)%5] - 1;
              f[(I-1)%5] = f[(I-1)%5] - 1;
              break; }
        if (i=SS.receive(int *I, int len, 'R')!=0)
            { f[(I+1)%5] = f[(I+1)%5] + 1;
              f[(I-1)%5] = f[(I-1)%5] + 1;
              break; }
       }
   }
//Class PHILOSOPHER Definition
class PHILOSOPHER : public DP_Process
```

```
 {   int *id;
     PHILOSOPHER::PHILOSOPHER(void)
     void lifecycle(int id);
 };
void PHILOSOPHER::lifecycle(int id)
 { while(1)
     { /* thinking about life*/
       SS.send(&id, sizeof(id), 'T');
       /* eating spaghetti*/
       SS.send(&id, sizeof(id), 'R');
     }
  }
Main()
 {   SERVER S1;
     PHILOSOPHER* Phil0 = new  PHILOSOPHER();
     PHILOSOPHER* Phil1 = new  PHILOSOPHER();
     PHILOSOPHER* Phil2 = new  PHILOSOPHER();
     PHILOSOPHER* Phil3 = new  PHILOSOPHER();
     PHILOSOPHER* Phil4 = new  PHILOSOPHER();
     S1.select_wait();
     Phil0->lifecycle(0);
     Phil1->lifecycle(1);
     Phil2->lifecycle(2);
     Phil3->lifecycle(3);
     Phil4->lifecycle(4);
  }
```

This solution displays the ease of using IPCC++ to solve concurrency problems designed for the distributed memory model. The following Section contains a summary of the features of IPCC++ which support the distributed memory model.

4.5 Summary of IPCC++ Distributed Memory Model

As described in the previous discussion, IPCC++ utilizes the socket object to support interprocess communication for a distributed memory model. The socket object is introduced with the IPCC_SOCKET class definition which supports the principle of orthogonality and utilizes C++ inheritance. This class definition offers methods to facilitate sending and receiving of messages. The send, receive, and probe methods return an integer variable which facilitates selective waiting and run-time error detection. Associating the send, receive, and probe methods with the socket object rather than the process object supports indirect naming. A socket object can be configured for either synchronous or asynchronous communication, and it supports type checking of messages.

In Chapter 5, a comparison to related research is performed. The comparison encompasses two different representations of C++ extensions. One representation supports centralized memory models and the other representation supports distributed memory models.

5. RELATED WORK

C++ is an extension of the C programming language that supports object-oriented programming paradigms by including: the concept of class, a mechanism for defining abstract data types (ADT's) and a means of providing inheritance and run-time type binding [Poh189]. IPCC++ is a natural extension of the C++ programming language in that it supports concurrent object-oriented programming for centralized and distributed computing environments by providing primitives of concurrency, each encapsulated in class definitions. The primitives include the process object, monitor object, condition object, and socket object.

The process object supports the concept of process and specification of concurrent execution. The monitor and condition objects support concurrency in a centralized computing environment. The monitor object guarantees mutual exclusion and uses the condition object to synchronize process objects. The socket object supports interprocess communication in a distributed computing environment. The synchronization of process objects is guaranteed by the sending and receiving of messages. The next section presents different programming languages that represent extensions of C++ which support concurrency features for centralized or distributed memory models.

5.1 Related Research

Research in the area of extending C++ with concurrency features has yielded numerous languages designed for different

* Portions reprinted by copyright permission of Proceedings of Nineteenth Annual International Computer Software and Applications Conference [Stub95b] and Journal of Object-Oriented Programming [Stub95a],[Stub95c].

computer environments. The C++ language extensions that support parallel execution are divided into two groups. One group, which supports centralized memory models, introduces concurrency features to facilitate interprocess communication based on shared memory. The other group, which supports distributed memory models, introduce concurrency features to facilitate interprocess communication in the absence of shared memory (a message passing system). Representative of the extensions for centralized memory models are: PRESTO [Bers88], μC++ [Buhr92], Parmacs C++ [Beck90], and Concurrent C++ [Geha88]. Table 1 provides an overview of these languages. Representative of C++ extensions for distributed memory models are: CHARM++ [Kale93], CC++ [Chan93], and MCC++ [Smit91]. Table 2 provides an overview of these languages along with IPCC++. The information within the tables represents our best interpretation of the languages based on the references cited.

Table 1 and Table 2 represent a comparison of the centralized and distributed memory model extensions of C++, respectively. The features of concurrency, the process management techniques, and the communication protocol are compared. The contents of the Table 1 and 2 focus on the features supported, the utilization of the C++ inheritance feature, and on the constructs and techniques to introduce and support centralized and distributed computing.

With respect to Tables 1 and 2, a language introduces concurrency as complete objects only if the class specifier mechanism of C++ is the only construct used to introduce

Table 4.1: Centralized Memory Model Extensions

LANGUAGES:	IPCC++	PRESTO	μC++	Parmacs C++	Concurrent C++
<u>Concurrency Features:</u>					
Complete Objects	Y	N	N	N	N
Explicit Concurrency	Y	Y	Y	Y	Y
C++'s Inheritance Capabability	Y	Y	Y	Y	Y
Inter-Object Concurrency	Y	Y	Y	Y	N
Intra-Object Concurrency	N	Y	N	N	N
<u>Process Management:</u>					
Complete Objects	Y	Y	Y	N	N
Static & Dynamic Process Declaration	Y	Y	N	N	N
Uncouples Declaration & Activation	Y	Y	N	N	N
Hierarchal Relationship	Y	Y	N	N	N
Synchronous Communication	Y	Y	Y	Y	Y
Asynchronous Communication	Y	Y	Y	Y	Y

concurrency. A language supports explicit concurrency only if it is the responsibility of the programmer to specify the concurrency, not the compiler. A language exploits the C++ inheritance capability only if the concurrency features are introduced as base class definitions and if subclass definitions can be derived from the base classes. The other characteristics in the tables are self explanatory.

5.2 Related Centralized Memory Models

The languages within the shared-memory representation support centralized computer systems and facilitate interprocess communication utilizing shared memory by using a concurrency primitive which guarantees mutual exclusion and supports process synchronization. The following discussion describes the specific techniques used by each language to introduce primitives of concurrency and how each compares with IPCC++.

PRESTO is a programming system implemented in C++ for writing object-oriented parallel programs in a multiprocessor environment. It provides a set of predefined object types that are used to simplify the construction of parallel programs. The PRESTO system consists of the language C++, a language library of basic tools constructed in C++, and a run-time system providing efficient support [Bers88]. PRESTO introduces concurrency features such as Monitor variables, Condition variables, and Threads as classes. PRESTO provides the type Monitor to guard access to blocks of code. The monitor is referred to as an object; however, a special syntax is used for

Table 4.2: Distributed Memory Model Extensions

LANGUAGES:	IPCC++	Charm++	CC++	MCC++
Concurrency:				
Complete Objects	Y	N	N	Y
Explicit Concurrency	Y	Y	Y	Y
C++'s Inheritance Capability	Y	Y	Y	Y
Inter-Object Concurrency	Y	Y	Y	Y
Intra-Object Concurrency	N	Y	Y	N
Process Management:				
Complete Objects	Y	Y	N	Y
Static & Dynamic Process Declaration	Y	Y	Y	N
Uncouples Declaration & Activation	Y	N	N	Y
Hierarchal Relationship	Y	Y	N	Y
Synchronous Communication	Y	Y	Y	Y
Asynchronous Communication	Y	Y	Y	Y
Communication Protocol				
Complete Object	Y	Y	N	Y
Selective Waiting	Y	Y	Y	N
Message Type Checking	Y	Y	N	N
Indirect Naming	Y	N	N	N
Communication Configuration	Y	N	N	N
Run-time Error Detection	Y	N	N	N

a monitor and a PRESTO monitor object doesn't support the inheritance feature of C++. In using the PRESTO monitor, a condition variable is required to be explicitly bound to the monitor. IPCC++ avoids this requirement by exploiting the C++ friend concept and by requiring that the condition object be declared within the monitor object. With the approach used in IPCC++, the condition object is encapsulated within the monitor that declared it and is invisible outside of the monitor.

A PRESTO thread is equivalent to a process in IPCC++ therefore, we will refer to a thread as a process. Although PRESTO introduces a process as a class, it does not take advantage of the C++ constructor mechanism to create the process. Instead, it uses the keyword NEW for process creation. PRESTO supports dynamic process creation that forms a hierarchal relationship, but it does not use the inheritance feature of C++. Also, it does not take advantage of the process class specifier being a type. An asset of PRESTO is that it does support separate process creation and instantiation. The invocation step must pass as parameters the object name, method to execute, and any other parameters required by the method. IPCC++ performs the same function somewhat more elegantly. IPCC++ simply declares an object of the process type then subsequently directs the process to execute one of the methods associated with that process object. Although process management in PRESTO is cumbersome, it has an advantage in that it supports intra-object concurrency. Therefore, PRESTO offers special class definitions of Relinquishing locks and

Non-relinquishing locks for intra-object process synchronization [Bers88].

The μC++ language uses a set of class definitions and special statements to introduce concurrency to C++. μC++ introduces concurrency features such as coroutines, monitors, and tasks as class type specifiers [Buhr92]. A task in μC++ is equivalent to a process in IPCC++. The monitor structure has been extended with the ability to postpone requests and is called a coroutine-monitor [Buhr92]. In μC++ type specifiers, uMutex and uNoMutex, are used to indicate a critical section [Hoar74] of code. IPCC++ has no need for the special type specifiers, for it simply exploits the guaranteed mutual exclusion associated with the monitor structure. In addition μC++ does not introduce concurrency to C++ as complete objects. Rather, a condition variable and new statements; uSuspend, uResume, uCoDie, uAccept, uWait, uDie are used to control task creation, termination, and interactions [Buhr92]. A task in μC++ is activated at the time of its declaration and is limited in its ability to pass parameters to constructors [Buhr92]. Again, IPCC++ uncouples process declaration and activation into separate constructs for added flexibility. μC++ does not offer the ability to dynamically create tasks. It does offer the ability to perform coroutine execution. IPCC++ does not contain a coroutine execution feature. The relationship between processes does form a hierarchy, but μC++ does not take explicit advantage of the hierarchical structure.

The Parmacs C++ language is essentially a set of C++ macros to facilitate the definition of concurrency and synchronization in a shared-memory environment [Beck90]. Parmacs C++ introduced primitives of concurrency as base class definitions. The primitives introduced are: BasicLock, Monitor, Barrier, Getsub, Askfor [Beck90]. These primitives form a hierarchal structure with BasicLock being at the lowest level and Getsub/Asksub at the highest level. Parmacs C++ omitted introducing a construct to simulate the condition variables associated with a monitor structure. Instead it provides wait() and signal() methods within the monitor class definition. This approach somewhat deviates from the original monitor definition, and this causes the structure to have the side affect of limiting processes that are suspended on different conditions to be placed in a single queue. Subsequently, the signal() operation may activate a process whose condition has not been satisfied and this can cause unnecessary state transitions of processes suspended in the queue. Special functions of pinit(), pmain(), pfinish(), and set_numprocs(int) are introduced to facilitate process management [Beck90]. Parmacs C++ uses these special functions rather than objects to introduce the process concept to C++. Also, these functions limit Parmacs C++ to only static creation of processes and couples process declaration and activation into one function. The relationship formed by PRESTO processes in Parmacs C++ is not hierarchical and therefore does not support the inheritance feature of C++.

The Concurrent C++ language was created by integrating C++ and Concurrent C to produce a language with both data abstraction and parallel programming facilities [Geha88]. The concurrency feature introduced is a process type specifier. The process type is not introduced as an object, but rather as a type. It does not require an extra construct for process synchronization, but it does incur overhead costs in that a transaction call is much slower than a member function call. This can lead to a bottleneck [Geha88]. IPCC++ introduces processes as class type specifiers which use the monitor class type specifier for synchronization. As processes are introduced as types, Concurrent C++ only supports static process creation. Also, the relationship among processes is not hierarchical and does not exploit the inheritance feature of C++. The developers of Concurrent C++ illustrate the use of classes to provide better user interfaces: greater functionality and more robustness [Geha88]. They also offer suggestions of future work in the direction of introducing the monitor structure and other primitives of concurrent programming. This suggestion in part motivated this research.

In summary, Table 1 represents a comparison of the centralized memory model extension of C++ with concurrency features. The features of concurrency and the process management techniques are compared. The contents of Table 1 focus on the features supported, the utilization of the C++ inheritance feature, and on the constructs and techniques used to introduce concurrency.

The IPCC++ language model supports concurrent object-oriented programming for centralized and distributed systems. IPCC++ introduces concurrency to C++ using the class construct primitive of C++. Introducing concurrency primitives as primitives of C++ (complete objects) supports the principle of orthogonality. The process concept, monitor structure, and condition variable are introduced as base class definitions of IPCC_PROCESS, IPCC_MONITOR, and IPCC_CONDITION, respectively. Each class definition supports inheritance, polymorphism, and dynamic run-time binding. IPCC++ utilizes the inheritance feature of C++ associated with class definitions to provide the flexibility to customize the base class definitions for specific applications. The monitor structure and condition object support interprocess communication for centralized memory models. Mutual exclusion is guaranteed by the monitor object which uses the condition object to perform process synchronization. By design, IPCC++ only supports inter-object concurrency. We feel encapsulating one single executing process within an object supports modularity. IPCC++ requires a condition object be declared within a monitor object and utilizes the friend concept of C++ to give the condition object access to data of the monitor object. This approach supports information hiding by encapsulating the condition object within the monitor object that declares. A monitor object declares different condition objects to express its different synchronization conditions. Each condition object has its own

queue of suspended processes which limits the number of process state transitions performed.

The IPCC_PROCESS class definition supports the creation, activation, and termination of multiple program controls (processes) within a single address space. In IPCC++, the declaration of a process object is static or dynamic and results in the creation of a suspended process. IPCC++ supports separate declaration and activation of process objects by utilizing the invocation of a method of the process object to activate the suspended process object. The IPCC++ process objects form a hierarchical relationship and support synchronous and asynchronous communication. The inheritance feature of C++ is used to make a monitor object visible to all process objects derived from the process class that defines the monitor object.

The comparison of the language features in Table 1 depicts the significance of the IPCC++ Language model. As shown, IPCC++ offers many benefits not supported by the other languages within the centralized memory model representation.

5.3 Related Distributed Memory Models

