The Implementation of ACT++ on a Shared Memory Multiprocessor

by

Manibrata Mukherji

Project submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science and Application

APPROVED:

Dennis Kafura

Dennis G. Kafura, Chairman

Marc Abrams

Marc Abrams

Calvin J. Ribbens

Calvin Ribbens

February, 1992

Blacksburg, Virginia

The Implementation of ACT++ on a Shared Memory Multiprocessor

by

Manibrata Mukherji

Committee Chairman: Dr. Dennis Kafura, Computer Science

ABSTRACT

ACT++ is a programming environment in which concurrent programs can be written in C++. The underlying model of concurrent computation is the Actor model. Programs in ACT++ consist of a collection of active objects called actors. Several actors execute simultaneously and cooperate with each other by sending request and reply messages. Request messages are processed by agents called behaviors of the actor. Each behavior of an actor is responsible for specifying a replacement behavior which processes the next available request message. One of the salient features of ACT++ is its ability to handle the Inheritance Anomaly - the interference between the inheritance mechanism of C++ and the specification of synchronization constraints in the methods of a class - using the notion of behavior sets. Another feature of ACT++ is its realization of I/O as an actor operation. A special type of actor, called an interface actor, provides a high level interface for a particular device which is sent request messages whenever I/O is necessary. The interface actors can also transparently perform asynchronous I/O. ACT++ has been successfully implemented on the Sequent Symmetry multiprocessor using the PRESTO lightweight threads package.

Acknowledgments

First, I would like to thank my advisor for giving me the opportunity to work in this project which has been an enlightening experience for me in many ways. I am grateful to him for the time and effort he has put into every aspect of the project since its start. Second, I would like to thank Dr. Verna Scheutz for encouraging and motivating me. Third, I would like to thank the members of my family for encouraging me to pursue graduate studies.

Finally, I would like to thank my wife for being a constant source of inspiration and encouragement, especially during the most difficult periods of my graduate studies.

Contents

Figures

Chapter Two:

Chapter Three:

Chapter One

Introduction

1. Object-Oriented Concurrent Programming and ACT++

The object-oriented concurrent programming (OOCP) paradigm combines the flexibility and modeling features of the object-oriented approach with the computational power associated with parallelism. In the OOCP approach, a program is comprised of a collection of independent objects which execute concurrently and communicate with each other via message passing. While each object is self-contained, hiding its internal details from the rest of the system, the group of objects cooperate to solve a single problem.

If multiple processors are available in the system, the concurrent objects can be distributed over them so that the operations in different objects can execute in parallel. If only a single processor is available, the processes execute in a multiplexed fashion thereby incurring the cost of context-switching. The recent availability of affordable parallel computers based on off-the-shelf microprocessors, developments in the field of networking and distributed computing, and availability of software systems that use *threads* or "lightweight" processes to exploit parallelism has enabled OOCP to be implemented successfully on parallel and distributed systems. The parallel systems are usually MIMD computers that are of the shared memory type or are a mix of shared and distributed memory types. The distributed systems consist of independent, possibly heterogeneous, computers connected via a network.

To implement a OOCP system, one needs the availability of an object-oriented programming language (OOPL), a model of parallel computation, the support of an underlying software system that will enable the management of concurrency, along with, preferrably, a parallel or a distributed computer system. This underlying support software could either be the operating system or some other system software enabling the creation and the management of concurrency. Alternative OOCP systems depend on the choice of the language and the model. One implementation strategy is to develop a compiler for a new OOPL whose semantics reflect the features of a new parallel computation model. An alternative strategy is to utilize an existing OOPL to implement an existing model of parallel computation. The latter strategy was followed in developing ACT++ in order to reduce cost and time, increase the flexibility of the system, increase the portability of the system to different machines, and improve its availability to other researchers.

The OOPL that was chosen for the purpose was C++ version 1.2 [Stroustrup 86] and the model of parallel computation that was chosen was the Actor model [Agha 86, Agha and Hewitt 87]. The programming environment that resulted from implementing Actors in C++ is named ACT++ [Kafura and Lee 90]. One reason for choosing C++ as the base language was that its support of object-oriented programming allowed for the mapping of the object-based features of the Actor model directly into ACT++. Another reason for the choice of C++ is that it supports inheritance; this facilitated the study of the interaction between concurrency and inheritance. Interesting aspects of this interaction are reported in [Kafura and Lee 89].

As far as the underlying software system and the computer architecture are concerned, one line of development utilizes the UNIX operating system and the single-processor Macintosh II computer to build a prototype of ACT++ [Lee 90]. A second line of development utilizes the ESKIT system [Joshi 92] to develop a prototype of ACT++ on a network of SUN workstations. The former development explored the issues in implementing a OOCP environment using the tools provided by UNIX to manage pseudo-parallelism on a single processor, whereas the latter development explored the issues in implementing a truly parallel system in which different parts of an application could be physically distributed over different computers.

The goal of the current project was to explore the issues in implementing a prototype of ACT++ on a shared memory multiprocessor. The shared memory computer that was chosen was the 10 processor Sequent Symmetry computer which is available in the Department of Computer Science at Virginia Tech. The underlying software system that was chosen for implementing the concurrency management aspects of ACT++ was PRESTO version 0.2 [Bershad et al 88]. PRESTO is a *threads* package implemented in C++ version 1.2 which runs on the Symmetry multiprocessor and the SUN workstations. The advantages of PRESTO were the availability of *threads,* which have less overhead than UNIX processes, and the use of C++ as the language for developing PRESTO, which matched our choice of the base language in ACT++. Using a minimal subset of the PRESTO features and affecting the minimal possible changes to the PRESTO system, we have successfully implemented ACT++ on the Symmetry system.

In the rest of this chapter we discuss the Actor model of computation on which ACT++ is based and specify the major deviations of the ACT++ model from the original model. A brief survey of several other Actor-based concurrent OOPLs is then made. In chapter two we provide a user manual for ACT++ and discuss the interface of the various classes that are available to ACT++ programmers. In chapter three we discuss the implementation of ACT++. Chapter four concludes the report by summarizing our work and laying out some directions for future work.

2. The Actor Model of Computation and ACT++

The details of the Actor model of computation is explained in [Agha 86, Agha and Hewitt 87]. In the following we summarize the salient features of the model in order to bring out the differences between the original model and the one that has been implemented in ACT++.

The Actor model attempts to integrate the attractive features of the object-based and the functional paradigms of concurrent programming. It supports shared objects that may change their internal state, objects which cannot be implemented in purely functional systems. It provides the ability to create objects and reconfigure their interconnection topology dynamically. Like the functional paradigm, the Actor model also provides for inherent concurrency, the ability to extract concurrency by merely inspecting the structure of programs instead of extensive reasoning about the language constructs used to specify the program. By disallowing the assignment statement, all subexpressions in a program can be executed concurrently.

The main features of the Actor model are as follows.

- The primary agents responsible for all activity in a program are active objects called *actors*. The actors are active objects because they can initiate activity in themselves. All objects in the system are actors.

- Several actors can be computing at the same time.

- Actors communicate via asynchronous messages which are called *request messages*. As such, each actor has the ability to buffer messages in a queue called the *mail queue*. An actor is identified by its mail queue address, which can be passed like any other item in messages.

- An actor might know (i.e. hold the mail address) about all or a subset of the other actors in the system which are called its *acquaintances*.

- Request messages contain requests for the execution of program segments in an actor called *behaviors*. Each behavior is responsible for processing exactly one request message. Request messages are processed by an actor in the order of their arrival, but the arrival order is nondeterministic.

- The processing of a request message by an actor could lead to the occurrence of one or all of the following actions:

  - send request messages to itself or other actors,
  - create more actors, and
  - specify the replacement behavior that would process the next message.

- Another type of message that is exchanged between actors is called a *reply message*. As a result of a request message to it, the requested actor sends a reply message to the requestor with the result. The requestor becomes insensitive (i.e. does not process any of the pending messages in its mail queue) if it needs a reply message which has not yet been received. The reception of the reply message enables an insensitive actor to continue with its computation.

The specification of a replacement behavior is a significant operation in the actor model that requires some explanation. A replacement behavior is specified using the *become* operation. The replacement behavior is responsible for processing the next request message in the actor. Since the functionality and the structural composition of the replacement behavior can be different from the one that specified it, replacement behavior specification is the means by which an actor changes its local state. Although replacement specification logically creates an actor with a new behavior in the system, it does not invalidate all the pointers to the actor held by other actors in the system. This is how the *referential transparency* of identifiers is maintained in the Actor paradigm.

Another important property of replacement specification is that it leads to intrinsic concurrency in the system. Since the specification of the replacement behavior can be carried out with all other actions in the actor, and the replacement behavior is completely independent of the current behavior of the actor, the replacement behavior can immediately start processing an available request message.

The choice of C++ as the base language for implementing ACT++ and the goal of making an efficient implementation prevented us from utilizing all the features of the Actor model. In the following we list the major deviations of the ACT++ model of computation from the original Actor model.

- As a result of the choice of C++ as the base language, the assignment statement can be used anywhere in ACT++ and the program does not consist of expressions and subexpressions which can be executed concurrently as specified in the original model. Instead, in ACT++, behavior scripts are considered to be pieces of code that are executed sequentially. Activating multiple actors by sending messages to them and the specification of replacement behaviors are the only means of achieving concurrency in an ACT++ program.

- An ACT++ program does not consist only of actors. Objects of all the basic types available in C++ along with those of any user-defined type coexist with actors in an ACT++ program.

- A mail queue addresse which is a unique identifier of an actor in the Actor model is implemented as the object pointer returned by C++ on the creation of an actor object.

- Acquaintances have no special representation in ACT++ in the sense that there is no object in ACT++ that can be distinguished as an acquaintance. Instead, an acquaintance is simply a pointer to an actor object. Acquaintances can be passed to other actors in messages just like any other pointer variable. The lack of a proper representation of acquaintances in ACT++ makes it impossible to construct a garbage collector. The representation of acquaintances and the garbage collection problem will be addressed in a future version of ACT++.

- Reply messages are not handled via insensitive actors. Instead, an object called a Cbox is used to buffer reply messages. Cboxes are defined with blocking semantics so that an actor blocks when it tries to extract a reply message from an empty Cbox.

- The arrival order of request messages is deterministic - the order of arrival of two messages to an actor is the same as the order in which they were sent.

- Behaviors in ACT++ are not restricted to process request messages in the strict order of their arrival. Instead, behaviors can consult the values of their instance variables to decide which request message to process next. Accordingly, a request message anywhere in the mail queue may be selected for processing by a behavior.

3. Other Actor-Based Languages

Several other concurrent languages and programming environments have chosen the Actor model of computation as the underlying model of concurrency. Just as in ACT++, many of these languages incorporate only partially the features of the Actor model either because of the choice of the base language or the requirements of the intended application domain. Many of these languages did not use an existing language as the base but defined a new language. In the following we discuss the features of some actor-based languages briefly.

The Actra language [Barry et al 87, LaLonde et al 86] uses Smalltalk [Goldberg and Robson 83] as the base language for implementing a concurrent actor-based language that runs on a shared memory multiprocessor. The code sharing mechanism in Actra is inheritance. Its intended domain of application is real-time systems. Among the salient features of Actra is that it lacks a *become* operation which is used to specify a replacement behavior in the Actor model. Moreover, message passing in Actra is unbuffered and synchronous in contrast to the asynchronous and buffered message passing semantics in the Actor model. Thus, Actra does not implement many of the basic features of the Actor model.

Actalk [Briot 89] is an actor-based concurrent language that uses Smalltalk-80 as the base language. Actalk runs on a uniprocessor system. Its intended use is for classifying and simulating other actor-based languages in a single framework. The message passing semantics in Actalk is asynchronous. Sharing of code is allowed through inheritance.

Lamina [Delagi and Saraiya 88] is an actor-based concurrent language which is based on Lisp and runs on a multiprocessor computer. The intended usage of the language is the application of artificial intelligence techniques to real-time problems. Message delivery in Lamina is nondeterministic and receipt of a message triggers an operation inside an object. There is no counterpart of the *become* operation in Lamina and there is no sharing mechanism defined.

Act3 [Agha and Hewitt 87] is a concurrent actor-based language which is based on Lisp and has been implemented on the Apiary architecture - a concurrent architecture based on a network of Lisp machines. The specification of replacement behaviors is done using the *become* operation. The message passing semantics is buffered and asynchronous as defined in the Actor model. Sharing is supported through the delegation mechanism.

ABCL/1 [Yonezawa 90] is an actor-based concurrent programming language which executes on a distributed system. It is based on a new language definition which incorporates all the features of Lisp and is intended for distributed applications. The message passing semantics is buffered and asynchronous. There are three types of messages, namely, *Past*, *Now*, and *Future*. Message passing can be done in two modes - ordinary and express. Express mode messages allow preemption of the current thread of control in an object. Reply messages are handled by the creation of future objects. Sharing is supported through the delegation mechanism.

Chapter Two

ACT++ User's Manual

1. Overview of the ACT++ Computation Model

The model of computation supported by C++ is strictly a sequential one where a single thread of control is used to execute methods in different objects at different points in time. The goal in designing ACT++ was to use C++ to build a *concurrent object-oriented* programming environment - an environment which uses the language features of C++, utilizes some features and program structuring techniques of the **Actor model** of computation, and utilizes a *threads package* to implement concurrently executing objects. The features of the actor model that were found useful to extend C++ to support concurrent programming have been implemented in ACT++. The notion of actors not only provided the ability to model *active objects* in ACT++ but also made the modeling of the dynamic nature of real-life objects possible through the notions of object state and behaviors.

1.1 Actors, Behaviors, and Messages

Actors are active objects which receive messages and have the potential of exhibiting different interfaces at different points in time. An interface is a subset of all the operations that the actor can perform. At any time an actor exhibits only one interface and the semantic processing associated with this interface is called the *current behavior* of the actor. The current behavior of the actor is responsible for processing a single message. In processing a message the current behavior can send messages to other actors it knows about (its *acquaintances*) or itself , create new actors, and specify the *replacement behavior* which will be responsible for handling another message to the actor. Thus a behavior is responsible for ensuring that the actor retains the ability to process subsequent messages by specifying a replacement behavior.

Messages play two roles in the actor model. First, messages are the means of communication among actors. The computation performed by an actor system is driven by the passing of messages. Second, the execution of an actor's current behavior is initiated by the availability of a message requesting the execution of some operation in that behavior's interface. The messages are buffered in a message queue in the actor and, in the primitive actor model, are processed in a First-In First-Out (FIFO) order by the different behaviors of the actor. The default FIFO message processing order has been extended in ACT++ to provide more control over the selection of messages.

Unlike the Actor model where every object is an actor, in ACT++, there are two types of objects - **active** and **passive**. The distinction between the two types is that when active objects process a message they create an independent thread of control to execute the requested operation, whereas passive objects process a message using the thread of control of the requestor. Thus, an ACT++ program is a coherent collection of active and passive objects - active objects execute independently and concurrently with other active objects whereas the passive objects act as subordinates of the active objects. The object creation and method execution facility provided in C++ suffices to implement the class of passive objects in ACT++. Special classes have been introduced to implement active objects, the actors, and the asynchronous method invocation associated with them.

Each of the three primary notions involved in programming with actors (actors, behaviors, and messages) are instantiations of predefined classes in ACT++. Of the three types of objects, actors are active whereas both behaviors and messages are passive. This means that only actors have the ability to schedule independent threads of control to process request messages whereas the message and behavior objects lack that ability.

To write an ACT++ program, first, one has to determine the different actors, behaviors and passive objects that will be necessary to perform the intended action of the application. Corresponding to each passive object the user will provide a regular C++ class definition defining the instance variables and methods of the class. Corresponding to each behavior the user will define a subclass of a predefined base class called **Behavior.** Each user-defined behavior class will have, as usual, instance variables and method definitions. Each actor is created as an instantiation of the predefined class **Actor**. Each message that is sent between the actors is also created as an instantiation of the predefined class **Message**.

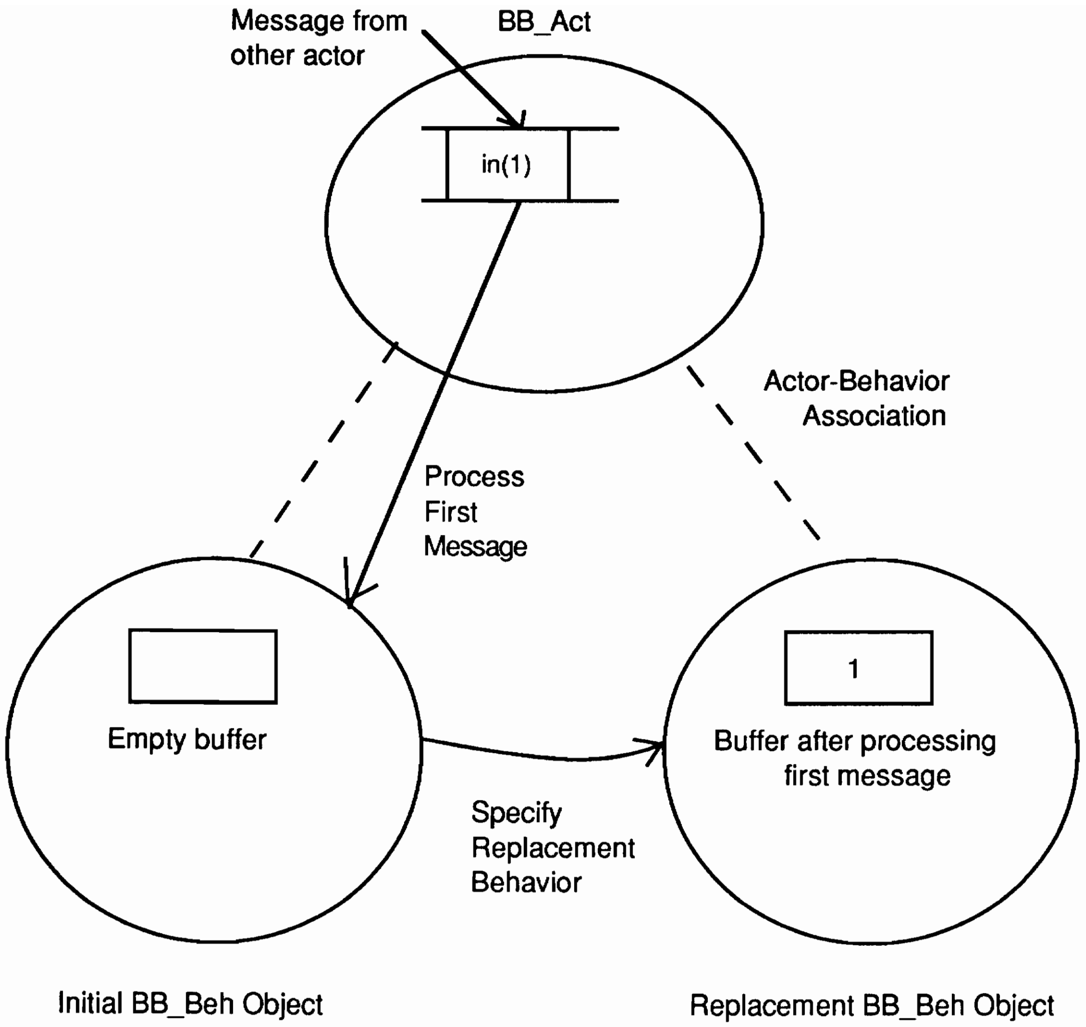

Message from other actor
BB_Act
in(1)
Actor-Behavior Association
Process First Message
Empty buffer
1
Buffer after processing first message
Specify Replacement Behavior
Initial BB_Beh Object
Replacement BB_Beh Object

Figure 1: Capturing the interaction between actors, behaviors, and messages. The BB_Beh object processes the first message object sent to the BB_Act object and specifies a replacement behavior object that wil process any subsequent message.

Let us consider an example to illustrate how the notions of actors and behaviors can be used to solve a practical problem. We will consider the bounded-buffer problem in which a buffer of a finite size is being read and written concurrently by multiple independent agents. One way to model the bounded buffer is to consider it to be an actor to which other actors send messages in order to either read or write information. As a result, we will instantiate a bounded-buffer actor object, called *BB_Act*, as shown in **Figure 1**. The actor object will be assigned an instantiation, *BB_Beh*, of the user defined behavior class, *Bounded_Buffer*, as its *initial behavior*. The *Bounded_Buffer* class has two methods in it, *in* and *out*. *In* adds one item into the buffer and *out* extracts one item from the buffer. A private member in the *BB_Beh* object realizes the bounded buffer. In the example in Figure 1, the first message requests the execution of the *in* method of *BB_Beh* with an argument of 1. After processing the message, *BB_Beh* specifies the replacement behavior that will be responsible for processing any subsequent message to the actor. Note that the instance variable representing the bounded buffer in the replacement *BB_Beh* object has a new value of 1 due to the processing of the first message by the initial behavior.

1.2 Replacement Behaviors

Specification of replacement behaviors is the means by which actors retain their ability to process messages. That is done in ACT++ by the *become* operation. The current behavior of an actor is responsible for specifying the replacement behavior that selects the next message to be processed. In ACT++, the replacement behavior object is always an instantiation of the *same class as the current behavior*. Technically, assigning objects of different behavior classes as replacement behaviors is not impossible. But, conceptually, an actor serves a unique purpose and the functionality of an actor is modeled by its behaviors at different points in time. For example, the *BB_Act* actor in Figure 1 is meant to serve as a bounded-buffer and hence all its behaviors are instances of the same behavior class, *Bounded_Buffer*, which provides the necessary operations related to managing a bounded-buffer. Therefore, each actor should be limited to providing a single service which is realized by the methods of a single behavior class.

There are two variations of the *become* operation. The first one is used to create a new object of the same class as the current behavior which processes the next message. The second variation marks the current behavior object as the replacement behavior instead of creating an entirely new object. The placement of the *become* operation in the methods of the behavior classes along with its two variations give rise to the following six possibilities for exploiting concurrency in actors.

	become as the last operation	*become* as the first operation	*become* anywhere in the middle
same object	serialized actor	intra-behavior concurrency	intra-behavior concurrency
new object	serialized actor	intra-actor concurrency	intra-actor concurrency

In the matrix above, the columns represent the possibilities of placing the *become* at the very beginning, very end, and anywhere between those two extremes in a method. The rows represent the possibilities of reusing the current behavior object versus creating a new behavior object. Since executing a *become* operation triggers the processing of any subsequent message to the actor, the placement of the *become* operation determines whether an actor will process its messages in a **serialized** or **concurrent** manner. A *serialized actor* processes its messages sequentially; only one message is processed at any point in time. A *concurrent actor*, on the other hand, has the ability to process more than one message simultaneously.

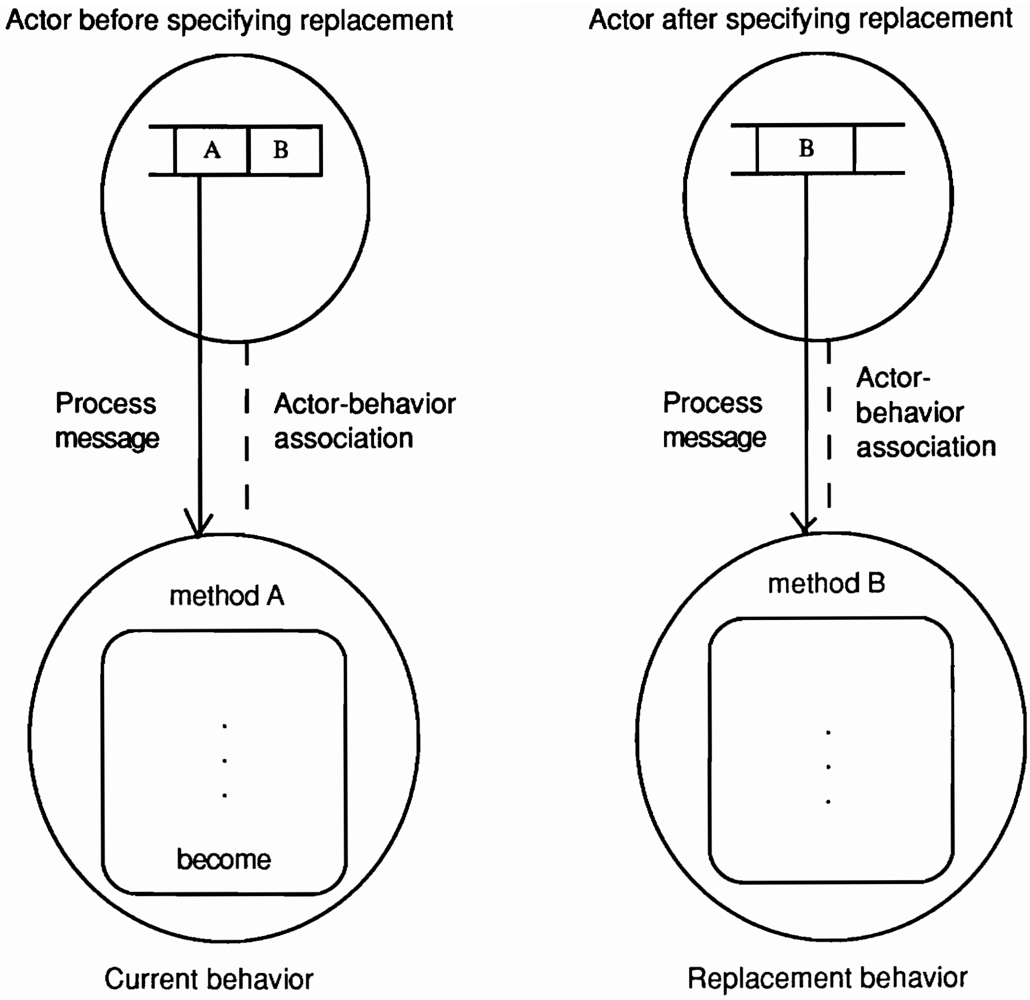

Actor before specifying replacement
Actor after specifying replacement
A
B
B
Process message
Actor-behavior association
Process message
Actor-behavior association
method A
become
method B
Current behavior
Replacement behavior

Figure 2: The operation of a serialized actor. The become is done as the very last operation in a method of the current behavior which creates a replacement behavior to process the next message.

The different options in the placement of the *become* operation in a method are used to avoid the *interference* problem - the simultaneous accessing of shared data by multiple independent threads of control. The shared data items involved in the interference are the *state variables* of the actor - the instance variables of the current behavior of the actor. The goal is to prevent the processing of any subsequent message to the actor until all the assignments to the state variables of the actor by the current behavior is complete. This ensures that there will be no interference when two or more behaviors are in execution simultaneously in the same actor. In a serialized actor the assignment to state variables continues right upto the end in every method and hence the replacement behavior can be specified only as the last statement in the current behavior. This case is shown pictorially in **Figure 2**. In actors whose behaviors are completely independent of each other in the sense that they do not access any shared state variable, the *become* can be specified as the very first operation in every method so as to exploit maximum concurrency. The third case arises when the behaviors of an actor are not completely independent, yet they can do a substantial part of their processing independently. In such a case the *become* can be placed anywhere in the body of a method.