The languages within the distributed memory model representation support distributed (network) computer systems and facilitate interprocess communication utilizing an application program interface (API) based on a message passing protocol. The following discussion describes the concurrency features supported by each language and how they compare with IPCC++.

CHARM++ is a portable object-oriented parallel programming system. It is basically the C++ language minus global variables plus a few extensions to support parallel execution.[Kale93] The parallel features introduced are designed for execution within the Charm parallel programming system, not UNIX. IPCC++ supports the popular UNIX environment. The CHARM++ language supports inheritance and specific modes of information sharing by offering the following abstractions: Read-Only Objects, Write-Once Objects, Accumulator Objects, Monotonic Objects, and Distributed Tables. CHARM++ supports explicit concurrency by distinguishing concurrent objects from sequential objects by changing the reference syntax of methods. CHARM++ supports dynamic load balancing and has a message driven scheduling strategy as its synchronization tool. It introduced a message construct which is very similar to the struct C++ construct and supports prioritizing of messages. Therefore, the features of concurrency are not introduced solely as objects as in IPCC++. IPCC++ supports some form of prioritizing of messages with the key code matching of send() and receive(). CHARM++ does not support cross machine pointers, but does offer a feature to pack and unpack a message structure.

A chare is a process in CHARM++. For the sake of the discussion, we will refer to a chare as a process. Processes in CHARM++ are introduced as objects and support the hierarchical structure but do not capitalize on all the features of C++ objects. CHARM++ supports a function of

new_chare() to create a process rather than using the class construct as in IPCC++. The arguments of new_chare() are used to direct the execution of the new process to some entry point. An entry point defines the initial execution of a process. This results in coupled process declaration and activation. The interprocess communication of CHARM++ supports selective waiting, synchronous and asynchronous communication, and direct naming. IPCC++ achieves the same communication with the flexibility of indirect naming.

Compositional C++ (CC++) is a declarative compositional C++. It extends C++ for writing declarative and concurrent programs. CC++ is a notation for reactive systems executing on heterogeneous distributed environments and on parallel super computers [Chan93]. The concurrency features are represented as constructs of CC++. They include Parallel blocks, Spawn statement, Atomic functions, Logical processors, Global pointers, and Sync variables. These primitives are tools used to create concurrent programs. IPCC++ introduced concurrency at a higher abstraction level in order to simplify the language and its use. In CC++, it is the programmers responsibility to construct functions of send() and receive(), where as in IPCC++, they are provided as methods of a Socket object.

CC++ uses the parallel block structure and the spawn statement to introduce concurrency to C++, not objects. A parallel block represented by the statement par parblock defines a parallel block. Parblock is the name associated with the parallel block. A parallel block follows the syntax of a

block in C++; however, restrictions exist. Some restrictions are that no local variables are allowed, component statements do not support labeling, and the return statement is not supported. These limitations deviate from the principle of orthogonality by using the same syntactic structure to express different semantics. IPCC++ introduces parallelism via activation of methods of process type objects and supports the principle of orthogonality. The parallel blocks of CC++ supports intra-object concurrency. By design, IPCC++ does not support inter-object concurrency. The spawn statement is used to create a new process and direct its action to the function name in the statement. This results in the creation of a process and its activation, that is, coupled process declaration and activation. IPCC++ uncoupled process activation and declaration for more flexibility. Sync variables are the synchronization tools of CC++. IPCC++ chose to introduce method calls of send() and receive() which are high level abstractions of low-level UNIX system calls rather than introducing a new variable type to achieve synchronization. CC++ supports distributed memory access by introducing the Logical processor.

An environment for distributed application execution developed by Microelectronics and Computer Technology Corporation (MCC) results in the development of a C++ based environment for the distributed execution of object-oriented applications[Smit91]. For the sake of clarity, we refer to this extension as MCC++. MCC++ provides implicit inter-object

communications to effect remote method invocations and utilizes the notion of futures to support both synchronous and asynchronous inter-object protocols[Smit91].

MCC++ utilizes the semantics of the C++ method call and extends it to encompass the notion of communication between objects distributed across nodes. It also introduces the notion of futures, common to the LISP programming language, to provide for synchronization necessary to allow distributed objects to work concurrently. MCC++ uses operator overloading and the inheritance feature of C++ to facilitate remote object invocation and synchronization. While IPCC++ achieves the same effect through preprocessing the application and inheritance, it goes one step further and supports explicit interprocess communication with the socket object rather than just implicit interprocess communication via a method invocation.

MCC++ overloads the "method call via an object pointer" in a class called RemoteBase. The overloaded operator builds a message containing the address of the target object, method identification, and copies of the arguments and then transmits the message. All distributable objects are derived from RemoteBase and therefore execute the overloaded "method call via object pointer". RemoteBase is derived from the class Handle which is used to act as a global pointer to an object that can be dereferenced by any node in the distributed system. MCC++ attempts to overload the new operator to support remote object invocation; but, due to the implicit constructor invocation, the developers of MCC++ decided to modify the GNU

C++ compiler to avoid the execution of the constructor. This approach was not an option when developing IPCC++. Instead, we chose to preprocess the application source and achieve the same affect. The IPCC++ preprocessing proved to be a sound and portable means of realizing distributed computing within C++.

MCC++ introduces the notion of futures to provide for process synchronization. A future allows the execution of the invoking object to proceed without waiting for the completion of the remote methods execution. Futures are introduced as a class definition and when a future object appears on the "right hand side" of an expression, synchronization occurs. IPCC++ performs the synchronization implicitly without the need for futures. The preprocessing of IPCC++ determines if synchronization is required. If it is, then suspension will occur, otherwise both processes will proceed in parallel. Although MCC++ clearly succeeds in utilizing the features of C++, IPCC++ achieves a higher level of abstraction by eliminating the need for futures. MCC++ does not support true interprocess communication among objects. It supports interprocess communication via method invocation which we refer to as implicit interprocess communication. IPCC++ provides explicit interprocess communication as well as implicit interprocess communication.

In summary, Table 2 represents a comparison of the distributed memory model extension of C++ with concurrency features. The features of concurrency, the process management techniques, and the communication protocol are compared. The

contents of Table 2 focus on the features supported, the utilization of the inheritance feature of C++, and on the constructs and techniques to introduce and support distributed computing.

IPCC++ uses a socket application program interface to support interprocess communication for distributed systems. IPCC_Socket is the base class definition which represent a socket structure. The IPCC_Socket class supports a typed message passing interprocess communication protocol. IPCC_Socket methods of send() and receive() return an integer type variable which supports the client/server paradigm as well as run-time communication error detection. IPCC++ utilizes the C++ inheritance features to provide communication configuration and indirect naming. A communication can be configured for a particular type of message and designed to support either synchronous or asynchronous communication. Indirect naming of a socket handle rather than a specific process identification give IPCC++ expressiveness.

The above discussion classifies the IPCC++ language model extension of C++ as a unique concurrent object-oriented language. When a base language is extended, the extension must support the underlying methodologies of that language. IPCC++ is the only language extension that supports the object-oriented programming paradigm in every facet of its extension used to introduce the concurrency paradigm. IPCC++ also provides features such as indirect naming which are not supported by the other languages in the representation.

Of the centralized memory model representation, Presto, μC++, Parmacs C++ and Concurrent C++ fail to extend C++ solely with objects. Presto, and Concurrent C++ do not support inheritance within their synchronization constructs. μC++ and Parmacs C++ do not support dynamic process creation and uncoupled process declaration and activation. Also, IPCC++ is the only member of this representation to support both interprocess communication for the distributed memory model as well as centralized memory models.

Of the distributed memory model representation, Charm++ and CC++ fail to extend C++ solely with objects. MCC++ support the underlying methodologies of C++, but it does not support support all of the features offered by IPCC++. It does not provide for explicit interprocess communication with send() and receive() primitives.

IPCC++ interjects the power and efficiency of concurrent programming into the object-oriented programming language C++ while maintaining the integrity of C++. IPCC++ supports inter-object concurrency, static and dynamic process creation, uncoupled process declaration and activation, synchronous and asynchronous communication, message type checking, indirect naming, communication configuration, run-time communication error detection, and selective waiting. IPCC++ supports modular software development for centralized and distributed memory models while maintaining a high level of abstraction and achieving concurrency.

In Chapter 6, we present the physical implementation of the theoretical IPCC++ language model as it pertains to its components and environment of execution.

6. IPCC++ IMPLEMENTATION MODEL

The IPCC++ implementation model defines the structure of the class definitions and the details necessary to implement concurrent object-oriented programming for centralized and distributed computing systems. The implementation design of IPCC++ is based on the PVM software system which supports an abstraction of UNIX system calls. The conceptual image of the IPCC++ environment is depicted in Figure 6.1. This illustrates the traditional layered operating system abstractions where

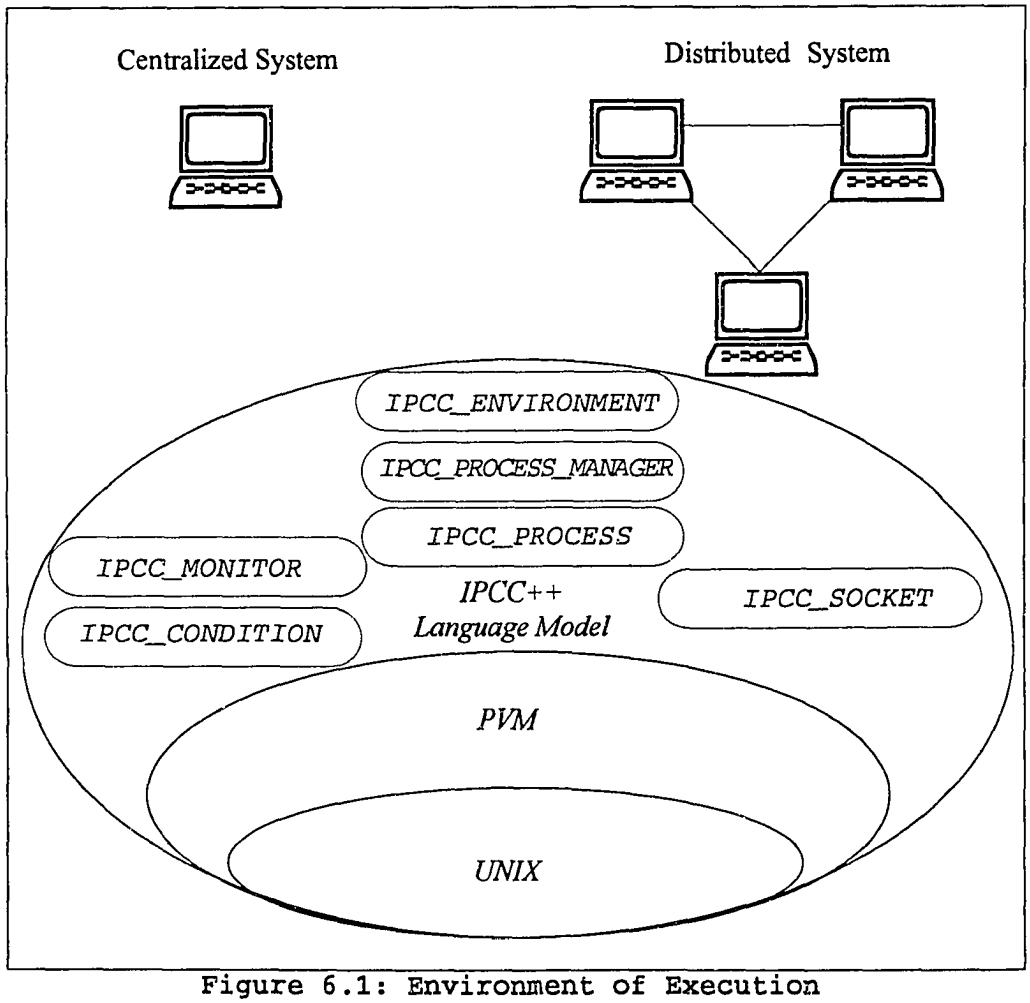


Figure 6.1: Environment of Execution

each of the layers: IPCC++ Language Model, PVM, and UNIX employ the functions provided by the layers below it. This design, in part, achieves the high level of abstraction associated with the IPCC++ language model. The components of the model illustrated in Figure 6.1 support the concurrency features of IPCC++. The class definitions IPCC_ENVIRONMENT, IPCC_PROCESS_MANAGER, IPCC_MONITOR, IPCC_CONDITION, and IPCC_SOCKET all have a peer relationship.

6.1 Implementation and the PVM Software System

The IPCC++ components makes use of the PVM abstraction of the UNIX operating system to implement the features of concurrency. The PVM software system enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource [Begu94]. The PVM environment gives IPCC++ the power to support heterogenous distributed computations.

The features of PVM used in the IPCC++ design include: Process Control, Dynamic Process Groups, and Communication. Process Control refers to the ability to enroll a process in and out of the PVM environment, create and terminate PVM tasks (processes), and send signals [Begu94]. IPCC++ enrolls the initial program control into the PVM environment and utilizes the PVM create and terminate routines to create and terminate a thread of control in a process object, respectively. Dynamic Process Groups refers to the ability to dynamically form groups of processes [Begu94]. PVM routines exist to add or delete a process from a group. Processes can be members of multiple

groups. IPCC++ uses the dynamic process group feature to support indirect naming and interprocess communication with a IPCC_SOCKET object. Communication refers to interprocess communication and the features used to support heterogeneous systems. PVM offers routines to send, receive, probe, pack, and unpack messages. With each process, there exists message buffers, and PVM routines necessary to create, clear, and delete the message buffers. These routines are used to implement synchronization and interprocess communication within IPCC++. PVM is a useful abstraction of the UNIX operating system. IPCC++ can be viewed as a subset abstraction of the PVM system. It replaces the details of the PVM system with objects and object components. The IPCC++ system preprocesses IPCC++ into C++ code and hides the implementation details from the user. The following section defines the preprocessing of IPCC++ to C++.

6.2 Preprocessing of IPCC++ to C++

The preprocessing of IPCC++ to C++ is performed in two phases. In phase one, the source code is parsed to create two files, the Declaration/Activation file and the Communication file. The record layout of each file is depicted in Figure 6.2. This phase can be compared with a lexical scan phase of a compiler which builds the symbol table of a program, except it builds the Declaration/Activation file and Communication file. The Declaration/Activation file contains the declaring process class name (ClassId), the instance variable declared (InstanceId), a system generated number which is used to

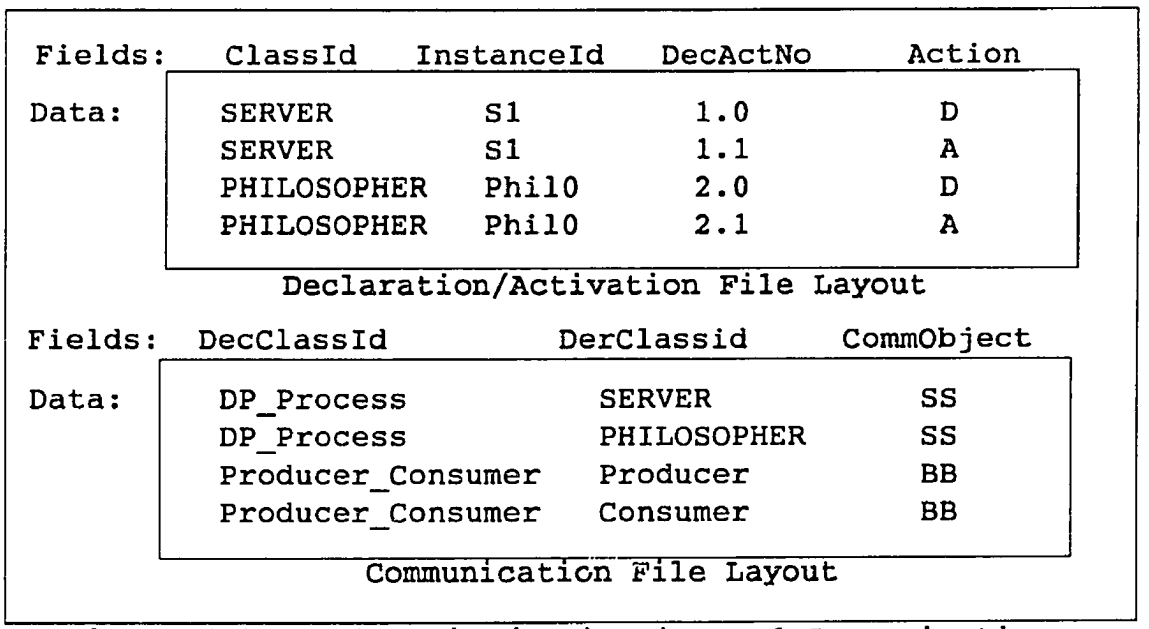


Figure 6.2: Declaration/Activation and Communication File Layouts

identify the object (ActDecNo), and a code (Action) indicating whether the action is a declaration of a process or an activation of a process. Its data represents every process declaration and activation which occurred in the program. A Declaration/Activation number is assigned to each occurrence and is used within the IPCC_PROCESS_MANAGER Class and the IPCC_PROCESS Class to facilitate declaration and subsequent activation of process objects (as described in Sections 2.3 and 2.4). The Communication file contains the declaring process class name (DecClassid), the derived class identification (DerClassid) which has access to the instance variable and the instance identification of the communication object (CommObject). The data of the file represents the hierarchy of every process class declaration which has access to a particular communication object. This information is used to associate process

groups with an instance of a socket object, that is to form communication groups of processes for a socket object.

During phase two, the two files are used and the source is parsed again to identify all process object declarations and activations. For every process declaration or activation, the parsing process generates code accessing IPCC++ class definitions. It also generates code which modifies the class constructor of each process class and socket class by passing parameters to the base class constructors, IPCC_PROCESS and IPCC_SOCKET, respectively. Figures 6.3, 6.4, and 6.5 illustrate code changes generated by the preprocessing phase. The C++ generated code illustrates the affects on the class constructor invocation and definition. Line numbers are assigned to the statements that result in code translation or generation. All statements which are not changed during the preprocessing have no line number assigned. The line number, also given in the C++ generated code, identifies the statement which generated its creation or modification.

Figure 6.3, which contains the IPCC++ code for the main() procedure and its C++ generated code, depicts the affect on process declarations and activations. As illustrated, the generated code from line 1, main() procedure call, results in the generation of lines 1a, 1b,..., 1e. Line 1a declares the IPCC_Environ object and line 1b declares IPCC_ProcessMgr object which establish the IPCC++ run time environment and provide declaration and activation information to child processes, respectively. The lines 1c, 1d, and 1e are used to