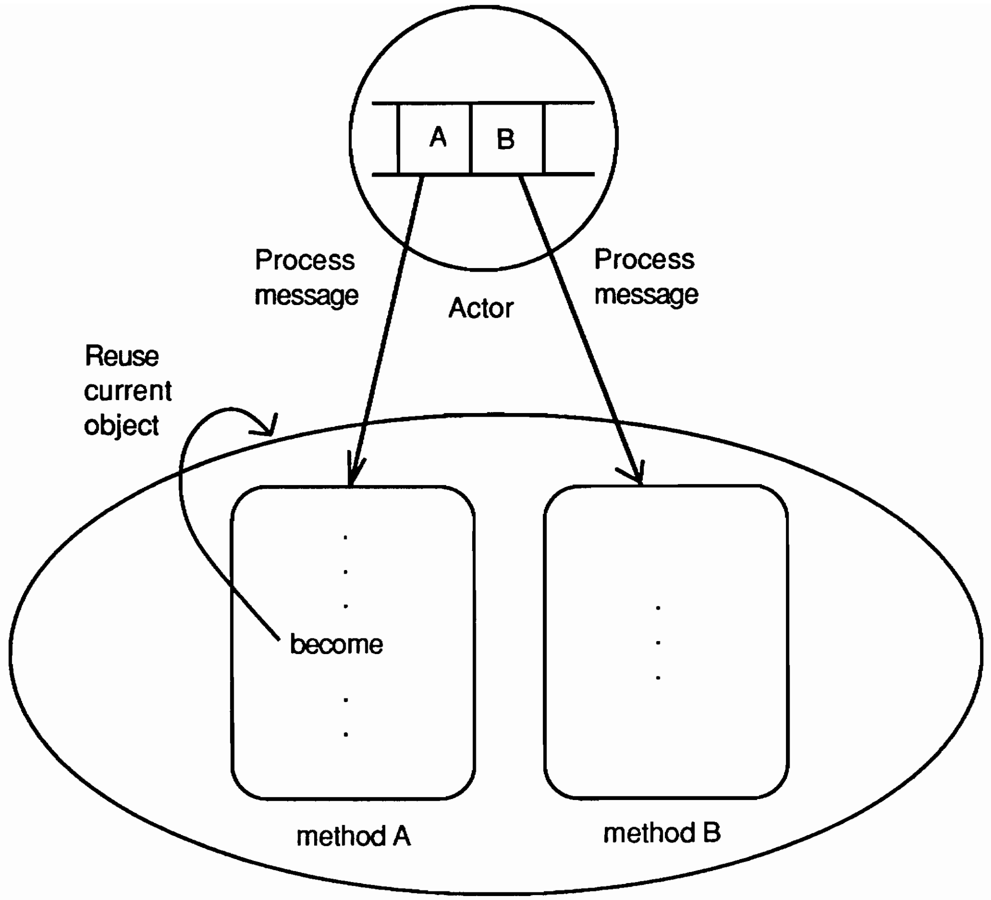
Note that the above is a conservative solution to the interference problem; introducing critical sections in order to synchronize access to shared variables could exploit more concurrency in an actor. The problem with such solutions will be apparent in the next section when we discuss the clash between using synchronization to control concurrency and the inheritance mechanism in object-oriented languages.

Concurrent message processing in actors manifests itself in two ways depending on which of the two variations of the *become* operation is used as shown in the above matrix:

- **intra-behavior concurrency** - more than one thread of control executing simultaneously inside the same behavior object, each thread of control representing the processing of a unique message, and

- **intra-actor concurrency** - more than one behavior executing simultaneously inside the same actor each processing a unique message.

Intra-behavior concurrency is realized by specifying the current behavior object as the replacement behavior. The replacement behavior, on the availability of a message, becomes active immediately, thereby introducing the possibility of multiple threads of control in the same behavior object. This is shown pictorially in **Figure 3**. Note that although the same behavior *object* is reused in this case, logically there exist two distinct behaviors, one responsible for processing message A and the other responsible for processing message B. One has to make this distinction to be consistent with the Actor model which stipulates that one behavior can process only one message.



Current and replacement behaviors in the same object

Figure 3: Operation of a concurrent actor exhibiting intra-behavior concurrency. Using a become operation to specify the current behavior object as the replacement behavior before the end of method A allows the simultaneous processing of two messages by the same behavior object.

Intra-actor concurrency is realized by creating a **new** behavior object as the replacement behavior. The new object, on the availability of a message in the actor's queue, becomes active immediately, thereby executing concurrently with the current behavior, if that is still executing. This is shown pictorially in **Figure 4**.

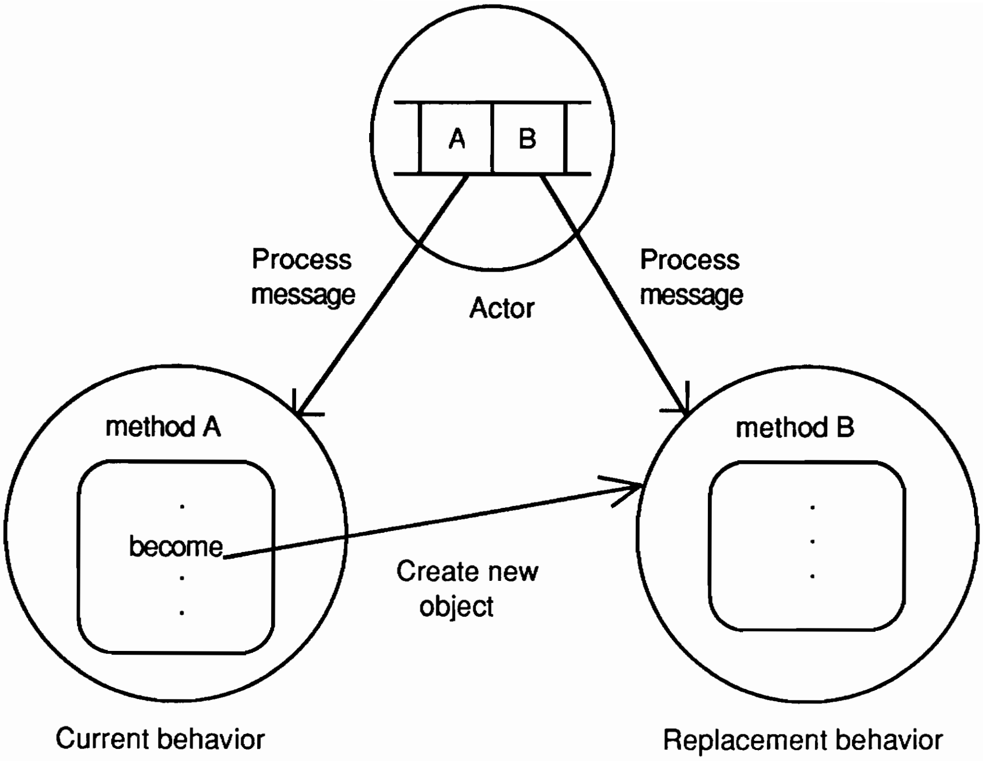


Figure 4: The operation of a concurrent actor exhibiting intra-actor concurrency. Using a become to create a new behavior object before the end of method A allows the simultaneous processing of messages A and B by two behaviors.

A final note about the cost of specifying new replacement behavior objects versus reusing the current behavior object. Creating a new behavior object is associated with time and memory overheads of object creation which is absent in the other case. Yet synchronization and concurrency considerations might lead one to specify new replacement behaviors quite often. Although there is an object creation overhead involved, new replacement behavior objects are destroyed automatically by the ACT++ run time system after they have processed a message.

1.2.1 Null Replacement Behaviors

The replacement behaviors discussed in the previous section are the agents that ensured the continuity of the message processing ability of actors. What if an actor does not want to process any more messages? It must be allowed to do so by specifying a null replacement behavior whose action would be to mark the actor as "dead" and delete any subsequent request message sent to it. Any request message sent to a "dead" actor with the expectation of getting getting any computation done will not be successful.

1.3 Behavior Sets

The notion of behaviors has been further embellished in ACT++ by introducing the concept of **behavior sets** [Lavender and Kafura 90]. Using only the concepts introduced thus far, the only difference between the current behavior and the replacement behavior of an actor is in the *state of the actor* (i.e. a particular assignment of values to the set of state variables in the current behavior of an actor). The same set of operations is available for execution irrespective of the state in which the actor is in. This property of actors is not conducive to modeling real-life objects that dynamically change their interface depending on their internal state. In order to model such dynamic properties using actors, one has to incorporate synchronizations involving the state variables in the methods of the behavior in order to simulate selective message processing. Such techniques yield cumbersome solutions that become an obstacle to the *inheritence mechanism* of object-oriented languages.

To be explicit, let us consider the *Bounded_Buffer* behavior once again. When the buffer is empty the *out* method is inapplicable and its execution due to a request message can be delayed by an explicit test in the method itself. While this is a plausible solution it introduces a problem called the **Inheritence Anomaly** [Matsuoka et al 90, America 87, Briot and Yonezawa 90, Kafura and Lee 89]. When the *Bounded_Buffer* class is subclassed, problems arise due to the state dependent synchronization code in the body of *out*. As a result, one has to provide a complete redefinition of the *out* method in the subclass which in turn renders the *out* method in the baseclass useless for any subsequent derived classes.

The solution proposed to this problem in [Lavender and Kafura 90] is to separate the state dependent synchronization code from the methods of a behavior and consider it to be a separate entity subject to explicit manipulation. The solution is believed to be free of any known Inheritence Anomaly. The specific notions introduced to implement the behavior set scheme are *behavior sets*, a *next behavior set function*, and *object-state functions*.

A *behavior set* is an object which is an instance of a predefined class called **Behavior_Set.** Each behavior object is associated with a single *behavior set.* A *behavior set* object contains the method identifiers of a subset of the methods of the class of which the behavior object is an instance. The behavior set represents the interface of the behavior object during the selection of a request message. A behavior will execute a message only if the method in the request message is an element of the *behavior set.* Otherwise the behavior will look for another message in the message queue of the actor or will wait for the arrival of an acceptable message. The *behavior set* is established by the *next behavior set function* at the time a replacement behavior is specified (and also when the *initial* behavior is created). The *next behavior set function* in turn depends on the *object-state functions* which consult the private data members of the behavior to determine the current state of the actor. The relationships between these elements is shown in **Figure 5**. The conceptual "state" of an actor is represented by the state variables which are consulted by the object-state functions. The next behavior set function invokes the object-state functions in order to compute the behavior set of the current behavior.

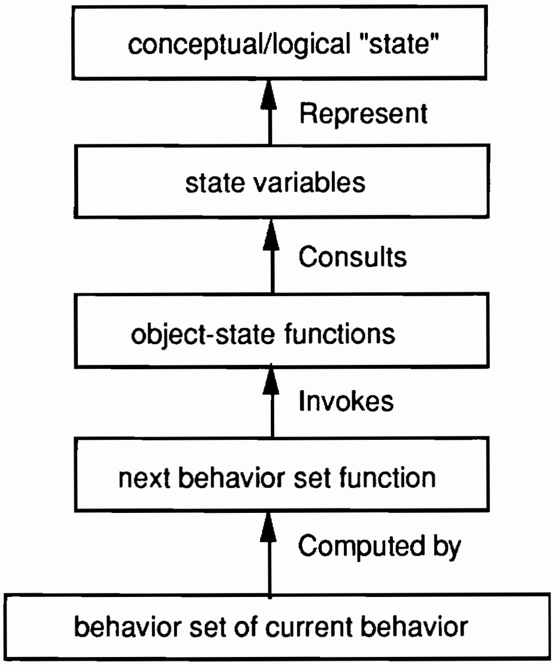
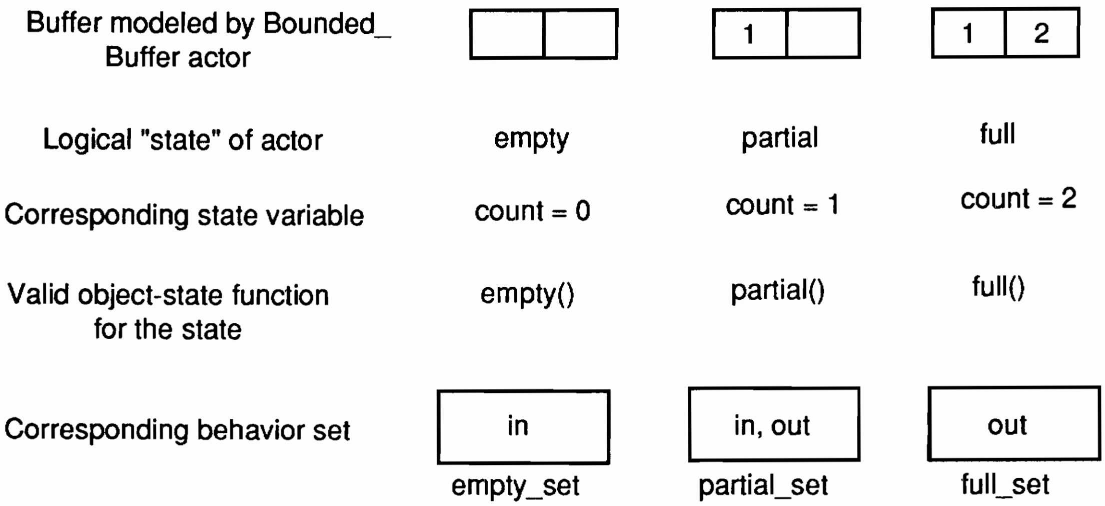

conceptual/logical "state"
Represent
state variables
Consults
object-state functions
Invokes
next behavior set function
Computed by
behavior set of current behavior

Figure 5: Data and function dependencies involved in the determination of a behavior set.

To illustrate the above ideas let us consider the *Bounded_Buffer* behavior once again. As shown in **Figure 6**, there are three possible states of the *Bounded_Buffer* - *empty*, *partial* and *full*. Corresponding to these three states we will define three *object-state functions* -

empty(), *partial()*, and *full()*. These three functions will have access to the private data member of the *Bounded_Buffer* class, *count*, which is used to keep a count of the number of elements in the buffer. There will be three *behavior sets* : *empty_set* which will have the identifier for the *in* method, *partial_set* which will have identifiers for both the *in* and *out* methods, and *full_set* which will have the identifier for the *out* method only. We will also define a *next behavior set function* which will use *object-state functions* to determine which one of these *behavior sets* to establish as the *behavior set* of the current behavior.



Buffer modeled by Bounded_ Buffer actor		1	1 2
Logical "state" of actor	empty	partial	full
Corresponding state variable	count = 0	count = 1	count = 2
Valid object-state function for the state	empty()	partial()	full()
Corresponding behavior set	in	in, out	out
	empty_set	partial_set	full_set

Figure 6: For a bounded-buffer of size two, the different configurations of the buffer, the state of the Bounded_Buffer behavior, the valid object-state function in each state, and the corresponding behavior set are shown.