```
LINE#   IPCC++ STATEMENT
1:      int main() {
        int x=3, z, length;
        float y=2;
2:      SERVER s1(x, y);
3:      PHILOSOPHER* phil0 = new PHILOSOPHER();
4:      s1.select_wait();
5:      phil0->lifecycle(0);
        return 1; }

LINE#   C++ GENERATED CODE
1a:     IPCC_ENVIRONMENT IPCC_Environ;
 b      IPCC_PROCESS_MANAGER IPCC_ProcessMgr;
 c      char IPCC_ProgName[20];
 d      int main(int argc, char**argv)
 e       { strcpy(IPCC_ProgName,argv[0]);
2a:     SERVER s1("s1", x, y);
 b      int ChildTID = s1.GetChildTaskId();
 c      SENDER <int> __Sender2(ChildTID,ARGUMENT_INT,&x,1,1);
 d      SENDER <float> __Sender3(ChildTID,ARGUMENT_FLOAT,&y,1,1);
3:      PHILOSOPHER* phil0 = new PHILOSOPHER("phil0");
4:      s1.Activate();
5:      phil0->Activate();
```

Figure 6.3: Main Procedure Translations

```
LINE#   IPCC++ STATEMENT
        class PHILOSOPHER : public DP_PROCESS
        { public:
1:      PHILOSOPHER() { ;}
        float lifecycle(int Key)
        { int *z, length;
        ss.Receive(z, &length, -1);
        return (float)*z; } };

LINE#   C++ GENERATED CODE
        class PHILOSOPHER : public DP_PROCESS
        { public:
1:      PHILOSOPHER(char *InstanceId) : DP_PROCESS("PHILOSOPHER",InstanceId) {;}
        float lifecycle()
        { int *z, length;
        ss.Receive(z, &length, -1);
        return (float)*z; } };
```

Figure 6.4: Process Constructor Translations

```
LINE#   IPCC++ STATEMENT
      class DP_SOCKET : public IPCC_SOCKET
      { public:
1:      DP_SOCKET (int mode, int type, int direction) { ;}
        ... } };
      class DP_PROCESS : public IPCC_PROCESS
      { public:
2:      DP_SOCKET ss(int socketmode, int sockettype,
                     int socketdirection) { ;}
        ... } };

LINE#   C++ GENERATED CODE
      class DP_SOCKET : public IPCC_SOCKET
      { public:
1:      DP_SOCKET (char* InstanceId, int mode, int type, int direction)
            : ("DP_PROCESS", "ss") { ;}
        ... } };
      class DP_PROCESS : public IPCC_PROCESS
      { public:
2:      DP_SOCKET ss("ss",int socketmode,int sockettype,
                         int socketdirection) { ;}
            ... } };
```

Figure 6.5: Socket Constructor & Declaration Translations

configure information for internal use within the IPCC++ class definitions.

Line 2 represents a static declaration of a process object. Line 2a represents the translated code of line 2 which passes the instance identification of the statistically declared process object as an additional parameter to the SERVER class constructor. The IPCC_PROCESS class uses the information to retrieve the DecActNo from the IPCC_ProcessMgr object. Lines 2b stores the PVM task identification (tid) into an internal field of the IPCC_Process. Lines 2c and 2d pass the arguments of the constructor to the newly created process object.

Line 3 represents a dynamic declaration of a process object. Line 3a represents the translated code of line 3 which

passes the instance identification of the dynamically declared process object as an additional parameter to the PHILOSOPHER class constructor. Since no programmer supplied arguments are passed to the constructor, no additional lines are generated.

Lines 4 and 5 represent activation statements for the process objects SERVER s1 and PHILOSOPHER philo, respectively. The IPCC++ code is changed to access the *Activate()* method of the IPCC_PROCESS object.

Figure 6.4 contains the IPCC++ process class code and its C++ generated code, illustrating the affect to the class constructor. It illustrates the code modifications to process object constructors. Line 1 represents the constructor of PHILOSOPHER. During the translation phase, line 1 is modified such that it passes the class identification ("PHILOSOPHER") and the Instance identification ("philo") to the parent class constructor.

Figure 6.5 contains an IPCC++ process class definition which declares a socket and the corresponding class definition of the socket. It illustrates the code modifications to socket object constructors and declarations. Line 1 represents the definition of the constructor for DP_Socket. During the translation phase, line 1 is modified such that it passes the process class identification which declares the object ("DP_Process") and the Instance identification ("ss") to the parent class constructor, IPCC_SOCKET. Line 2, which represents the declaration of a socket object, is modified to include the instance id ("ss") of the socket instance variable.

```
// This method is called by all spawned PVM process. It waits for declare
// message and depending on the message value, creates a process object.
// This method is generated by the language translator.

void IPCC_ENVIRONMENT::Server() {
  int ReturnVal, MessageTag;
  float DecActNo;
  while(1) {
    // Check for a declare message
      ReturnVal = pvm_nrecv(ParentTID, DECLARE_TAG);
       <error code>

     // Get the declare number
      ReturnVal = pvm_upkfloat(&DecActNo, 1, 1);
        <error code>

     // Depending on the declare number, get the required parameters for
     // the constructor and create the process object. Then subsequently
     //  execute methods of the object.  Return a result if required.
      if (DecActNo == 1.0) {
        int  __Argument1;
        float __Argument2;
        int Result;

    RECEIVER <int> __Receiver1(ParentTID, ARGUMENT_INT, &__Argument1, 1, 1);
    RECEIVER <float> __Receiver2(ParentTID, ARGUMENT_FLOAT, &__Argument2,
                                    1,1);

     SERVER s1("s1", __Argument1, __Argument2);
        ReturnVal = pvm_recv(ParentTID, ACTIVATE_TAG);
          <error code>
       s1.selective_wait('A', 655);
        ReturnVal = pvm_recv(ParentTID, ACTIVATE_TAG);
          <error code>
        s1.selective_wait('B', 755);
        fflush(stdout);
        pvm_exit();
        exit(-1);
      }
      else if (DecActNo == 2.0) {
        float Result;
       PHILOSOPHER* phil0 = new PHILOSOPHER("phil0");
        ReturnVal = pvm_recv(ParentTID, ACTIVATE_TAG);
         <error code>
        Result = phil0->lifecycle('A');
         SENDER <float> s1(ParentTID, ACTIVATE_TAG, &Result, 1, 1);
        fflush(stdout);
        pvm_exit();
        exit(-1);
        }
       }
      }
     } [Dedh95]
```

<error code>

```
if (ReturnVal < 0) {
       pvm_perror(NULL);
       pvm_exit();
       exit(-1);
     }
```

Figure 6.6: IPCC_ENVIRONMENT::Server() Generated C++ Code

The preprocessing of *IPCC++* to C++ also results in the creation of the IPCC_ENVIRONMENT::Server() method. Figure 6.6 is the system generated method which represents the code of the main() procedure of Figure 6.3. This method is called by all spawned PVM processes, one PVM process for each declared process object. The method body is an infinite loop which waits for a declaration message from the initial program control. Upon receiving a declaration message, it creates a process object. The selection of which process object to create depends upon the *DecActNo* information within the message. Using the value of *DecActNo*, the spawned child enters the appropriate code block, receives parameters for the constructor (if applicable) by utilizing RECEIVER template(s), and results in the creation of a suspended process object with a process type corresponding directly with the process object in main(). The process is suspended by invoking a PVM receive command which results in waiting for an activation message from the initial program control. This two step procedure supports the separate declaration and activation feature of IPCC++. The code of Figure 6.6 uses <error code> to represent the error code block provided in the box, which detects PVM communication errors.

In summary, the Figures 6.3 through 6.6, illustrate how to create and activate process objects in the IPCC++ environment. The preprocessing translates IPCC++ code into C++ code by mapping process declarations, activations and communications into the underlying IPCC++ system. The decision to preprocess the language rather then modify any specific

compiler gives IPCC++ portability and flexibility because it can be used with any C++ compiler and environment which supports PVM. In Section 6.3 the implementation of the IPCC_ENVIRONMENT is discussed.

6.3 Implementation Details of IPCC_ENVIRONMENT

The functionality of the IPCC_ENVIRONMENT results in the constructor enrolling the process into the PVM environment and determining if it is a child process or the initial program control. If it is a child process, then its action is directed

```
IPCC_ENVIRONMENT::IPCC_ENVIRONMENT()
 {  int TID;

  // Enroll process into PVM
  TID = pvm_mytid();
  if (TID < 0) {
    pvm_perror(ProgName);
    exit(-1); }

  // Find out parent of this process
  ParentTID = pvm_parent();
  if ((ParentTID < 0) && (ParentTID != PvmNoParent)) {
    pvm_perror(ProgName);
    pvm_exit();
    exit(-1);  }

  // If this is a process spawned by another PVM process, execute
  // the server method.
  if (ParentTID != PvmNoParent)  IPCC_ENVIRONMENT::Server();
 };

IPCC_ENVIRONMENT::~IPCC_ENVIRONMENT()
 { fflush(stdout);
  pvm_exit(); }[Dedh95]
```

Figure 6.7: Implementation Detaills of IPCC_ENVIRONMENT Object

to the IPCC_ENVIRONMENT::Server() method (discussed in Section 6.2) where it waits for a message from the initial program control to direct its action. If it is the initial program

control, then it simply returns. The destructor is responsible for removing the processes from the PVM environment. The code representing these functions is given in Figure 6.7. For a complete discussion on the IPCC_ENVIRONMENT Server() method, please refer to Figure 6.6 of Section 6.2. In Section 6.4, the implementation details of the IPCC_PROCESS_MANAGER is discussed.

6.4 Implementation Details of IPCC_PROCESS_MANAGER

The constructor of IPCC_PROCESS_MANAGER is responsible for accessing the Declaration/Activation file created during the first phase of the preprocessing and loading it into dynamically allocated table structures of this class. Two tables are created, the DeclareTable and the ActivateTable. The methods of the class access the tables using the *char *ClassId, char *InstanceId* and return the *float ActDecNo* which is used to direct the suspended child process in the IPCC_ENVIRONMENT::Server() method to the appropriate code block. The destructor is responsible for deleting the dynamically allocated memory. The code which supports the functionality of IPCC_PROCESS_MANAGER is given if Figure 6.8. In Section 6.5, the implementation details of the IPCC_PROCESS class are discussed. The IPCC_PROCESS utilizes the data and functionality of the IPCC_ENVIRONMENT and the IPCC_PROCESS_MANAGER to control the declaration and activation of processes.

```
IPCC_PROCESS_MANAGER::IPCC_PROCESS_MANAGER()
{ FILE *fp;
  char Buffer[LINE_LENGTH];
  char ClassId[CLASS_ID_LENGTH];
  char InstanceId[INSTANCE_ID_LENGTH];
  char TableFile[50];
  float Number; char Type; int ReturnVal;
  int DeclareTableIndex = 0, ActivateTableIndex = 0;
  DeclareTableSize = 0; ActivateTableSize = 0;
  getcwd(DirectoryName, PATH_MAX);
  fp = fopen(TableFile, "r");
  do { ReturnVal = fscanf(fp, "%s %s %f %c\n", ClassId, InstanceId, &Number,
&Type);
    if (Type == 'D') DeclareTableSize++;
    if (Type == 'A')  ActivateTableSize++;
  } while (!feof(fp));
  DeclareTable = (DECLARE_STRUCT *)malloc(DeclareTableSize *
sizeof(DECLARE_STRUCT));
  ActivateTable = (ACTIVATE_STRUCT *)malloc(ActivateTableSize *
sizeof(ACTIVATE_STRUCT));
   rewind(fp);
  do {
    ReturnVal = fscanf(fp, "%s %s %f %c\n", ClassId, InstanceId, &Number,
&Type);
   if (Type == 'D') {
      strcpy(DeclareTable[DeclareTableIndex].ClassId, ClassId);
      strcpy(DeclareTable[DeclareTableIndex].InstanceId, InstanceId);
      DeclareTable[DeclareTableIndex].DeclareNo = Number;
      DeclareTableIndex++;   }
   if (Type == 'A') {
      strcpy(ActivateTable[ActivateTableIndex].ClassId, ClassId);
      strcpy(ActivateTable[ActivateTableIndex].InstanceId, InstanceId);
      ActivateTable[ActivateTableIndex].ActivateNo = Number;
      ActivateTableIndex++;  }
  } while (!feof(fp));
fclose(fp); }
IPCC_PROCESS_MANAGER::~IPCC_PROCESS_MANAGER()
{ if (DeclareTable)
    free(DeclareTable);
  if (ActivateTable)
    free(ActivateTable); }
float IPCC_PROCESS_MANAGER::GetDeclareNo(char *ClassId, char *InstanceId)
{ int i = 0;
  while (i < DeclareTableSize) {
    if (!strcmp(DeclareTable[i].ClassId, ClassId))
      if (!strcmp(DeclareTable[i].InstanceId, InstanceId))
        return (DeclareTable[i].DeclareNo);
    i++;  }
  return -1.0; }
float IPCC_PROCESS_MANAGER::GetActivateNo(char *ClassId, char *InstanceId)
{ int i = 0;
  while (i < ActivateTableSize) {
    if (!strcmp(ActivateTable[i].ClassId, ClassId))
      if (!strcmp(ActivateTable[i].InstanceId, InstanceId))
        return (ActivateTable[i].ActivateNo);
    i++;  }
  return -1.0; } [Dedh95]
```

Figure 6.8: Implementation Details of IPCC_PROCESS_MANAGER Object

6.5 Implementation Details of IPCC_PROCESS

The constructor of the IPCC_PROCESS class receives char *ClassIdIn and char *InstanceIdIn as arguments and uses the information to retrieve the ActDecNo from the IPCC_PROCESS_MANAGER object. It determines if the calling process is the initial program control. If it is the initial program control, then it spawns a child using PVM system calls and sends it a declaration message. This class contains an Activation method for all of the standard C++ types. The types correspond with the value returned by the method of the process which is to be executed. The preprocessing inserts a dummy argument of the same type as the return type associated with the process method invocation to dynamically select the appropriate method. The activation methods are responsible for sending the activate method to the spawned child. If the spawned process is activated to a method which returns a value, then the initial program control will suspend and wait for the process to return a value. If no return value is associated with the method, that is (void), then the initial program control is not suspended and the two processes (initial program control and spawned child) proceed in parallel.

Figure 6.9 contains the implementation details of the IPCC_PROCESS object. Only a subset of the activation methods have been shown; however, the class supports all standard C++ types. Process objects utilize the IPCC_SOCKET structure to support message passing interprocess communication. The design of the IPCC++ system is such that each process object