The advantage of using *behavior sets* is that the behavior object corresponding to an empty buffer will not have the *out* method in its *behavior set* and as a result will never execute a message requesting an item extraction when the buffer is empty. Similarly, the behavior object corresponding to a full buffer will not have the *in* method in its *behavior set* and as a result will never execute a message requesting an item insertion when the buffer is full. Therefore, the state dependent synchronizations need not be encoded in the methods of a behavior thereby allowing reuse of superclass methods through the inheritance mechanism of C++.

1.4 Cboxes

Another notion required to program with actors in ACT++ is that of a **Cbox**. We have seen before that an actor can send request messages to other actors requesting some specific actions to be taken on its behalf. If the requesting actor requires some information returned as a result of its request, called a *reply message* in ACT++, then we must specify how the returned information is delivered to the requestor. In Agha's model the concept of **insensitive actors** is introduced for this purpose. An actor that awaits a reply from any of its acquaintances becomes an insensitive actor - an actor that does not process any of the pending messages in its queue until the reply message arrives.

Conceptually, insensitive actors is an elegant way of specifying the above interaction, but they lead to a very costly implementation. The insensitive actor creates and forwards to a buffer actor all subsequent request messages. When the reply message arrives the insensitive actor processes it and requests the buffer actor to send to it all accumulated request messages. The overhead stems from the fact that one entity, the message queue of the actor, is used both for receiving request messages and also for receiving a reply message.

To distinguish a reply message from all the request messages, ACT++ separates the handling of request messages and reply messages. The repository of reply messages is called a *Cbox* and, unlike the unique message queue of an actor object, there can be many

Cboxes declared in a behavior. While several types of Cboxes are possible, three types of *Cboxes* have been implemented in ACT++ as shown in **Figure 7**. *Cboxes* that store only the first reply message sent to it are denoted by FIRST. *Cboxes* that store only the most recent reply message sent to it (overwriting any previous one) are denoted by LAST. *Cboxes* that queue all the reply

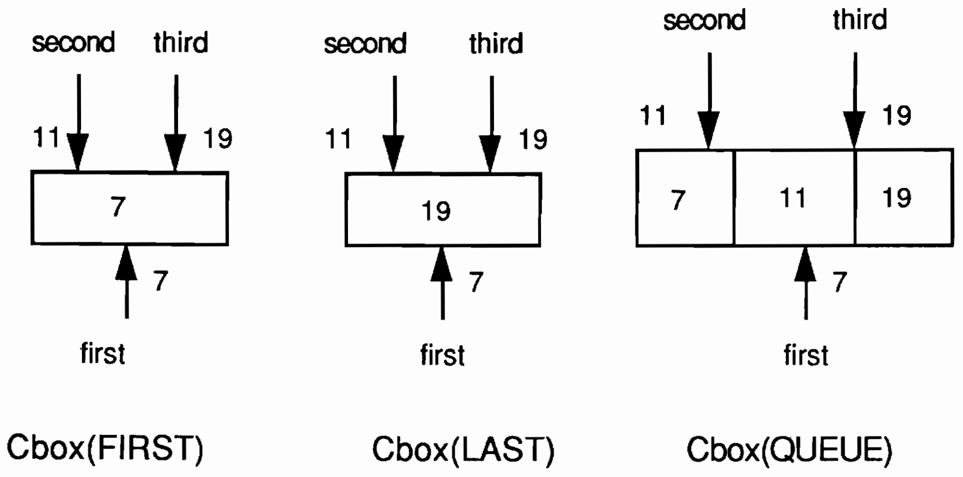


Figure 7: The different semantics associated with the three types of Cboxes. Three reply messages, first, second, and third that carry the integer messages 7, 11, and 19 respectively, arrive at the Cboxes of the different types. Cbox FIRST stores only the first item received, Cbox LAST stores only the last item received, and Cbox QUEUE queues all the item in the order of arrival.

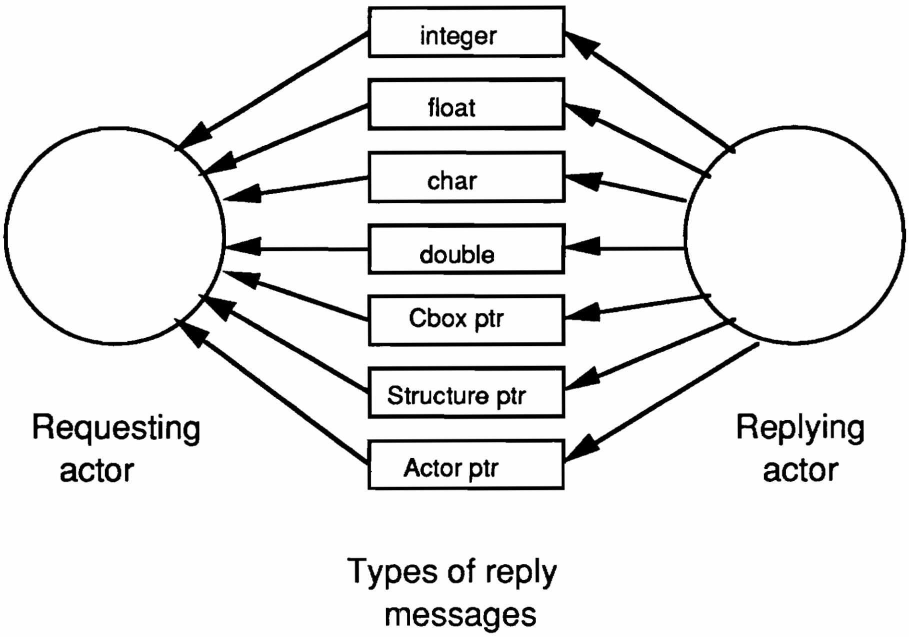


Figure 8: The seven types of reply messages that a replying actor can send to a requesting actor.

messages sent to it are denoted by QUEUE. *Cbox* objects are instantiations of a predefined class called **Cbox** and FIRST, LAST, and QUEUE are predefined macros in ACT++.

A reply message can carry one of seven types of information (**Figure 8**) - an integer, a floating point number, a character, a double, a Cbox pointer, a pointer to a structure, and an actor pointer. Every Cbox has the capability to store an item of each type.

1.5 Input and Output - Interface Actors, Rboxes, and Wboxes

This section considers three problems related to concurrent input and output (I/O for short) and the I/O structures introduced in ACT++ to cope with these problems. These problems are:

- the interference between concurrent sequences of I/O operations directed at the same file,

- the complexity of avoiding the blocking effects of low-level I/O system calls, and

- the consistency of the interpretation of I/O commands executed in different process contexts.

The first of these problems is inherent to concurrent computation and simply another instance of the general problem of interference among concurrent activities over their access to shared resources. The latter two problems are specific to our choice of implementing ACT++ on UNIX. These last two problems are discussed in detail because they will occur in any attempt to perform concurrent computation involving I/O on a UNIX system.

1.5.1 The Problem of Interference in Concurrent I/O

The inherent problems of concurrent I/O are due to the interference that arises if multiple concurrently executing processes perform I/O to or from the same destination. UNIX ensures I/O calls are non-preemptive, that is, if I/O is initiated by a process on a file descriptor, the corresponding file table entry will remain locked until the I/O is complete. But in between system calls, there is no such locking available. As a result, different interleaved executions of the I/O calls might lead to different results in different executions of the same program. In the case of writing to a terminal, multiple concurrent writes might cause overlapped and incomplete display of information depending on the relative speeds and the volume of I/O performed by the processes.

The solution suggested to the problem of interference due to concurrent I/O is to do I/O from a *critical section* or use an *I/O server* to do all I/O. Neither solution reduces the burden on the user. In the case of using critical sections, the user has to explicitly manage the locks used to implement each critical section. In the case of a server, the user must both define the server and ensure that the server does not become a performance bottleneck. Given the choice between doing I/O using critical sections and using an I/O server, an I/O server is more attractive because a server encapsulates all low level operations and provides high level abstractions for the user.

A useful language feature would allow the user to create I/O servers for particular files and then control the sharing of the server between the different active agents running the application. The creation of one server for each file reduces the possibility of the server becoming a bottleneck and also reduces the complexity of an individual server since it has to manage I/O to only a single file. As an example use of this feature consider a message passing environment and the two processes A and B where process A does two consecutive reads and process B does two consecutive writes to the same file. The different combinations of the *read* and *write* operations could be avoided, if through a *global synchronizer*, the processes could send the I/O requests in such a way that they were queued one after the other in the server and got processed in order. The global synchronizer enables co-ordinated message passing simulating a distributed simultaneous-P operation and also provides control over the sharing of the server.

1.5.2 The Problem of Blocking I/O Calls

The current version of ACT++ is implemented on the UNIX operating system. As such, the basic mechanisms for doing I/O are those available in UNIX. Among the I/O features available in UNIX are a set of system calls like *open*, *close*, *read*, and *write*, which perform I/O using a unique identifier called the *file descriptor,* which the operating system returns when an *open* system call is executed. File descriptors are unique identifiers which act as an internal representation of the *standard files* or the *special device files* they are associated with. There are a few other system calls like *fcntl* and *ioctl* that can be used to modify the status of the file descriptors. Only unformatted raw byte I/O is possible through the *read* and *write* system calls - no low level facility for formatted I/O is available.

The *read* and *write* system calls are blocking calls. This means that if the I/O is not possible immediately when the call is made, the process making the I/O call will block. When the I/O is possible, the operating system will unblock the process allowing it to complete the I/O. As a result of the blocking nature of the I/O calls, applications doing real-time monitoring of the external world might miss important external events if all the processes running the application block on I/O. Blocking I/O is also a problem in multi-threaded applications (like ACT++) as described below.

UNIX also provides non-blocking **asynchronous I/O** facilities for *terminals* and *sockets*. Only asynchronous terminal I/O will be discussed here. To perform asynchronous I/O the user must

- write a *signal handler* for the SIGIO signal that the operating system will deliver to the process when the I/O is ready,

- set up the file descriptor for asynchronous I/O by using a special option of the *fcntl* system call, and

- identify the process / process group to which the SIGIO signal will be delivered.

All this work has to be done by the user. A language construct that hides all of the above details for doing asynchronous I/O will certainly be easier to use.

1.5.3 The Problem of Consistency of File Descriptors

Another problem in concurrent I/O is the consistency of file descriptors across processes. A file descriptor is actually an index to a process specific table called the *file descriptor table*. As a result, a file descriptor is meaningful only in the context of the single process in which it was created. The current version of ACT++ is built on the PRESTO *threads* package in which a *thread* can execute in the context of different processes during its lifetime, if more than one process is being used to run the application. As a result, an *open* system call executed by a *thread* while running on process A will be rendered meaningless when the *thread* executes on process B. Therefore, the run time system must ensure that a *thread* that executes an *open* call on a particular process is scheduled to run on the same process throughout its lifetime.

1.5.4 Concurrent I/O in ACT++

ACT++, being implemented in C++, has available the C++ stream I/O facilities. The predefined stream I/O classes provide formatted terminal and file I/O. The inherent problem associated with concurrent I/O still exists in C++ and there is no facility to insure file descriptor consistency. Moreover, there is no high level abstraction available for doing asynchronous I/O. Given the shortcomings of the I/O facilities in C++ and recognizing the need for performing asynchronous I/O, providing high level abstractions to address the inherent problem of concurrent I/O, and the maintenance of file descriptor consistency, we have incorporated special features in ACT++ which address all of these issues.

The primary I/O abstraction introduced in ACT++ is that of an **interface actor** (IA). An IA is an I/O server which manages I/O to a single I/O device - a standard file or a terminal special file. The IA encapsulates all low level details for performing I/O and relieves the user from managing file descriptors explicitly. The underlying scheduling mechanism in PRESTO has been modified to address the problem of file descriptor consistency due to *thread* migration. IAs are capable of doing both synchronous and asynchronous I/O. If a terminal I/O request cannot be satisfied immediately then the IA makes all the arrangements for performing asynchronous I/O. The IA transparently prepares the file descriptor for asynchronous I/O and interacts with the relevant signal handler when the I/O is ready. The problem of arbitrary interleavings of concurrent I/O requests can be solved by using **global synchronizer actors** which model simultaneous-P operations.

The introduction of IAs has enabled the modeling of I/O in terms of actor operations thereby presenting a uniform environment to the users of ACT++. Just as every actor is associated with a behavior at any point in time, an IA is associated with a behavior that is an instantiation of a predefined class called **Tty_Beh**. The association of this behavior with an IA is done transparently by the ACT++ run time system. The *Tty_Beh* behavior objects manage the low level aspects of performing I/O and provide an interface consisting of three methods - *Read*, *Write*, and *Close*. The user is responsible for creating an IA object and sending request messages to it whenever I/O needs to be performed.

The media through which information pertaining to an I/O operation is transmitted are two special kinds of Cboxes called a **Rbox** (read box) and a **Wbox** (write box). As the names signify, a Rbox acts as the buffer for *Read* operations and a Wbox plays the same role for a *Write* operation. The user can create any number of Rboxes and Wboxes. When a *Read* operation is requested of an IA, a Rbox is created and is passed as an argument to the *Read* operation. Just before the information being read is necessary in the program, a blocking operation is performed on the Rbox to determine whether the I/O operation has completed. If the Rbox is still empty the behavior automatically blocks on the Rbox. Whenever the I/O completes the system automatically awakens the blocked behavior.

For a *Write* operation the user creates a Wbox and passes it as an argument to the *Write* operation. When a Wbox is created the user specifies the data that is to be written and a copy of that data is made inside the Wbox. Then the user has the option of either waiting on the Wbox to be emptied or doing other work. If no indication is necessary as to when the *Write* operation completes, the user can fill up a Wbox and send it along with the *Write* request without ever doing a subsequent blocking operation on it.

The *Close* operation defined in the *Tty_Beh* class is used to close the file descriptor that the IA uses to perform I/O. The closing of the file descriptor renders the IA useless and hence it is marked as "dead" thereby losing the capability of processing any subsequent I/O requests.