```
IPCC_PROCESS::IPCC_PROCESS(char *ClassIdIn, char *InstanceIdIn)
{ float DecNo;
  int ReturnVal;
  strcpy(ClassId, ClassIdIn);
  strcpy(InstanceId, InstanceIdIn);
  // If this is the main process, find out the declare number of the
  // object to be created, spawn a process and send the declare number
  // to that process for creation of process object.
  if (IPCC_Environ.IsParent()) {
    char Program[50];
    DecNo = IPCC_ProcessMgr.GetDeclareNo(ClassIdIn, InstanceIdIn);
    ReturnVal = pvm_spawn(Program, (char **)0, PvmTaskDefault, (char *)0, 1,
&ChildTaskId);
    if (ReturnVal == 0) {
      printf("Spawn failed\n");
      pvm_perror(InstanceIdIn);
      pvm_exit();
      exit(-1);  }
    SENDER <float> s1(ChildTaskId, DECLARE_TAG, &DecNo, 1, 1);  } }
IPCC_PROCESS::~IPCC_PROCESS() { }
from activated
void IPCC_PROCESS::Activate(void)
{ float ActNo;
  int ReturnVal;
  ActNo = IPCC_ProcessMgr.GetActivateNo(ClassId, InstanceId);
  SENDER <float> s1(ChildTaskId, ACTIVATE_TAG, &ActNo, 1, 1); }
// This method activates method of an object and returns integer value of
activated method
int IPCC_PROCESS::Activate(int x)
{ float ActNo;
  int ReturnVal;
  ActNo = IPCC_ProcessMgr.GetActivateNo(ClassId, InstanceId);
  // Send Activate tag for execution of method of the object.
  SENDER <float> s1(ChildTaskId, ACTIVATE_TAG, &ActNo, 1, 1);
  // Wait for an interger return value from the activated method of the
object.
  RECEIVER <int> r1(ChildTaskId, -1, &ReturnVal, 1, 1);
  return ReturnVal; }
              ...
// This method activates method of an object and returns byte array of acti-
vated method
char *IPCC_PROCESS::Activate(char *x)
{ float ActNo;
  char *ReturnVal;
  ActNo = IPCC_ProcessMgr.GetActivateNo(ClassId, InstanceId);
  // Send Activate tag for execution of method of the object.
  SENDER <float> s1(ChildTaskId, ACTIVATE_TAG, &ActNo, 1, 1);
  // Wait for a char * return value from the activated method of the object.
  RECEIVER <char> r1(ChildTaskId, -1, ReturnVal, 1, 1);
  return ReturnVal; }
// private methods of class used internally
int IPCC_PROCESS::GetChildTaskId(void)
{ return ChildTaskId; }
char *IPCC_PROCESS::GetClassId(void)
{ return ClassId; }
char *IPCC_PROCESS::GetInstanceId(void)
{ return InstanceId; } [Dedh95]
```

Figure 6.9: Implementation Details of IPCC_PROCESS Object

has its own version of a socket object; therefore, it uses the methods of IPCC_SetSocketMode(), IPCC_SetSocketType() and IPCC_SetSocketDirection() (defined in Section 4.1) to configure the socket object for the role of this process object in interprocess communication associated with this socket object. Section 6.6 describes the implementation details of the IPCC_SOCKET class definition which uses the PVM system to support interprocess communication.

6.6 Implementation Details of IPCC_SOCKET

The IPCC_SOCKET object facilitates interprocess communication for a group of process objects. As depicted in Figure 6.10, the constructor of the IPCC_SOCKET class receives *char *ClassIdIn, char *InstanceIdIn, int SocketModeIn, int SocketTypeIn,* and *int SocketDirectionIn* as arguments and uses the parameters, *ClassIdIn and InstanceIdIn,* to register the group identification of the socket with the PVM environment and enroll the process into the communication group for this socket (*GroupID*), respectively. The parameters of *SocketModeIn, SocketTypeIn,* and *SocketDirectionIn* are used to configure the socket for the particular communication role of the process declaring the object. The destructor of the IPCC_SOCKET class is responsible for removing the process object from the communication group for this socket. The communication methods support the standard C++ types and each make use the corresponding private internal communication methods of IPCC_Send_Internal(), IPCC_Receive_Internal(), and IPCC_Probe_Internal() to support message passing.

```
class IPCC_SOCKET {
 char GroupID[GROUP_ID_LENGTH];
  int SocketMode;
  int SocketType;
  int SocketDirection;
  // Internal methods for message handling. We handle all messages as
  // series of bytes.
  int IPCC_Send_Internal(char *Message, int Length, int Key);
  char *IPCC_Receive_Internal(int *Length, int Key);
  char *IPCC_Probe_Internal(int *Length, int Key);
public:
   IPCC_SOCKET(char *ClassId, char *SocketId, int SocketModeIn = IPCC_ASYNCHRONOUS,
        int SocketTypeIn = IPCC_CLIENT, int SocketDirectionIn = IPCC_UNIDIRECTION);
       { int ReturnVal;
         // Set values of private fields
       SocketMode = SocketModeIn;
       SocketType = SocketTypeIn;
       SocketDirection = SocketDirectionIn;
       if (!IPCC_Environ.IsParent()) {
       // If this is a pvm process, register group id
        ReturnVal = pvm_joingroup(GroupID);  } }
   ~IPCC_SOCKET()
       { if (!IPCC_Environ.IsParent()) {
       // If this is a pvm process, leave group
      pvm_lvgroup(GroupID);  }}

  // Methods for sending message. The supported message types are int, short,
  // long, float, double and char array. All these methods call the internal
  // send method for sending message.
  int IPCC_Send(char *Message, int Length, int Key);
  int IPCC_Send(float *Message, int Length, int Key);
  int IPCC_Send(double *Message, int Length, int Key);
  int IPCC_Send(int *Message, int Length, int Key);
  int IPCC_Send(long *Message, int Length, int Key);
  int IPCC_Send(short *Message, int Length, int Key);
  int IPCC_Send(char *Message, int Key);

  // Methods for receiving message. The supported message types are int, short,
  // long, float, double and char array. All these methods call the internal
  // receive method for receiving message.
  int IPCC_Receive(char *Message, int *Length, int Key);
  int IPCC_Receive(float *Message, int *Length, int Key);
  int IPCC_Receive(double *Message, int *Length, int Key);
  int *IPCC_Receive(int *Length, int Key);
  int IPCC_Receive(long *Message, int *Length, int Key);
  int IPCC_Receive(short *Message, int *Length, int Key);
  int IPCC_Receive(char *Message, int Key);

  // Methods for probing message. The supported message types are int, short,
  // long, float, double and char array. All these methods call the internal
  // probe method for probing message. A probe message is not removed from
  // the queue, but its contents are returned for examination.
  int IPCC_Probe(char *Message, int *Length, int Key);
  int IPCC_Probe(float *Message, int *Length, int Key);
  int IPCC_Probe(double *Message, int *Length, int Key);
  int IPCC_Probe(int *Message, int *Length, int Key);
  int IPCC_Probe(long *Message, int *Length, int Key);
  int IPCC_Probe(short *Message, int *Length, int Key);
  int IPCC_Probe(char *Message, int Key);
  // Methods for changing values of private fields
  void IPCC_SetSocketMode(int Mode);
  void IPCC_SetSocketType(int Type);
  void IPCC_SetSocketDirection(int Direction);
};[Dedh95]
```

Figure 6.10: Implementation Details of IPCC_SOCKET Object

The methods of IPCC_Send_Internal(), IPCC_Receive_Internal(), and IPCC_Probe_Internal() are used to manipulate the sending and receiving of messages between processes. All messages are transmitted as an array of bytes. Each of these methods use internal message identification keys to identify the type of message sent, received, or probed. The internal keys are ASYNC_REQUEST, ASYNC_RESPONSE, SYNC_REQUEST, SYNC_RESPONSE, CLIENT_ACK, SERVER_ACK. They are defined as integer values in the range 0 - 5, respectively, and are reserved key values. When a process is a client for a particular socket, then its socket communication is restricted to sending requests to the socket object. When a process is a server for a particular socket, then its communication is restricted to receiving or probing requests from the socket object. When a process sets SocketType as inactive (not a client or a server), then its communication is not restricted so it can send, receive, or probe messages. The classification of processes into communication roles and the internal key codes support the indirect naming communication feature of IPCC++. As depicted in Figure 6.10, the methods of IPCC_Send(), IPCC_Receive(), and IPCC_Probe() support standard C++ types.

Figure 6.11 illustrates the details of the IPCC_Send_Internal() communication method. The general functionality of the IPCC_Send_Internal() method performs the following steps.

- It broadcast a message to the group which contains the task identification (tid) of the invoking process.

```
int IPCC_SOCKET::IPCC_Send_Internal(char *Message, int Length, int Key)
{ int ReturnVal;
  int GSize, i;
  if (((SocketType != IPCC_INACTIVE) &&
               (SocketDirection == IPCC_BIDIRECTION)) ||
       (SocketType == IPCC_SERVER)) {
// Initialize message sending only if the socket is not inactive and is bidirectional
// or the socket type is server.  Broadcast the key so that the receiver can know the
// sender task id for receiving message.
    if (Key != -1) {
      ReturnVal = pvm_initsend(PvmDataDefault);
      if (ReturnVal < 0) {
        pvm_perror(NULL);
        return(-1);      }
      ReturnVal = pvm_bcast(GroupID, Key);
      if (ReturnVal < 0) {
        pvm_perror(NULL);
        return(-1);      }    }
    ReturnVal = pvm_initsend(PvmDataDefault);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(-1);}
    ReturnVal = pvm_pkint(&Key, 1, 1);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(-1); }
    ReturnVal = pvm_pkint(&Length, 1, 1);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(-1); }
     ReturnVal = pvm_pkbyte(Message, Length, 1);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(-1); }
// Broadcast the message to all other processes in the group.
    if (SocketType == IPCC_CLIENT) //If the socket is a client, then it makes a request.
      if (SocketMode == IPCC_SYNCHRONOUS) {
        ReturnVal = pvm_bcast(GroupID, SYNC_REQUEST);      }
      else {
        ReturnVal = pvm_bcast(GroupID, ASYNC_REQUEST);      }
    else // If the socket is server, then it responds to a request
      if (SocketMode == IPCC_SYNCHRONOUS) {
        ReturnVal = pvm_bcast(GroupID, SYNC_RESPONSE);      }
      else {
        ReturnVal = pvm_bcast(GroupID, ASYNC_RESPONSE);      }
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(-1);    }
   if (SocketMode == IPCC_SYNCHRONOUS) {
// Wait for an acknowledgement from the receiver when the communication is synchronous
        if (Key == -1) {
        // Wait for acknowledgement from all processes.
        GSize = pvm_gsize(GroupID);
        for (i = 0; i < GSize - 1; i++) {
          if (SocketType == IPCC_CLIENT)
            RECEIVER <int> __Receiver1(-1, SERVER_ACK, &ReturnVal, 1, 1);
          else
            RECEIVER <int> __Receiver1(-1, CLIENT_ACK, &ReturnVal, 1, 1); } }
      else {
        if (SocketType == IPCC_CLIENT) {
          RECEIVER <int> __Receiver1(-1, SERVER_ACK, &ReturnVal, 1, 1);}
        else {
          RECEIVER <int> __Receiver1(-1, CLIENT_ACK, &ReturnVal, 1, 1); }}}
    return 0; }
  else
    return -1;} [Dedh95]
```

Figure 6.11: Implementation Details of IPCC_Send_Internal() Method

- It broadcast a message to the group which contains the actual message.
- If the socket is configured for synchronous communication (rendezvous), then it suspends the invoking process by issuing a PVM receive message which waits for an acknowledgment response from the receiver process of the rendezvousing pair.

The details of the IPCC_Receive_Internal() communication method are illustrated in Figure 6.12. The general functionality of the IPCC_Receive_Internal() method performs the following steps.

- If the invoking process is a client, then it neglects all messages with ASYNC_REQUEST and SYNC_REQUEST; however, it sends an acknowledgment message (SYNC_RESPONSE) in response to the received SYNC_REQUEST message.
- If the invoking process is a server, then it neglects all messages with ASYNC_RESPONSE and SYNC_RESPONSE; however, it sends an acknowledgment message (SYNC_REQUEST) in response to the received SYNC_RESPONSE message.
- If the invoking process is configured as inactive and the socket is configured for bidirectional communication, then it must determine if it is expecting a matching key code. If a matching key code is expected, then it makes sure key codes match and that the message is sent from the correct process.

```
char *IPCC_SOCKET::IPCC_Receive_Internal(int *Length, int Key)
{ int ReturnVal;
  int TID;
  int bytes, msgtag, SentKey;
  char *Message;
  if (SocketType == IPCC_CLIENT) {
// Client neglects all asynchrounous request messages
    ReturnVal = pvm_nrecv(-1, ASYNC_REQUEST);
    while (ReturnVal > 0) {
      ReturnVal = pvm_nrecv(-1, ASYNC_REQUEST); }
// Neglect all synchrounous request messages, but send an acknowledgemnt
    ReturnVal = pvm_nrecv(-1, SYNC_REQUEST);
    while (ReturnVal > 0) {
      pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
      SENDER <int> __Sender1(TID, CLIENT_ACK, &ReturnVal, 1, 1);
      ReturnVal = pvm_nrecv(-1, SYNC_REQUEST);    }}
  if (SocketType == IPCC_SERVER) {
// Server neglects all asynchrounous response messages
    ReturnVal = pvm_nrecv(-1, ASYNC_RESPONSE);
    while (ReturnVal > 0) {
      ReturnVal = pvm_nrecv(-1, ASYNC_RESPONSE);    }
// Neglect all synchrounous response messages, but send an acknowledgemnt
    ReturnVal = pvm_nrecv(-1, SYNC_RESPONSE);
    while (ReturnVal > 0) {
      pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
      SENDER <int> __Sender1(TID, CLIENT_ACK, &ReturnVal, 1, 1);
      ReturnVal = pvm_nrecv(-1, SYNC_RESPONSE);    } }
  if (((SocketType != IPCC_INACTIVE) &&
              (SocketDirection == IPCC_BIDIRECTION)) ||
      (SocketType == IPCC_CLIENT)) {
    if (Key != -1) {
// look for key and get the tid of the process sending the key. Do a receive for a
message from that process.
      ReturnVal = pvm_recv(-1, Key);
      if (ReturnVal < 0) {
        pvm_perror(NULL);
        return(NULL);      }
      pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
      ReturnVal = pvm_recv(TID, -1);}
    else
      ReturnVal = pvm_recv(-1, -1);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(NULL);    }
    pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
    while ((Key != -1) && (msgtag < ASYNC_REQUEST || msgtag > SERVER_ACK))
    { // Neglect all key messages
      ReturnVal = pvm_recv(TID, -1);
      if (ReturnVal < 0) {
        pvm_perror(NULL);
        return(NULL); }
      pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID); };
    // Found a real data message so Get the key of the message
    ReturnVal = pvm_upkint(&SentKey, 1, 1);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(NULL);  }
// Check if the key in the message matches the key we are looking for.
    if (Key != -1) {
      while (SentKey != Key) {
        // Get the next message from it and check for the key. We have
        // to neglect this message.
        ReturnVal = pvm_recv(TID, -1);
```

Figure 6.12: Implementation Details of IPCC_Receive_Internal() Method (figure continued)

```
      if (ReturnVal < 0) {
        pvm_perror(NULL);
        return(NULL); }
      pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
      if (msgtag < ASYNC_REQUEST || msgtag > SERVER_ACK)
        continue;  // Neglect key message
      // Get the key of the message
      ReturnVal = pvm_upkint(&SentKey, 1, 1);
      if (ReturnVal < 0) {
        pvm_perror(NULL);
        return(NULL);    } } }
  // Get the length of the message
  ReturnVal = pvm_upkint(Length, 1, 1);
  if (ReturnVal < 0) {
    pvm_perror(NULL);
    return(NULL);    }
  // Allocate memory to store the message
  Message = (char *)malloc(*Length * sizeof(char));
  if (Message == NULL) {
    fprintf(stderr, "Memory Allocation Error \n");
    return (NULL);  }
  // Get the message contents
  ReturnVal = pvm_upkbyte(Message, *Length, 1);
  if (ReturnVal < 0) {
    pvm_perror(NULL);
    return(NULL);  }
  if (msgtag == SYNC_RESPONSE || msgtag == SYNC_REQUEST) {
    // Send an acknowledgement to the sender
    ReturnVal = 1;
    if (SocketType == IPCC_SERVER) {
      SENDER <int> __Sender1(TID, SERVER_ACK, &ReturnVal, 1, 1);}
    else {
      SENDER <int> __Sender1(TID, CLIENT_ACK, &ReturnVal, 1, 1);} }
  return Message;  }
else
  return NULL;} [Dedh95]
```

- Receives the message by unpacking the length, allocating dynamic memory, and storing the message contents in the allocated memory.

The details of the IPCC_Probe_Internal() communication method are illustrated in Figure 6.13. The general functionality of the IPCC_Probe_Internal() method follows the same procedure as the IPCC_Receive_Internal(), except that the message is not removed from the message receive buffer of that process. The distinction in functionality is summarized below:

- Probe into the message and receive the contents without actually removing the message from the message receive

```
char *IPCC_SOCKET::IPCC_Probe_Internal(int *Length, int Key)
{ int ReturnVal;
  int TID;
  int bytes, msgtag;
  char *Message;
  if (SocketType == IPCC_CLIENT) {
    // Neglect all asynchrounous request messages
    ReturnVal = pvm_nrecv(-1, ASYNC_REQUEST);
    while (ReturnVal > 0) {
      ReturnVal = pvm_nrecv(-1, ASYNC_REQUEST);    }
    // Neglect all synchrounous request messages, but send an acknowledgemnt
    ReturnVal = pvm_nrecv(-1, SYNC_REQUEST);
    while (ReturnVal > 0) {
      pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
      SENDER <int> __Sender1(TID, CLIENT_ACK, &ReturnVal, 1, 1);
      ReturnVal = pvm_nrecv(-1, SYNC_REQUEST);    } }
  if (SocketType == IPCC_SERVER) {
    // Neglect all asynchrounous response messages
    ReturnVal = pvm_nrecv(-1, ASYNC_RESPONSE);
    while (ReturnVal > 0) {
      ReturnVal = pvm_nrecv(-1, ASYNC_RESPONSE);    }
    // Neglect all synchrounous response messages, but send an acknowledgemnt
    ReturnVal = pvm_nrecv(-1, SYNC_RESPONSE);
    while (ReturnVal > 0) {
      pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
      SENDER <int> __Sender1(TID, CLIENT_ACK, &ReturnVal, 1, 1);
      ReturnVal = pvm_nrecv(-1, SYNC_RESPONSE);    } }
  if (((SocketType != IPCC_INACTIVE) && (SocketDirection == IPCC_BIDIRECTION)) ||
       (SocketType == IPCC_CLIENT)) {
    if (Key != -1) {
      // look for key and get the task id of the process sending the key.
      // Now do a probe for a message from that process.
      ReturnVal = pvm_probe(-1, Key);
      if (ReturnVal < 0) {
        pvm_perror(NULL);
        return(NULL);    }
      pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
      ReturnVal = pvm_probe(TID, -1); }
    else  // do a probe for any message from any task
      ReturnVal = pvm_probe(-1, -1);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(NULL);    }
    if (ReturnVal == 0)
      return NULL;
    pvm_bufinfo(ReturnVal, &bytes, &msgtag, &TID);
      // Get the length of the message
    ReturnVal = pvm_upkint(Length, 1, 1);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(NULL);   }
      // Allocate memory to store the message
    Message = (char *)malloc(*Length * sizeof(char));
    if (Message == NULL) {
      fprintf(stderr, "Memory Allocation Error \n");
      return (NULL);  }
      // Get the message contents
    ReturnVal = pvm_upkbyte(Message, *Length, 1);
    if (ReturnVal < 0) {
      pvm_perror(NULL);
      return(NULL);  }
      return Message; }
  else
    return NULL;
}[Dedh95]
```

Figure 6.13: Implementation Details of IPCC_Probe_Internal() Method

```
//Send Message
int IPCC_SOCKET::IPCC_Send(int *Message, int Length, int Key = -1)
{return IPCC_Send_Internal((char *)Message, Length * sizeof(int), Key);}
int IPCC_SOCKET::IPCC_Send(char *Message, int Length, int Key = -1)
{ return IPCC_Send_Internal((int *)Message, Length * sizeof(short), Key);}
                ...
int IPCC_SOCKET::IPCC_Send(short *Message, int Length, int Key = -1)
{ return IPCC_Send_Internal((char *)Message, Length * sizeof(short), Key);}

//Receive Message
int *IPCC_SOCKET::IPCC_Receive(int *Length, int Key = -1)
{ int *Message;
  Message = (int *)IPCC_Receive_Internal(Length, Key);
  return Message; }
int IPCC_SOCKET::IPCC_Receive(char *Message, int *Length, int Key = -1)
{ Message = IPCC_Receive_Internal(Length, Key);
  return (Message == NULL);}
                ...
int IPCC_SOCKET::IPCC_Receive(short *Message, int *Length, int Key = -1)
{ Message = (short *)IPCC_Receive_Internal(Length, Key);
  return (Message == NULL);}