To see an example use of the special I/O features of ACT++ let us consider the example in **Figure 9**. An arbitrary actor wants to read from a terminal A and write the read information to terminal B. To do so two interface actors are needed, each responsible for I/O to one terminal device. It is assumed that the IAs exist when the current behavior of the actor starts executing. The current behavior first creates a Rbox. Then a *Read* request message containing the identity of the Rbox is sent to the IA for terminal A (action 1 in the figure). Then the behavior blocks on the Rbox (action 2). When the IA for terminal A processes the *Read* request message from the behavior, the *Read* method of the current *Tty_Beh* object is invoked. The method interacts with the SIGIO signal handler and reads the information from the terminal asynchronously (action 3) and puts the data in the Rbox (action 4). As soon as the Rbox receives the read data, the behavior is awakened and the data is read (action 5).

To write the information just read to terminal B, the behavior creates a Wbox using the data in the Rbox (action a). Then the behavior sends a *Write* message to the IA for terminal B (action b). Then the behavior blocks on the Wbox (action c). When the IA for terminal B processes the request message from the behavior, the *Write* method of the current *Tty_Beh* object is invoked. The method first obtains the data from the Wbox (action d) and then does a synchronous write to the terminal (since the terminal was ready to be written) (action e). At the completion of the *Write* operation the blocked behavior is automatically awakened.

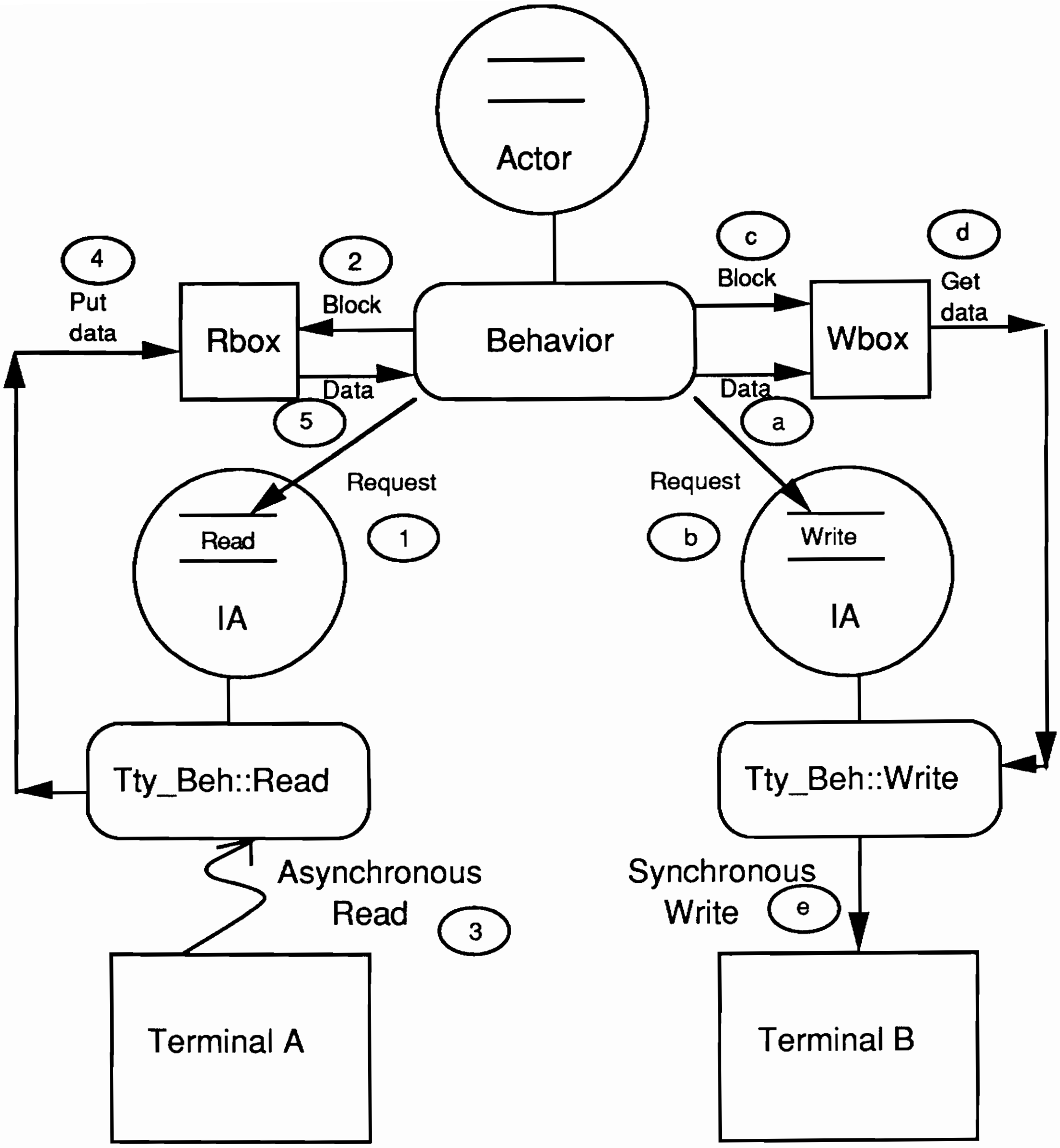


Figure 9: Showing the interaction between a general actor, interface actors (IA), and Rboxes and Wboxes. The numbers 1 through 5 and the letters a through e denote the sequence of operations in the case of a read and a write respectively.

2. Overview of the ACT++ Classes

The following sections provide a detailed discussion of the components of ACT++ that are necessary to construct programs. The specific components that will be discussed are:

- Actors
- Messages
- Behaviors
- Behavior Sets
- Cboxes
  - FIRST
  - LAST
  - QUEUE
- Asynchronous Input/Output
  - Interface Actors
  - Rboxes and Wboxes

For each component the C++ class interface will be introduced, the operations of the class that can be used by programmers will be explained followed by examples showing the intended use of the component in ACT++ programs.

2.1 Actors

All actor objects in ACT++ are instantiations of the predefined `Actor` class. The methods of this class that can be used in an ACT++ program are the constructor for the class and the print method. The constructor has the following signature:

```
Actor(Behavior* init_beh, int th_num = 1, char* name = 0).
```

The behavior pointer `init_beh` specifies the initial behavior object that is responsible for processing the first message to the actor. The integer variable `th_num` signifies the number of simultaneously active *threads* that can be associated with the actor. The default number of *threads* is one. The user can specify more that one *thread* to enable the possibility of utilizing intra-actor concurrency by allowing two or more behavior objects to process request messages simultaneously. The discussion on placement of the *become* operation and state variable interference in section one is referred to for a discussion of the subject of concurrent actors. The character pointer `name` associates a name with an actor that can be printed when debugging an ACT++ program.

Every object in ACT++ has a `print` method defined in it that can be used to print the values of the private data members of the class. This method is useful only during debugging an ACT++ program. This method uses the stream I/O facility of C++. The `print` method has the following signature:

```
virtual void print(ostream& = cout).
```

Henceforth, this method will not be discussed for any other class.

A few example uses of the above operations are shown below. Since the second and third arguments of the actor constructor have default values the user need not provide actual arguments corresponding to them when creating an actor object. In the following, we show three different ways of creating an actor. The variables `user_beh1, user_beh2,` and `user_beh3` refer to instantiations of user defined behavior classes.

```
Actor* my_act1 = new Actor(user_beh1, 2, "ROOT_ACTOR");
      //Actor allowing two threads simultaneously
Actor* my_act2 = new Actor(user_beh2, 3);
      //Actor with no name and 3 simultaneous threads
Actor* my_act3 = new Actor(user_beh3);
      //Actor with no name and only one thread
my_act1->print();
```

2.2 Messages

All message objects are instantiations of the predefined class `Message`. There are two constructors in the `Message` class. Their signatures are:

```
Message(PFany, ...);
Message(Actor*, PFany, ...);
```

A message object consists of the physical address of the method that is being requested for execution along with all its actual arguments. The first argument of the first constructor is of type `PFany` (pointer to a function) and it is obtained in the same way as the elements of a behavior set were obtained. The ellipsis after `PFany` signifies that an unknown number of arguments can follow the name of the method. The same discussion applies to the second constructor except that its first argument is a pointer to an actor object. If constructed in the second way, the pointer to the actor acts as a possible destination of the message - the actual sending does not take place when the message is constructed and at the time of sending, the message need not be sent to the actor recorded in it.

Messages are sent to actors using the send method of the Message class. There are two overloadings of the send method. They are:

```
void send();
void send(Actor*);
```

The first overloading is used to send a message that is constructed using the second Message constructor above - the actor recorded in the message becomes the destination of the message. The benefit of this form of sending a message is that the sender need not be aware of the destination of the message. This implies that the sender need not construct all the messages it sends - it can receive a message object as an argument from another actor and send it whenever a certain condition arises without actually knowing the exact destination of the message.

The second overloading is used to send a message constructed using either of the Message constructors above. For a message object constructed using the first constructor, the actor pointer in the argument of send specifies the destination of the message. For a message object constructed using the second constructor, the existing actor pointer in the message object is ignored and the message is sent to the actor specified in the argument.

An important issue related to message passing in ACT++ is the reuse of message objects. A message is created in a behavior, sent to an actor, processed by a behavior, and deleted at the end of the processing. Therefore, if a single message object is sent to two or more actors, then, whoever finishes processing the message first, will delete the object. This would delete the message from the queues of all the actors that have not processed the message yet and will disable access to data members of the message from those behaviors that were in the middle of processing the message. Therefore the same message object *must never* be sent to more than one actor.

Another important issue related to message passing in ACT++ is that of type checking of the arguments. There is **no type checking** of the arguments of the methods that are invoked through a message. This implies that the user bears full responsibility for ensuring type consistency between a message and the signature of the method that will get executed as a result of its delivery. The lack of type checking stems from the fact that message passing simulates asynchronous method invocation instead of the standard synchronous way of invocation in C++. When a method is invoked in C++ the compiler does the type checking of the arguments, sets up the callee's activation record, and generates code to start executing the callee. What would have been ideal for ACT++ is if the compiler did just the type checking and setting up the callee's activation record and provided the ability to indicate the start of execution of the callee to the user. The lack of this feature has not only affected ACT++, PRESTO suffers from this problem too. All type information is lost when a *thread* is started asynchronously inside the method of an object. Since the asynchronous message passing in ACT++ is built on the asynchronous *thread* execution feature of PRESTO, parameter passing restrictions present in PRESTO also apply to ACT++. The following is an excerpt from "The PRESTO User's Manual" [Bershad 90] (page 8) indicating the restrictions involved.

> The asynchronous nature of start() affects the styles of parameter passing that may be used in a started function. Default and reference parameters will not work properly although virtual and inline functions work fine. Overloaded functions will also not work since the compiler cannot discern the types of the arguments to the function (or its return value) at compile time.
> Disallowing reference parameters implies that all parameters (including pointers) are passed by value. Consequently the programmer should be careful that pointers reference objects

```
residing in the global address space (static or created by
new) and do not point to objects on the stack of the thread
that is starting the new thread. Failure to abide by this
may result in the passed pointer referencing garbage
(consider the stack's behavior on an asynchronous
invocation).
```

In the case of ACT++ the above discussion applies to the methods that are executed as a result of message passing. The user has to be careful not to invoke any method through message passing that matches with any of the above descriptions.

2.3 Behaviors

All behavior objects in ACT++ are instantiations of user defined subclasses of the predefined `Behavior` class. The constructor in the `Behavior` base class has the following signature:

```
Behavior().
```

User defined behavior classes can have their own constructors. A behavior object is created before the corresponding actor object because the actor object needs to know the object which defines the actor's initial behavior. For example, the following sequence creates and binds the `Bounded_Buffer` actor to its initial behavior:

```
class Bounded_Buffer : public Behavior    {
        ...
public:     Bounded_Buffer(int size);
        ...
};

Bounded_Buffer* b = new Bounded_Buffer(100);
Actor* BB_act = new Actor(b);
```

The initial behavior, and each subsequent behavior in turn, has the responsibility of specifying its replacement. A replacement behavior is specified by the `become` operation defined in the `Behavior` class. The signature is:

```
void become(Behavior* b).
```

The argument to the `become` operation specifies a pointer to a behavior object of the same class as the current behavior. This replacement behavior could be an entirely new object as shown in the following example that specifies a replacement for the `Bounded_Buffer` behavior:

```
become(new Bounded_Buffer(100)).
```

It is also possible to specify the current behavior object as the replacement behavior by using

```
become(THISBEH)
```

where THISBEH is a predefined macro.

To specify a null replacement behavior corresponding to a "dead" actor one must use:

```
become(DEAD_ACTOR)
```

where DEAD_ACTOR is a predefined macro.

The last operation defined in the `Behavior` class is the `SELF` operation where `SELF` is a predefined macro. This is used to access the actor object associated with a behavior object -

the SELF macro returns a pointer to the actor object. This operation is useful when an actor wants to send a mail message to itself which can be done from a method in the following way:

```
self_mess->send(SELF)
```

where self_mess is a pointer to a message object.

2.4 Behavior Sets

Behavior set objects are instantiations of the predefined class Behavior_Set. The elements in a behavior set are addresses of the methods of a behavior class. The address is obtained using the following expression:

```
(PFany)&Class_name::Method_Name.
```

PFany is a predefined type name in PRESTO that denotes a pointer to a member function; Class_name is the name of a behavior class; Method_Name is the name of a method in the Class_name behavior class.