//Probe Message
nt IPCC_SOCKET::IPCC_Probe(long *Message, int *Length, int Key = -1)
{ Message = (long *)IPCC_Probe_Internal(Length, Key);
  return (Message == NULL);}
int IPCC_SOCKET::IPCC_Probe(char *Message, int *Length, int Key = -1)
{  Message = IPCC_Probe_Internal(Length, Key);
   return (Message == NULL);}
                ...
int IPCC_SOCKET::IPCC_Probe(short *Message, int *Length, int Key = -1)
{  Message = (short *)IPCC_Probe_Internal(Length, Key);
  *Length = *Length / sizeof (short);
  return (Message == NULL);}
// This method sets value of SocketMode field. The default value is
// IPCC_ASYNCHRONOUS
void IPCC_SOCKET::IPCC_SetSocketMode(int Mode)
{ if (Mode == IPCC_SYNCHRONOUS)
    SocketMode = IPCC_SYNCHRONOUS;
  else  // for any other value set the socket mode to asynchronous
    SocketMode = IPCC_ASYNCHRONOUS; }
// This method sets value of SocketType field. The default value is IPCC_INACTIVE
void IPCC_SOCKET::IPCC_SetSocketType(int Type)
{ if (Type == IPCC_SERVER)
    SocketType = IPCC_SERVER;
  else if (Type == IPCC_CLIENT)
    SocketType = IPCC_CLIENT;
  else  // for any other value set the socket type to IPCC_INACTIVE
    SocketType = IPCC_INACTIVE;}
// This method sets value of SocketDirection field. The default value is
IPCC_UNIDIRECTION
void IPCC_SOCKET::IPCC_SetSocketDirection(int Direction)
{if (Direction == IPCC_BIDIRECTION)
    SocketDirection = IPCC_BIDIRECTION;
  else // for any other value set the socket type to IPCC_UNIDIRECTION
    SocketDirection = IPCC_UNIDIRECTION;} [Dedh95]
```

Figure 6.14: Implementation Details of IPCC_SOCKET Communication Methods and Methods to access Private Data

buffer by unpacking the length, allocating dynamic memory, and storing the message contents in the allocated memory.

Figure 6.14 illustrates the details of the IPCC_Send(), IPCC_Receive(), and IPCC_Probe() communication methods as well as methods necessary to modify the private data of the class. Each method returns the value of its corresponding internal communication method; that is, IPCC_Send() invokes the IPCC_Send_Internal() and returns its value. The corresponding action is performed for IPCC_Receive() and IPCC_Probe() methods as well. The methods to update the private data are IPCC_SetSocketMode(), IPCC_SetSocketType(), and IPCC_SetSocketDirection(). These methods modify the private data of IPCC_SocketMode, IPCC_SocketType, and IPCC_SocketDirection, respectively. The user accesses each of these methods when the IPCC_SOCKET defaults need to be re-configured for a particular communication need.

In summary, the IPCC_SOCKET implementation facilitates interprocess communication and supports indirect naming, socket configuration, and run-time error detection by utilizing features of the PVM system. For each socket object, there exists processes which have access to it. These processes form a communication group and use the IPCC_SOCKET as a means of communication. The dynamic group structure of PVM is used to broadcast messages to every member of the group. The PVM message tag feature is used internally by the IPCC++ environment to facilitate a form of direct naming by key code. It is also used to support internal message passing, that is implicit messages sent by the IPCC++ system not explicitly by the programmer. The expressiveness of IPCC++ is enhanced with

the ability to define the role of a process (client, server, or inactive) with the socket class rather than with the process class. It allows one process to perform different roles.

6.7 System Configuration and Testing Scenarios

The IPCC++ implementation was tested on an RS/6000 computer system which consisted of three virtual machines. The IPCC++ system was developed using the following software packages:

- GNU C++ version 2.6,
- PVM Software System version 3.3.2, and
- AIX Unix Operating System version 3.2.5.

A version of the Dining Philosophers problem (as defined in Section 4.3) has been tested. A process class of DP_Process was derived from the base class IPCC_PROCESS. Two different process class definitions where established: SERVER and PHILOSOPHER, both descendents of DP_Process. The DP_Process class declared a socket object and configured it for both the synchronous and asynchronous communication modes.

The client/server paradigm was tested by having the SERVER process object configure its socket object for a server role and PHILOSOPHER process objects configure their socket object for a client role. The socket object was configured for asynchronous communication to support the client/server paradigm. Indirect naming by socket handle was tested along with the probe language feature. This mode of communication supports many-to-one communication between process objects. The use of the key code was tested. It allowed two process objects associated with a socket object to communicate on a one-

to-one basis rather than broadcast to the group. The SERVER process selectively responded to different PHILOSOPHER objects by utilizing the matching key code feature associated with the communication features of the socket object.

Rendezvous communication between process objects was tested by configuring a socket object for synchronous communication. The socket object was configured by each process for an inactive role, that is, no established client or server role defined for each process. Each process was suspended until another process issued a matching communication request. The key code feature was tested for synchronous communication. It allowed different processes to be suspended and activated by communication methods based on matching key codes.

Various process activation methods were tested and their affect on process suspension observed. Activation methods that return values resulted in the suspension of the caller until the process executing the method completed. Activation methods that did not return a value resulted in no suspension of the caller; therefore, the newly created process and the caller proceeded in parallel.

This section described the physical implementation which supports the theoretical design of the IPCC++ language model. Chapter 7 contains a summary of the IPCC++ language and its research contributions. The future research directions for the language model are also discussed.

7. Summary

The IPCC++ language model represents a concurrent object-oriented language which supports centralized and distributed memory models. The principle of orthogonality is supported by introducing concurrency as objects. This method follows the true spirit of C++, representing a clean, natural approach for supporting concurrent programming. IPCC++ maintains the high level of abstraction associated with object-oriented languages by encapsulating the power of concurrent programming within objects. IPCC++ is a reliable and efficient software development environment that satisfies the need of software engineers for reliable software development while achieving the power of efficient software development associated with concurrent programming languages.

IPCC++ is a programming language, an extension of C++, designed to simplify the software solutions of concurrent programming problems by encapsulating them into the paradigm of object-oriented programming. It utilizes the PVM software system to support interprocess communication for centralized and distributed computing systems. IPCC++ provides a set of predefined class definition which support the monitor structure, condition variable, and socket application program interface.

The unique design of IPCC++ supports inter-object concurrency and encapsulates a single thread of control in a process object. The IPCC++ language model consists of the language C++ and a set of predefined components implemented

with C++ and interfacing with the PVM software system. IPCC++ supports the concept of process and mechanisms for process activation, termination, synchronization, and communication as complete objects.

An IPCC++ process represents a new thread of control executing within the program, possible on different processors (nodes in a network). IPCC++ supports static and dynamic process creation which forms a hierarchal structure with the relationship that the creating process is the parent of the created process. Inheritance is achieved with the hierarchical structure and is used to make synchronization objects (monitor object or socket object) visible to derived process class definitions. An asset of IPCC++ is that it supports separate process creation and instantiation. IPCC++ declares an object of the process type then subsequently directs the process to execute a method associated with that process object. The interprocess communication supports synchronous and asynchronous communication. Using a C++ class definition to support the process concept provides IPCC++ the capability to define different execution paths for different process types by using different derived class definitions.

The monitor object is used in conjunction with the condition object to facilitate interprocess communication and synchronization for centralized memory models. The IPCC++ design implements the condition class as a friend of the monitor class. The condition object is a component of the monitor object; therefore the monitor encapsulates the condition

object within the monitor that declares it and makes the condition object invisible outside of the monitor. The condition object represents a suspension condition of the monitor. A queue is asscciated with each condition object. This design eliminates unnecessary state transitions of processes by using different condition objects to express different suspension conditions. Coupling the synchronization methods with the condition object rather than the monitor object simplifies the monitor structure and provides greater expressiveness within IPCC++.

IPCC++ introduces the concept of socket as the vehicle used to provide an application program interface and to implement interprocess communication for distributed memory models. The IPCC++ socket object is not a communication endpoint but rather a class definition that provides communication methods of send, receive, and probe. The send, receive, and probe methods are designed to return an integer variable indicating whether communication is pending, communication has successfully completed, or a communication error has occurred. This design supports the client/server paradigm by offering the feature of selective waiting. It also supports communication error detection.

An IPCC++ socket represents an instance of the class IPCC_SOCKET. Using the C++ class definition to support the socket concept provides IPCC++ the capability to configure a socket for a specific communication task and supports the principle of orthogonality. An IPCC++ programmer uses the

derived class definition to specify the type of a message and other data attributes used to type check the messages. Coupling the synchronization methods of send, receive, and probe with the socket object rather that the process object provides greater expressiveness within IPCC++ by supporting indirect naming. The socket class defines two sets of send, receive, and probe methods, one set with a key parameter the other set without a key parameter. The key parameter is used to match communication requests and can be used to simulate direct naming.

Section 7.1 states the research contributions of IPCC++ with a brief description of each.

7.1 Research Contributions

The IPCC++ model advances the state of object-oriented concurrent languages by uniquely combining desirable object-orientation and concurrency into a single model. IPCC++:

- Supports the principle of orthogonality and uses the paradigms of object-oriented programming to introduce concurrency to C++.

> The primitive of concurrency are introduced to the C++ programmer using the primitives of object-oriented programming which allowing object-oriented programmers to focus on the semantics of concurrent programming, not the syntax. This approach simplifies concurrent programming while adhering to the principle of orthogonality by introducing concurrency as

complete objects and encapsulating the low-level system calls common to concurrent programming into high level constructs.

- Supports heterogeneous distributed computations.

  The environment of the IPCC++ language model makes use of the PVM software system. PVM supports heterogeneous interprocess communication which allows a network of computers (possibly using different operating systems) to appear as one computational resource.

- Supports centralized and distributed memory models.

  The IPCC++ environment supports interprocess communication for the centralized and the distributed memory models. It uses the monitor object and condition object to support centralized interprocess communication and the socket object to support distributed interprocess communication. IPCC++ is the only language in the centralized group (discussed in Section 2.2) to support both centralized and distributed memory models.

- Supports separate process declaration and activation.

  The process class structure of IPCC++ separates the declaration of a process from its activation. When a process object is declared, it is suspended until it receives a message indicating which method to direct its execution to. The separation of declaration and activation supports the principle of orthogonality by allowing the programmer to declare a process object using the same language construct for any object in C++.

  It uses the method access feature of C++ objects as a means of activating the process and indicating which method to execute. This feature supports inter-object concurrency and encapsulates at most one single thread of control within an object. Information hiding and encapsulating features of C++ are employed and the power of concurrency is achieved.

- Supports indirect naming of communication objects.

  The inheritance feature of C++ and the dynamic naming group feature of PVM are exploited to support indirect naming of communicating processes within IPCC++. When

process objects are to communicate, they are derived public from a common ancestor class which declares a communication object (monitor or socket). This method gives each process object a handle to a monitor object or socket object which in turn supports the interprocess communication. Therefore, processes can communicate solely by accessing a common communication object and do not need to know each others process identification. Indirect naming of processes in part is achieved by associating the send and receiving of messages with the socket object rather than the process object. This design is unique to IPCC++.

- Supports configuration of a socket object for synchronous or asynchronous communication as well as for a specific message type.

The inheritance feature of C++ allows the programmer the ability to customize the base socket class definition for a particular communication task. The programmer simply sets a flag indicating synchronous or asynchronous communication and declares the type of message to be supported by the socket object which is statically typed checked by the C++ typing system.

- Provides a language feature which supports viewing the contents of a message prior to actually receiving it.

  The socket class offers a probe() method which allows the programmer to peek inside of a message to view its contents without actually receiving the message. This expressive feature of IPCC++ allows the programmer to base receipt of messages on local constraints related to the message content.

- Supports the client/server programming paradigm.

  The socket class offers methods of send(), receive(), and probe() which are designed to return an integer variable of 0, 1, -1 to indicate communication is not pending, communication is successful, or communication error, respectively. This design supports selective waiting and the client/server paradigm.

IPCC++ achieves concurrency within C++ with the components of the IPCC++ language model and the support of the PVM software system. The design goal to producing a reliable and efficient software development environment while supporting orthogonality, utilizing complete objects, and exploiting inheritance has been achieved. The comparison to existing

research illustrates the unique approach of the design, the expressiveness of the language, and the features of concurrency introduced. As illustrated, IPCC++ extends existing research efforts in language design and concurrency features.

7.2 Future Research

Optimization techniques are elements of possible future research. Areas to optimize include: optimization in communication and data structures techniques. Optimization in communication refers to exploring techniques to reduce the number of implicit communications operations between processes, that is, communication expressed by the underlying system, rather than the programmer. Data structure techniques refers to exploring data structures and optimization techniques to optimize the internal data structures of the IPCC_ENVIRONMENT class and the IPCC_PROCESS_MANAGER class.

Another future research direction of IPCC++ is to capitalize upon the ability of the language to simplify concurrent programming (by hiding low-level system calls common with interprocess communication) and offer IPCC++ as a tool used to support the instruction of concurrency object-oriented programming.

REFERENCES

[Amer87] America, P., *POOT-T: A Parallel Object-Oriented Language*, Object-Oriented Concurrent Programming, Cambridge, Mass: MIT Press, pp. 199-220.

[Andr91] Andrews, Gregory R., *Concurrent Programming Principles and Practice*, The Benjamin/Cummings Publishing Company Inc., California,(1991).

[Atki91] Atkinson, Colin, *Object-Oriented Reuse, Concurrency and Distribution an Ada based Approach*, Addison-Wesley Publishing Company (1991).

[Beck90] Beck, B., *Shared-Memory Parallel Programming in C++*, IEEE Software, 38-48, (July 1990).

[BenA90] Ben-Ari, M., *Principles of Concurrent and Distributed programming*, Prentice Hall International (UK) Ltd, (1990)

[Begu94] Beguelin, A, Donagarra J, Geist, G, Jiang, W., Mancheck, R. Moore, K., Sunderanm, V., *The PVM Project*, Oak Ridge National Laboratory, Oak Ridge, TN, (1994)

[Bers88] Bershad, B.N., Lazowska, E.D., Levy, H.M., *PRESTO: A System for Object-oriented Parallel Programming*, Software and Practice and Experience 18(8),713-732 (August 1988).

[Blac87] Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L., *Distribution and Abstract Types in Emerald*, IEEE Transactions on Software Engineering, vol SE13, No. 1. Jan,, pp. 65-76, (January 1987).

[Borl90] Borland International Inc, *Turbo C++ Programmer's Guide*, Borland International Inc Scotts Valley CA, (1990).

[Buhr92] Buhr P.A., Ditchfield, G., Stroobossher, R.A., Younger, B.M., and Zanrke, C.R., *μC++: Con-*

currency in the Object-oriented Language C++, Software and Practice and Experience 22(2), 137-172, (February 1992).

[Chan93] Chandy, K. Mani, Kesselman, Carl, *CC++: A Declarative Concurrent Object-Oriented Programming Notation*, Research Directions in Concurrent Object-Oriented Programming, Cambridge, Mass: MIT Press, C1993 pp. 281-313.

[Cox86] Cox, B., *Object-Oriented Programming - An Evolutionary Approach*, Addison-Wesley Publishing Company (1986).

[Davi91] Davis, Stephen R., *Hands-On Turbo C++*, Addison-Wesley Publishing Company (1991).

[Dedh95] Dedhia, Anil & Jones, B., *Implementation of the IPCC++ Language Model for Distributed Processing*, Louisiana State University, Masters Project (1995).

[Dijk76] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ, 1976

[Geha88] Gehani, N.H. Roome, W.D., *Concurrent C++: Concurrent Programming with Class(es)*, Software and Practice and Experience 18(12), 1157-1177 (December 1988).

[Havi87] Havilan, Keith, Salama, Ben, *UNIX System Programming*, Addison-Wesley Publishing Company (1987).

[Hoar74] Hoare, C.A.R., *Monitors: an operating system structuring concept*, Comm. of the ACM 17, 549-557 (1988).

[Kafu90] *Kafura, Dennis,Lee, Keung Hae, ACT++: Building a Concurrent C++ with Actors*, JOOP: Journal of Object Oriented Programming, May/June Vol.3,No.1 1990 pp. 25 37.

[Kale93] Kale, Laxmikant V., Sanjeev, Krishnan, *CHARM++: A Portable Concurrent Object Oriented System based on C++,* SIGPLAN Notices Vol 28 Number 10 Oct 1993 pp. 91-108.

[Maek87] Maekawa, Oldehoeft, Oldehoeft, *Operating Systems Advanced Concepts,* The Benjamin/Cummings Publishing Company Inc., California, (1987)

[Meye87] Meyer, B., *Object-Oriented Software Construction,* Prentice Hall,Englewood Cliffs, NJ, (1987)

[Pohl89] Pohl, I., *C++ for C programmers,* The Benjamin/Cummings Publishing Company Inc., California, (1989)

[Stev90] Stevens, W. Richard, *Unix Network Programming book,* Prentice Hall,Englewood Cliffs, NJ, (1990)

[Sebe89] Sebesta, Robert W., Concepts of Programming Languages, Addison-Wesley Publishing Company (1989).

[Smit91] Smith,K.S., Chatteerjee A, *A C++ environment for Distributed Application Execution,* M C C Technical Report Number ACT-ESP-275-90. (1990)

[Stro86] Stroustrup, Bjarne, *The C++ Programming Language,* Addison-Wesley, Menlow Park CA. (1986)

[Stub95a] Stubbs, Shelly S., Carver D.L., Hoppe, Andrew *IPCC++: A concurrent C++ based on a shared-memory model,* Journal of Object-Oriented Programming, May 1995, Volume 8, No.2, pp. 45-50.

[Stub95b] Stubbs, Shelly S., Carver D.L., *IPCC++: A C++ Extension for InterProcess Communication with Objects,* Proceedings of Nineteenth Annual International Computer Software and Applications Conference, August 1995, pp. 205-210.

[Stub95c] Stubbs, Shelly S., Carver D.L., *Interprocess Communication with C++ for Distributed Memory Models,* To appear in Journal of Object-Oriented Programming.

APPENDIX A: Letter of Permission

03/17/95 09:53 FAX 2122427574 SIGS PUBLICATION 001

MAR-15-95 WED 15:16 P.01

SHELLY S. STUBBS
20470 Alexander Street
Covington, Louisiana 70435
Internet: [illegible]
(504) 892-6653

March 15, 1995

Journal of Object-Oriented Programming
Ms. Kristina Joukhadar
Managing Editor
P.O. Box 3000 Dept. OOP
Denville, N.J. 07834
Fax: 212-242-7574

Dear Ms. Joukhadar,

I would like to request copyright permission pertaining to the article *IPCC++: A Concurrent C++ for Shared-Memory Models* co-authored by Shelly S. Stubbs, D.L. Carver, and Andrew Koppe. This article is to appear in the May/June 1995 issue of JOOP. The information presented in this article represents research pertaining to my dissertation. This request is for permission to use portions of the article within my dissertation.

Due to time constraints, I would like to request the response letter be faxed to (504) 893-7928, attention Shelly S. Stubbs. If this option is not available, please mail the letter to the above address.

Sincerely,

Shelly S. Stubbs

3/17/95

Dear Ms Stubbs:

We herewith grant permission as requested above. Please kindly acknowledge the JOOP article somewhere within your dissertation if possible.

Thank you for publishing with us!

Kristina Joukhadar

03/17/95 15:17 TX/RX NO.0544 P.001

APPENDIX B: Letter of Permission

Shelly S. Stubbs
20470 Alexander Street
Covington, LA 70435
(504) 892-8653
Fax: (504) 893-7928

November 10, 1995

IEEE Service Center
P.O. Box 1331
Piscataway, NJ 08855-1331
Fax: 908-981-8062

RE: Copyright Permission

TO WHOM IT MAY CONCERN:
PERMISSION, AS REQUESTED,
IS HEREBY GRANTED.
William S. Hagen
WILLIAM S. HAGEN, MANAGER
IEEE COPYRIGHTS

To Whom It May Concern,

I would like to request copyright permission pertaining to the article: *IPC++: A C++ Extension for InterProcess Communication with Objects* co-authored by Shelly S. Stubbs and D.L. Carver. This article was published in the Proceedings of The Nineteenth Annual International Computer Software & Applications Conference held in Dallas, Texas, August 1995. The information presented in this article represents research pertaining to my dissertation. This request is for permission to use portions of the article within my dissertation.

Due to graduation time constraints, I would like to request a response be faxed to (504) 893-7928, attention Shelly S. Stubbs. I need this as soon as possible so that I can graduate this semester. If this option is not available, please mail the letter to the above address. Thank you for your assistance with this matter.

Sincerely,

Shelly S. Stubbs

APPENDIX C: Letter of Permission

Shelly S. Stubbs
20470 Alexander Street
Covington, LA 70435
(504) 892-8653
Fax: (504) 893-7928

November 10, 1995

Journal of Object-Oriented Programming
Ms. Kristina Joukhadar
Managing Editor
P.O. Box 3000 Dept. OOP
Denville, NJ 07834
Fax: 212-242-7574

Dear Ms. Joukhadar,

I would like to request copyright permission pertaining to the article: *Interprocess Communication with C++ for Distributed Memory Models* co-authored by Shelly S. Stubbs and D.L. Carver. We received notification of the acceptance of this article early this month. The information presented in this article represents research pertaining to my dissertation. This request is for permission to use portions of the article within my dissertation.

Due to graduation time constraints, I would like to request a response be faxed to (504) 893-7928, attention Shelly S. Stubbs. I need this as soon as possible so that I can graduate this semester. If this option is not available, please mail the letter to the above address. Thank you for your assistance with this matter.

Sincerely,

Shelly S. Stubbs

11/13/95

Dear Ms. Stubbs:
We herewith grant you permission requested above. We would appreciate your reference to the publication of the original article by SIGS Publications if possible. Thank you very much for publishing with us, and best of luck to you!

K. Joukhadar

11.13.95 MON 13:34 FAX 212 242 7574 SIGS PUBLICATIONS

VITA

Shelly S. Stubbs received her B.S. degree in Computer Science from Southeastern Louisiana University in 1985. She has been an employee of Medtron Intelligence Corporation since 1985. Mrs. Stubbs contributes to the design and implementation of medical software applications for physicians and hospitals as well as performing consulting in the field of computer science for independent clients. She is also currently employed by Southeastern Louisiana University as an instructor in Computer Science. Her current research interests include object-oriented programming languages, distributed programming languages and architectures, and operating system.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Shelly S. Stubbs

Major Field: Computer Science

Title of Dissertation: IPCC++: A Concurrent C++ for Centralized and Distributed Memory Models

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

November 7, 1995