There are two constructors in the Behavior_Set class whose signatures are:

```
Behavior_Set();
Behavior_Set(PFany, ...).
```

The first constructor is a default constructor that is used to construct an empty behavior set object. The second constructor is used to construct a behavior set whose elements are addresses of the methods supplied as arguments to the constructor. This constructor can take an arbitrary number of arguments (indicated by the ellipsis) all of which must be of type PFany. Considering the Bounded_Buffer example once again and assuming that there are two methods, *in* and *out*, defined in the class, the following would construct three behavior sets corresponding to the three states of the Bounded_Buffer behavior - empty, partial and full.

```
Behavior_Set* empty_set = new Behavior_Set((PFany)&Bounded_Buffer::in);

Behavior_Set* partial_set = new Behavior_Set((PFany)&Bounded_Buffer::in,
                              (PFany)&Bounded_Buffer::out);

Behavior_Set* full_set = new Behavior_Set((PFany)&Bounded_Buffer::out);
```

There are three overloaded operators defined which must be used in association with operations on behavior sets. These operators are defined as friend functions of the Behavior_Set class. There are two overloadings of the '=' operator and one overloading of the '+' operator. The signatures of these overloadings are:

```
friend void operator =(Behavior_Set&, Behavior_Set*);
friend BehSetq* operator +(Behavior_Set&, Behavior_Set&);
friend void operator =(Behavior_Set&, BehSetq*);
```

The first overloading for '=' is to be used as follows. Suppose one has declared a null behavior set (i.e., a behavior set that has no method addresses in it) and later on wants to initialize it with actual method addresses. Then this overloading can be used as follows:

```
Behavior_Set empty_set;
       ...
empty_set = new Behavior_Set((PFany)&Bounded_Buffer::in).
```

The left hand side corresponds to the first argument and the right hand side (which is actually the pointer to the new behavior set object constructed by the *new* operator) corresponds to the second argument of the operator function. As a result of this assignment the behavior set queue (the method addresses are actually stored in a queue inside the behavior set object which is called the behavior set queue) of empty_set is deleted, the behavior set queue of the new object is assigned to empty_set, the behavior set queue pointer of the new behavior set is set to zero, and finally the new behavior set object is deleted.

The overloading for '+' is used to form the union of two behavior sets non-destructively. Note that the arguments of '+' are references to behavior set objects, not pointers to behavior sets. Therefore to use '+' one has to add two behavior set objects that have been declared (e.g. as done by Behavior_Set empty_set above) instead of creating them by using the new operator. The '+' operator extracts each item from the behavior set queues of its two arguments non-destructively and creates an entirely new behavior set queue that it returns as its result. The '+' operator is always used with the second overloading of the '=' operator. The new behavior set queue returned by '+' matches the second argument of '='. The '=' operator deletes the existing behavior set queue of the behavior set that appears as its first argument and reassigns it to point to the new behavior set queue. The following shows an application of these two operators used to define the partial_set behavior set for the Bounded_Buffer object.

```
Behavior_Set empty_set;
Behavior_Set full_set;
      ...
empty_set = new Behavior_Set((PFany)&Bounded_Buffer::in);
full_set = new Behavior_Set((PFany)&Bounded_Buffer::out);
      ...
partial_set = empty_set + full_set;
```

Behavior sets are used by behavior objects to project different interfaces to the outside world depending on the internal state of the behavior object. As a result, behavior sets are always used as an integral part of behavior objects. To incorporate the addition of behavior sets to behavior objects two protected items have been introduced in the `Behavior` class. They are as follows:

```
protected:
            Behavior_Set  curr_bset;
            virtual Behavior_Set NextBehavior_Set() { };
```

In the constructor of the `Behavior` class `curr_bset` is set to zero and, as shown above, the `NextBehavior_Set` method is empty. The `curr_bset` member is used to store the current behavior set of the behavior object. The `NextBehavior_Set` function is used to determine the behavior set of the initial behavior object and those for all subsequent replacement behaviors. It is the responsibility of the user to provide a definition of the `NextBehavior_Set` method in the behavior subclasses and ensure that the `curr_bset` member is maintained properly. Note that the determination of the next behavior set is based on the values of the state variables of the behavior object. The determination can be done by explicitly accessing the state variables of the object. A cleaner and more logical way is to define boolean functions each of which will be responsible for determining one particular state of the object. This is illustrated at the end of this section by the `empty()` and `full()` functions.

After defining the state computing functions and the `NextBehavior_Set` function the user has to do the following assignment to ensure that the behavior set is properly set.

```
curr_bset = NextBehavior_Set()
```

The above assignment is available in a shorter form in ACT++ as the predefined macro `CURRENT` which expands to the above assignment.

The placement of CURRENT has to be done carefully for behavior sets to work properly. The constructor used to construct a behavior object for the first time must have a CURRENT statement after all the state variables have been initialized. Subsequently, if new behavior objects are created as replacement behaviors then the constructor used to create a new object must have a CURRENT specification in it. Otherwise, if the same behavior object is reused as the replacement behavior then a CURRENT statement has to be executed before the *become* operation is specified in the method- this has to be done from all the methods of the behavior from which a *become* of this type is executed.

We give a complete class definition of the Bounded_Buffer behavior below that uses behavior sets and exemplifies all the aspects discussed above.

```
class Bounded_Buffer : public Behavior    {

      int data[2];              //A buffer of only two integers
      int num_items;            //Keeps track of the number of items

protected:

      Behavior_Set empty_set;
      Behavior_Set partial_set;
      Behavior_Set full_set;
      virtual int empty();//The empty state computing func
      virtual int full();//The full state computing func
      virtual Behavior_Set NextBehavior_Set();
                  //Redefining the next behavior set method

public:

      Bounded_Buffer(); //Constructs the initial behavior
      Bounded_Buffer(int*, int);
            //Constructs the replacement behaviors
      void in(int); //Insert an item into the buffer
      int out(); //Extract an item from the buffer
}

Bounded_Buffer::Bounded_Buffer()
{
      num_items = 0;
      data[0] = -100; //-100 signifies empty
      data[1] = -100;
      empty_set =
            new Behavior_Set((PFany)&Bounded_Buffer::in);
      full_set =
            new Behavior_Set((PFany)&Bounded_Buffer::out);
      partial_set = empty_set + full_set;
      CURRENT;    //in order to set the behavior set
}

Bounded_Buffer::Bounded_Buffer(int* i, int num)
{
      num_items = num;
      data[0] = *i++;
      data[1] = *i;
      empty_set =
            new Behavior_Set((PFany)&Bounded_Buffer::in);
      full_set =
            new Behavior_Set((PFany)&Bounded_Buffer::out);
      partial_set = empty_set + full_set;
      CURRENT;    //in order to set the behavior set
}

void Bounded_Buffer::in(int i)
{
      if (num_items == 0)
            data[0] = i;
      else
            data[1] = i;
      num_items++;
      become(new Bounded_Buffer(&data, num_items));
                  //Create new replacement behavior object
                  //CURRENT is invoked from the constructor
}

int Bounded_Buffer::out()
{
      int i;
      if (num_items == 1)     {
            i = data[0];
            data[0] = -100;
      }
      else  {
            i = data[1];
            data[1] = -100;
      }
      num_items--;
      become(new Bounded_Buffer(&data, num_items));
                  //Create new replacement behavior object
                  //CURRENT is invoked from the constructor
      return i;
}

int Bounded_Buffer::empty()
{
	if (num_items == 0)
		return 1;
	else
		return 0;
}

int Bounded_Buffer::full()
{
	if (num_items == 2)
		return 1;
	else
		return 0;
}

Behavior_Set Bounded_Buffer::NextBehavior_Set()
{
	if (empty())
		return empty_set;
	else if (full())
		return full_set;
	else
		return partial_set;
}
```

2.5 Cboxes

All Cbox objects in ACT++ are instantiations of the predefined class `Cbox`. The signature of the constructor is

```
Cbox(Cbox_Type).
```

The argument to the constructor specifies one of the three types of Cboxes available in ACT++. The three types differ in the way they handle the reply message they receive. The three types are denoted by three macros respectively, namely, `FIRST`, `LAST`, and `QUEUE`. The Cbox of type `FIRST` stores only the very first reply message sent to it and ignores any subsequent reply messages. The Cbox of type `LAST` stores only the very last reply message and overwrites any existing reply message when it receives a new one. The Cbox of type QUEUE enqueues all the reply messages in an internal queue. The following shows how to create Cboxes of the three types.

```
Cbox* cbox_f = new Cbox(FIRST);
Cbox* cbox_l = new Cbox(LAST);
Cbox* cbox_q = new Cbox(QUEUE);
```

The receiver of the reply message receives the reply using the different overloadings of the receive method defined in the Cbox class. Each overloading is used to receive a single type of reply message. The only argument to the receive methods specifies the name of a **reference** variable whose type matches the type of message expected. The signatures are as follows:

```
void receive(int&);
void receive(float&);
void receive(char&);
void receive(double&);
void receive(void*&);
void receive(Cbox*&);
void receive(Actor*&).
```

The first four overloadings of receive are used to receive reply messages of the four basic types, namely, integer, floating point, character, and double. The fifth overloading with a void*& argument is used to receive a structure pointer. Since the name of the structure is user-defined, a void*& has been used to refer to a pointer to any structure. The last two overloadings are used to receive a Cbox pointer and an actor pointer respectively.

Note that receiving a reply message from a Cbox is a destructive process - the reply message is extracted from the Cbox and the Cbox is marked empty.

The receive operation is a blocking operation. If the reply message is available in the Cbox when the operation is invoked the call returns immediately. If the item has not yet been received by the Cbox then the current *thread* of execution in the behavior object blocks. The *thread* unblocks automatically when the reply message arrives in the Cbox.

The sender of the reply message has to use one of the following six overloadings of the send method defined in the Cbox class to send all types of reply messages except a structure pointer. To send a pointer to a structure, the SENDSTR macro has to be used.

```
void send(int&);
void send(float&);
void send(char&);
void send(double&);
void send(Cbox*);
void send(Actor*);
SENDSTR(x, y);    //x is a pointer to a structure and y is the name of
                  //the structure type of which x is an instance
```

Note that using the six overloadings of send the user can send reply messages that contain only a single data item of a particular type. To send more than one item of the same or different types one must declare a structure containing all the elements and send back a pointer to the structure using the SENDSTR macro. The structure being sent back must not be a local item - it must reside in the global address space (i.e. either static or created by new). Note that the behavior receiving the structure-pointer reply message has the responsibility of extracting the data items in the structure. Moreover, both the sender and receiver must use the same structure definition to exchange information. Otherwise the extracted items may not match in type.

A point to be noted about QUEUE type Cboxes is that they can be used to receive multiple items either from the same or different replying actors. The important point to note about using QUEUE Cboxes is that a particular Cbox must be used to store *multiple reply messages of only one type*. This is because when the receiving behavior does a receive on the Cbox, the item at the head of the queue will be extracted and assigned to the local variable supplied as an argument. If the Cbox is not used to store homogeneous items then the above could result in an invalid assignment. Since the functionality of searching a QUEUE type Cbox for a reply message of the desired type is not implemented, the above restriction has to be adhered to.

Since receiving a reply message removes the message from the Cbox and marks it as empty, the FIRST and LAST type of Cboxes can be reused immediately after a receive operation on them completes. But that is not true for QUEUE type Cboxes which might have several reply messages waiting to be extracted after a receive operation completes. If it is ever desired to ignore all the reply messages accumulated in a QUEUE type Cbox before receiving the next reply message then the flush method must be used whose signature is as follows:

```
void flush()
```

The following is an example showing how a receiver and a sender of reply messages would use the different types of Cboxes to exchange information.

```
In the receiving behavior:

int ret_int1;
int ret_int2;
float ret_float;
char ret_char;
double ret_double;
Cbox* ret_cb;
Actor* ret_act;
struct xyz { int a; float b; };
xyz* ret_struct = new xyz;
```

```
Cbox* cb1 = new Cbox(QUEUE);
Cbox* cb2 = new Cbox(LAST);
Cbox* cb3 = new Cbox(FIRST);
Cbox* cb4 = new Cbox(FIRST);
Cbox* cb5 = new Cbox(FIRST);
Cbox* cb6 = new Cbox(FIRST);
Cbox* cb7 = new Cbox(FIRST);
      ...
cb1->receive(ret_int1);
cb1->receive(ret_int2);
cb2->receive(ret_float);      //Depending on when this is
            //executed, ret_float will be either 7.4 or 8.4
cb3->receive(ret_char);
cb4->receive(ret_double);
cb5->receive(ret_cb);
cb6->receive(ret_struct);
cb7->receive(ret_act);

In the sending behavior:

struct pqr { int a; float b;};
pqr* abc = new pqr;
abc->a = 123;
abc->b = 67.3;
Cbox* cb_ret = new Cbox(LAST);
c1->send(4);
c1->send(5);
c2->send(7.4);
c2->send(8.4);
c3->send('a');
c4->send(123.456e-12);
c5->send(cb_ret);
c6->SENDSTR(abc, pqr);
Bounded_Buffer* beh = new Bounded_Buffer(100);
Actor* act = new Actor(beh);
c7->send(act);
```

2.6 I/O in ACT++

To perform I/O in ACT++ one has to use the three following items in combination:

- Interface actors (IAs),
- Rboxes, and
- Wboxes.

An IA is a special type of actor that is responsible for doing I/O using a unique file descriptor. Thus there is a one-to-one correspondence between IAs and file descriptors. Currently only null-terminated string I/O can be done using IAs. When the user wants to do I/O on a particular file (either a special file corresponding to a terminal or an ordinary character file) the exact name of that file has to be obtained first - for a terminal this can be done using the *tty* command in UNIX at the shell prompt. After that the user has to create an instance of the IA using the following constructor:

```
IActor(char* fname, Tty_Beh* init_beh, char* name = 0).
```

The first argument is the name of the file to and from which the interface actor will do I/O. The second argument specifies the initial behavior object that is to be associated with the interface actor. The third argument specifies an optional name for the IA.

The behavior that is responsible for processing request messages sent to an IA is predefined in ACT++. The initial behavior object of an IA is an instantiation of the predefined `Tty_Beh` class. The initial behavior object is created by specifying the `TTYBEH` macro in the place of the second argument of the `IActor` constructor. The following shows how to create an IA associated with the /dev/ttyp9 special file corresponding to a terminal.

```
char* fname = "/dev/ttyp9";
IActor* my_iact = new IACTOR(fname, TTYBEH);
```

An Rbox is a character buffer that is used by a behavior to read data. All Rboxes in ACT++ are instantiations of the predefined class `Rbox`. There are two constructors in the Rbox class:

```
Rbox(), and
Rbox(int).
```

The first one is the default constructor which creates an empty Rbox. The second constructor is used to create a Rbox containing a buffer of a specified size - the size is passed as the only argument to the constructor.

Rboxes can be reused. To do so, one has to delete any prior data placed in the Rbox. That can be achieved using the two overloadings of the `refresh` method defined in the `Rbox` class. Their signatures are as follows.

```
void refresh();
void refresh(int);
```

The first overloading is used to delete the existing buffer in the Rbox and create a new one having the same size as the previous buffer. The second overloading provides the ability to resize the buffer in the Rbox by specifying a new size. These two methods provide some flexibility in reusing existing Rboxes without having to create new ones.

After data has been read into a Rbox the `get` method has to be used to extract information characterwise from the Rbox. The signature of the method is

```
char get(int).
```

The integer argument specifies the position of the character in the buffer that one is interested in reading. This is similar to indexing into an array of characters. Note that the reading from a Rbox is a non-destructive process; the read character remains in the buffer of the Rbox.

Another useful method in the Rbox class is the `size` method that returns the current size of the buffer in the Rbox. The signature is

```
int size().
```

A Wbox is the medium through which null-terminated strings can be written. All Wbox objects in ACT++ are instantiations of the predefined class `Wbox`. There are three constructors in the `Wbox` class whose signatures are as follows.

```
Wbox();
Wbox(char*);
Wbox(Rbox*);
```

Unlike Rboxes which are used to receive information from the outside world, Wboxes are used to output information to the outside world. As a result, before a Wbox can be used in a write operation it has to be loaded with the information to be written. The first constructor is used to create an empty Wbox which can be filled later with the data to be written. The second constructor is used to create a Wbox that contains the information to be written - the argument specifies a pointer to the information that is copied into the Wbox.

The third constructor provides the ability to construct a Wbox using the contents of a Rbox. Suppose that one wants to write something read from one device to another device. Then there is no point in reading the information into local memory and then constructing a Wbox to output it. The straightforward way is to construct the Wbox directly from the Rbox. That is achieved through the third constructor which creates a Wbox of the same size as the Rbox being passed as the argument and makes a copy of the content of the Rbox in the new Wbox.

Wboxes can be reused just as Rboxes. The two overloadings of the `refill` method can be used to do so. Their signatures are as follows.

```
void refill(char*);
void refill(Rbox*);
```

These two serve the same purpose as that of the second and third `Wbox` constructors respectively. The only difference is that these operate on already existing Wboxes. The first overloading provides the ability to refill a Wbox with new information - the argument provides a pointer to the new information. It can also be used to fill an empty Wbox created using the first constructor. The second overloading provides the ability to refill a Wbox from a Rbox - the size and content are extracted from the Rbox.

To delete the contents of a Wbox and make it ready for reuse one can use the `refresh` method whose signature is

```
void refresh().
```

This deletes the existing buffer in the Wbox and marks it empty.

Another useful method in the Wbox class is the `size` method that returns the current size of the buffer in the Wbox. The signature is

```
int size().
```

The actual reading and writing of information using Rboxes and Wboxes is achieved through two methods of the predefined `Tty_Beh` class. These methods are `Read` and `Write` whose signatures are as follows.

```
void Read(Rbox* r, int nbytes);
void Write(Wbox* w, int nbytes);
```

When a behavior wants to read it creates a Rbox and sends a message to the IA for the file requesting the invocation of the `Read` method. The Rbox pointer is sent as the first argument and the number of bytes to be read is sent as the second argument. When a behavior wants to write it creates a Wbox and sends a message to the IA for the file requesting the invocation of the `Write` method. The Wbox pointer is sent as the first argument and the number of bytes to be written is sent as the second argument. When the read is complete, a Rbox is marked as full and any behavior waiting on the Rbox is awakened. Similarly, when the write is complete, a Wbox is marked as empty and any behavior waiting on the Wbox is awakened.

The `wait` method defined in the `Rbox` and `Wbox` classes are used to implement blocking on these boxes. The signature of the `wait` method is as follows.

```
void wait().
```

Before a `Read` opeartion is requested by a behavior it creates a Rbox object. Then when the read data is required in the program a `wait` operation is invoked on the appropriate Rbox. The operation blocks if the Rbox has not yet received the data. Otherwise the call returns immediately. The `wait` on Wboxes is used to determine whether the `Write` operation has completed. If the `Write` has completed the call returns immediately. Otherwise the operation blocks. Automatic awakening of the blocked behavior is ensured by `wait` for both types of boxes.

When a message for an IA is to be created, the TTYACT macro name must be used to obtain the address of the Read/Write method of the Tty_Beh class.

```
Message* m1 = new Message(TTYACT::Read, rb, 72);
Message* m2 = new Message(TTYACT::Write, wb, 72);
```

Since Tty_Beh is not a user defined class, TTYACT ensures that the proper name and syntax is always used when a message for an IA is created.

A note regarding the use of IAs. As stated before, an IA is an I/O server for a single file which serializes all I/O requests to the file. The serialization of requests ensures that only one I/O operation is active on any file descriptor at any time. This solves the problem of arbitrary interleaving of I/O operations when performing concurrent I/O. In order to ensure the prevention of arbitrary interleaving of I/O requests one has to further ensure that only one IA is ever active for a particular file. Otherwise, the prevention of the problem cannot be guaranteed.

The potential for creating multiple IAs for the same file exists when an IA is created from the constructor of a replacement behavior everytime a replacement is specified. Although a new behavior object is created to process the next message the existing IA must be used by it. To ensure this one must declare an IA pointer as a private data member of the user defined behavior. Then in the constructor that constructs the initial behavior, one has to create a new IA and store a pointer to it in the private member. When replacement behaviors are created later, one has two options depending on which *become* operation is used. If the same behavior object is reused then nothing special has to be done. If a new behavior object is created, then the constructor creating the replacement *must not* create a new IA for the same file. Instead a pointer to the existing IA must be passed to the constructor in which the private data member for storing the IA pointer must be set.

The final I/O related operation is the *Close* method defined in the *Tty_Beh* class which has the following signature:

```
void Close().
```

The *Close* method must be used to indicate that an IA will no longer process request messages. The effect of the operation is to close the file descriptor used by the IA to perform I/O and mark the IA as "dead". Any messages sent to the IA after *Close* has been executed will not be processed and will be deleted.

Now that we have discussed all the components required to perform I/O in ACT++ we give several examples below showing how to use them.

```
char* fname = "/dev/ttyp9";
IActor* my_iact = new IACTOR(fname, TTYBEH);
      ...
Rbox* rb_empty = new Rbox();  //Create an empty Rbox
Rbox* rb_n_empty = new Rbox(72);    //Create a non-empty Rbox
char* mess1 = "Please enter a line of text";
char* mess2 = "Have you finished using the program?"
Wbox* wb_empty = new Wbox();  //Create an empty Wbox
Wbox* wb_n_empty = new Wbox(mess1); //Create a non-empty Wbox
wb_empty->refill(mess1);      //Refill empty Wbox with mess1
wb_n_empty->refill(mess2);    //Refill non-empty Wbox with mess2
rb_empty->refresh(72);  //Refill empty Rbox with a buffer of size 72
Message* read_mess = new Message(TTYACT::Read, rb_empty,
                        rb_empty->size());  //Message for IA
```

```
read_mess->send(my_iact);     //Send Read message to IA
rb_empty->wait(); //Wait for read to be over
wb_empty->refill(rb_empty);    //Refill Wbox from Rbox
Wbox* wb_from_rb = new Wbox(rb_empty);    //Create Wbox from Rbox
char first_char = rb_empty->get(0); //Read first character
char thirty_first = rb_empty->get(30);    //Read 31st character
rb_empty->refresh();
Message* write_mess = new Message(TTYACT::Write, wb_from_rb,
                            wb_from_rb->size());    //Message for IA
write_mess->send(my_iact);
```

3. How to Write an ACT++ Program

To write an ACT++ program one has to specify all the class definitions required in the application along with the definition of at least the second one of the following methods that belong to a predefined class called **Actorprog**. The methods are:

```
void Actorprog::Setup(),
void Actorprog::Execute(), and
void Actorprog::Terminate().
```

The `Setup` method is used to specify the number of processors and the quantum size for the preemptive scheduler. Note that ACT++ can be run as either a preemptive or a non-preemptive system (i.e., the run time system can use either a preemptive or a non-preemptive scheduler). The `-DPREEMPT` option in the make file for building PRESTO controls the preemptive or non-preemptive behavior of ACT++. The default definition of `Setup` specifies ACT++ to run on a single processor as a preemptive system with a quantum size of 500ms. To change the number of processors and/or the quantum size, the user needs to use the predefined macro `SCHEDULER(x, y)` where x is an integer value specifying the number of processors and y is the new quantum size specified in microseconds. There are ten predefined macros of the form ONE_TICK, TWO_TICKS, THREE_TICKS, through TEN_TICKS that are available for specifying the quantum size. Note that a TICK is equivalent to 100ms which is the default minimum quantum size in PRESTO. For example, to run an application on three processors with a quantum of 700ms, one would do the following.

```
void Actorprog::Setup() {
      SCHEDULER(3, SEVEN_TICKS);
}
```

The `Execute` method contains the main program of the user. The user must create the initial set of behaviors and actors from `Execute` and send the relevant messages to start the computation. Then `Execute` must wait for the computaion to finish - this is a necessity in the current system to ensure correct computation. If at any time all the processors are found to be quiescent (i.e., executing the scheduler *thread*) and there are no ready *threads* in the queue of the scheduler object then the PRESTO run time system will terminate itself. With the possibility of processes waiting for asynchronous I/O requests to complete, *a quiescent system does not indicate an idle system.* Therefore, in the current system, a straightforward way for `Execute` to wait would be to block on a Cbox expecting a reply message from an actor that can determine the end of the computation.

The `Terminate` method provides the user with the ability to do any extra cleanup before the ACT++ run time system terminates. The cleanup could involve doing some house keeping work or deleting program objects, etc.

4. Solution to the Dining Philosophers Problem Using ACT++

We give a complete ACT++ program that solves the dining philosophers problem using semaphores built with eventcounts and sequencers. The program is deadlock free and is modeled around the Pboth procedure for doing simultaneous-P operation on two semaphores that appears in [Reed and Kanodia 79]. We reproduce a slightly different version of Pboth in the following that appears in [Maekawa et al 87].

```
var G, R, S : record
                      T: sequencer;
                      E: eventcount
               end;
procedure Pboth(R, S);
var r, s:integer;
begin
      /*    First lock the coordinated ticket generator     */
      await(G.E, ticket(G.T));
      /*    Get a coordinated set of tickets    */
      r := ticket(R.T);
      s := ticket(S.T);
      advance(G.E);
      /*    Now wait for both R and S to happen */
      await(R.E, r);
      await(S.E, s)
end Pboth
```

The Pboth procedure can be used by a process to enter a critical section that is gaurded by two semaphores. In the procedure records R and S represent the semaphores. Since Pboth is meant to be used in a concurrent environment, the ticket operations on the semaphores themselves have to be done inside a critical section that is implemented by using the semaphore represented by record G. After awaiting for G to reach the value held by it, the current process does two ticket operations on R and S to mark its turn to use the semaphores. After that it does an advance operation on semaphore G so that some other process can use it. Then the process waits for its turn to enter the critical section by doing two consecutive await operations.

We use a `global synchronizer actor` (GSA) to implement the global semaphore `G`. Whenever a philosopher is hungry he sends a message to the GSA. The processing of the request message by the GSA simulates the `await(G.E, ticket(G.T))` in `Pboth`. Then the GSA sends messages to the right and left fork actors of the philosopher who sent the request message. This simulates getting the `r` and `s` tickets in `Pboth`.

After the GSA has serviced its request, the philosopher blocks on two Cboxes expecting reply messages from the left and right fork actors. These simulate the `await(R.E, r)` and `await(S.E, s)` in `Pboth`. When the left and right fork actors process the request messages for this philosopher, they send reply messages to the philosopher unblocking him. Then the philosopher eats and eventually sends reply messages to the fork actors indicating that he is done with them. At the very end, the philosopher replaces himself with a new self that goes through the *Think-Eat-Release* sequence once again. The program follows.

```
#include "actor.h"
class Philo : public Behavior {
      char* philo_id;
      int num_eaten;
      Actor* right_fork;
      Actor* left_fork;
      Actor* gsa;
      IActor* my_iact;
public:
      Philo(Actor*, Actor*, Actor*, char*, IActor*)
      void Think_Eat();
};

class global_sync : public Behavior {
      Cbox* main_cbox;
      int num_req_processed;
      int max_req;        //Max # of requests to be processed
                          //before terminating program
public:
      global_sync(Cbox*, int);
      void decide_turn(Actor*, Actor*, Cbox*, Cbox*);
```

```
};

class fork : public Behavior  {
      int fork_num;
public:
      fork(int f_num)
      {
            fork_num = f_num;
      }
      void serve_philo(Cbox*);
};

Philo::Philo(Actor* r_f, Actor* l_f, Actor* gsa_arg, char* ph_id,
                              IActor* iact_arg)
{
      my_iact = iact_arg;
      philo_id = ph_id;
      num_eaten = 0;
      right_fork = r_f;
      left_fork = l_f;
      gsa = gsa_arg;
}

void Philo::Think_Eat()
{
      int x;
      Cbox* got_rf;
      Cbox* got_lf;
      Cbox* c_rf = new Cbox(FIRST);
      Cbox* c_lf = new Cbox(FIRST);
      Message* m_self = new Message((PFany)&Philo::Think_Eat);
      Message* m_gsa = new Message((PFany)&global_sync::
            decide_turn, right_fork, left_fork, c_rf, c_lf);
      Message* m_iact = new Message(TTYACT::Write, wb, 23);
      for (int i = 1; i < (num_eaten * rand() * 1000000); i++)
            continue;   //Think
      m_gsa->send(gsa); //Send message to GSA
      c_rf->receive(got_rf);  //Wait for right fork
      c_lf->receive(got_lf);  //Wait for left fork
//    Ready to eat at this point
      Wbox* wb = new Wbox(philo_id);
      Message* m_iact = new Message(TTYACT::Write, wb, 23);
      m_iact->send(my_iact);  //Write on terminal
      num_eaten++;      //Increment number of times eaten
      got_rf->send(1);  //Release right fork
      got_lf->send(1);  //Release left fork
      m_self->send(SELF);     //Send message to self
      become(THISBEH);  //Get ready for the next cycle
}

global_sync::global_sync(Cbox* main_cb, int max_requests)
{
      main_cbox = main_cb;
      num_req_processed = 0;
      max_req = max_requests;
```

```
}

void global_sync::decide_turn(Actor* r_f, Actor* l_f,
                  Cbox* c_rf, Cbox* c_lf)
{
      Message* m_rf = new Message((PFany)&fork::serve_philo,
                                                c_rf);
      Message* m_lf = new Message((PFany)&fork::serve_philo,
                                                c_lf);
      m_rf->send(r_f);
      m_lf->send(l_f);
      num_req_processed++;
      if(num_req_processed == max_req)
            main_cbox->send(1);  //Terminate program
                  //Do not specify replacement behavior. If
                  //you do program will not terminate.
      else
            become(THISBEH);  //Otherwise prepare to process
                                    //next message
}

void fork::serve_philo(Cbox* c_act)
{
      int x;
      Cbox* c1 = new Cbox(FIRST);
      c_act->send(c1);
      c1->receive(x);
      become(THISBEH);
}

void Actorprog::Setup()
{
      SCHEDULER(6, ONE_TICK); //Run on 6 processors with a
                                    //100ms quantum
}

void Actorprog::Execute()
{
      int x;
      srand(10);
      Message* m1 = new Message((PFany)&Philo::Think_Eat);
      Message* m2 = new Message((PFany)&Philo::Think_Eat);
      Message* m3 = new Message((PFany)&Philo::Think_Eat);
      Message* m4 = new Message((PFany)&Philo::Think_Eat);
      Message* m5 = new Message((PFany)&Philo::Think_Eat);
      fork* fork_1 = new fork(1);
      fork* fork_2 = new fork(2);
      fork* fork_3 = new fork(3);
      fork* fork_4 = new fork(4);
      fork* fork_5 = new fork(5);
      Actor* f1_act = new Actor(fork_1, 1, "fork1");
      Actor* f2_act = new Actor(fork_2, 1, "fork2");
      Actor* f3_act = new Actor(fork_3, 1, "fork3");
      Actor* f4_act = new Actor(fork_4, 1, "fork4");
      Actor* f5_act = new Actor(fork_5, 1, "fork5");
```

```
        Cbox* my_cbox = new Cbox(FIRST);
        global_sync* gsa_beh = new global_sync(my_cbox, 100);
              //Let the GSA process only 100 request messages
              //before the program terminates.
        Actor* gs_act = new Actor(gsa_beh, 1, "GSA");
        char* fname = "/dev/ttypj";
        IActor* my_iact = new IActor(fname, TTYBEH);
        char* p1 = "\n PHILO A IS EATING NOW \n";
        char* p2 = "\n PHILO B IS EATING NOW \n";
        char* p3 = "\n PHILO C IS EATING NOW \n";
        char* p4 = "\n PHILO D IS EATING NOW \n";
        char* p5 = "\n PHILO E IS EATING NOW \n";
        Philo* philo_beh_A = new Philo(f1_act, f5_act, gs_act, p1,
                                                           my_iact);
        Philo* philo_beh_B = new Philo(f2_act, f1_act, gs_act, p2,
                                                           my_iact);
        Philo* philo_beh_C = new Philo(f3_act, f2_act, gs_act, p3,
                                                           my_iact);
        Philo* philo_beh_D = new Philo(f4_act, f3_act, gs_act, p4,
                                                           my_iact);
        Philo* philo_beh_E = new Philo(f5_act, f4_act, gs_act, p5,
                                                           my_iact);
        Actor* philo_act_A = new Actor(philo_beh_A, 1, "Ph1");
        Actor* philo_act_B = new Actor(philo_beh_B, 1, "Ph2");
        Actor* philo_act_C = new Actor(philo_beh_C, 1, "Ph3");
        Actor* philo_act_D = new Actor(philo_beh_D, 1, "Ph4");
        Actor* philo_act_E = new Actor(philo_beh_E, 1, "Ph5");
        m3->send(philo_act_C);
        m1->send(philo_act_A);
        m2->send(philo_act_B);
        m5->send(philo_act_E);
        m4->send(philo_act_D);
        my_cbox->receive(x);
        cout << "\n THANKS FOR USING ACT++ \n";
}
```

Chapter Three

ACT++ Implementation Guide

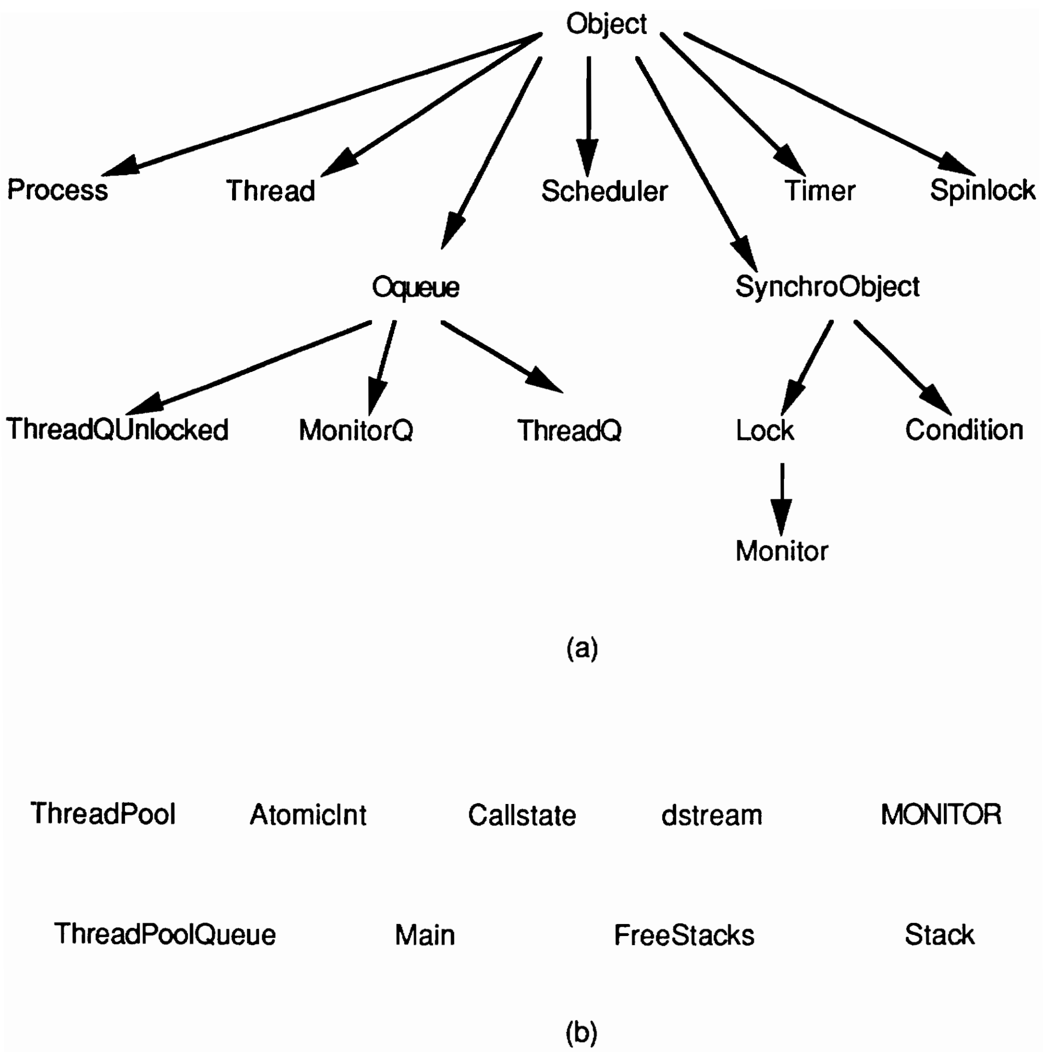
1. Introduction

The current implementation of ACT++ is based on the PRESTO *threads* package. PRESTO is a C++ based programming environment that supports concurrent object-oriented programming. It can be used either as an application programming language or as a systems programming language to implement other programming environments. In the current project, PRESTO has been used to implement ACT++, a concurrent, actor based, object-oriented programming environment.

The class hierarchy of PRESTO is shown in **Figure 1**. The class hierarchy of ACT++ is shown in **Figure 2**. In ACT++ all the classes of PRESTO have not been utilized. The following table shows the PRESTO classes that have been utilized in different ACT++ classes. The utilization has been in the form declaring data members in ACT++ classes which are either pointers to or are declarations of objects that are instantiations of the PRESTO classes.

In the remainder of this chapter some of the basic design issues of PRESTO are discussed. Then the design issues involved in the realization of ACT++ using PRESTO are highlighted. The goal in implementing ACT++ was to avoid any change in PRESTO. That goal could not be achieved. Some of the class definitions of PRESTO have been modified in order to implement some features of ACT++. Those modifications are described and the rationale behind the modifications are discussed.

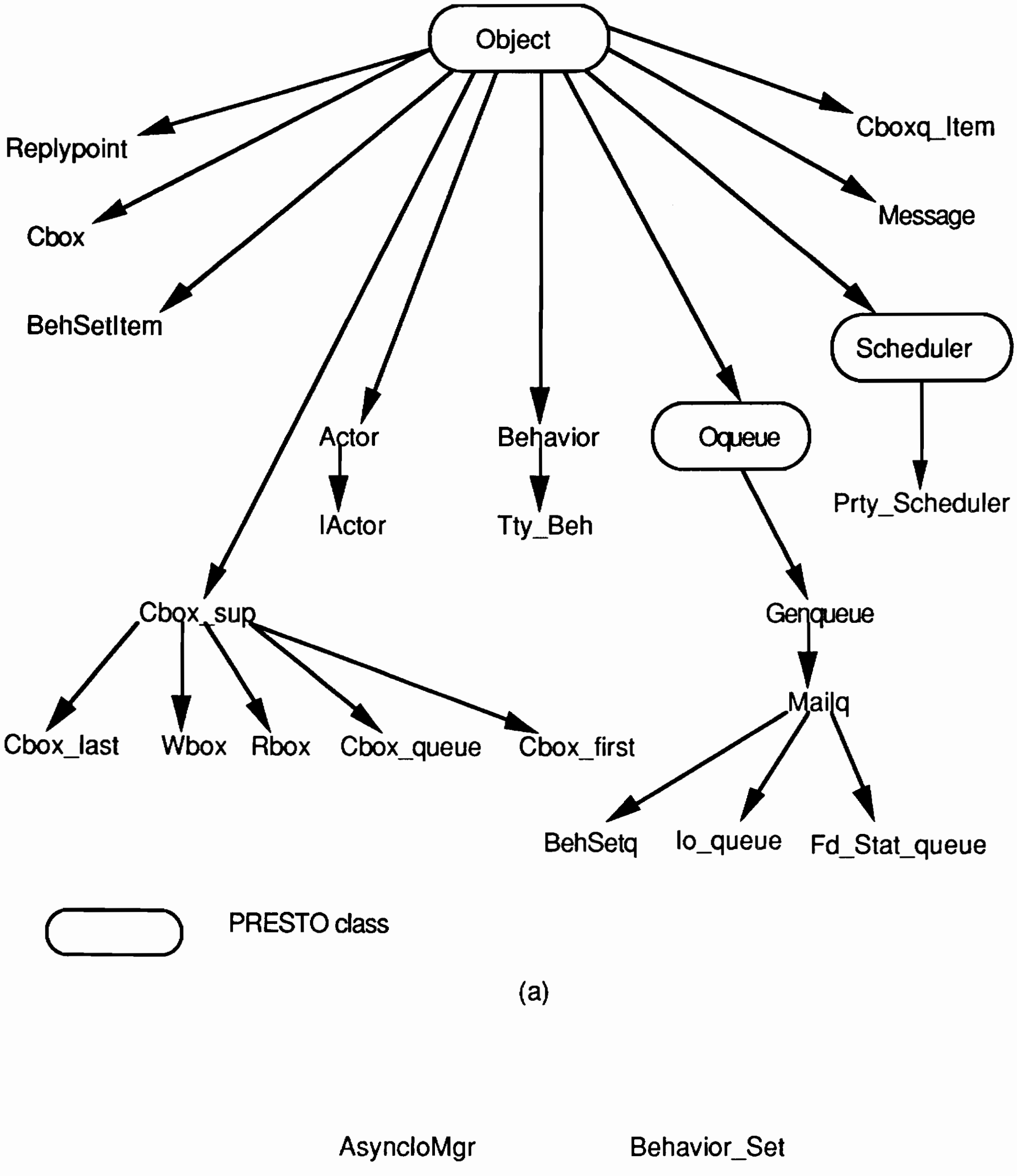
ACT++ class	PRESTO class
Actor	Spinlock, Process
Cbox_queue	Oqueue
Cbox_sup	SynchroObject
Genqueue	Object
Message	Callstate
AsyncIoMgr	Oqueue, SynchroObject
Prty_Scheduler	Spinlock, ThreadPool



(a)

(b)

Figure 1: (a) The class hierarchy in PRESTO. (b) The isolated classes in PRESTO.



(a)

(b)

Figure 2: (a) The ACT++ class hierarchy. (b) Isolated classes in ACT++.

2. Asynchronous Method Invocation in PRESTO

C++ assumes an underlying synchronous and sequential execution environment. There is a single thread of control executing a C++ program and when a method of an object is invoked, it is executed immediately using the thread of control that was executing the invoking method. PRESTO extends the C++ computation environment into a concurrent and an asynchronous one.

The PRESTO model of concurrent computation considers an application program to be a collection of objects that execute concurrently and cooperate with each other to solve a problem. The cooperation is achieved by the objects by invoking each other's operations. But unlike in C++, operation invocation in an object need not be synchronous. That is, the operation invoked in an object need not start executing immediately using the thread of control of the invoking object. Instead, the invoking object can initiate an independent thread of control in the invoked object and schedule it to execute the intended operation. This implies that both the invoking object and the invoked object can execute concurrently.

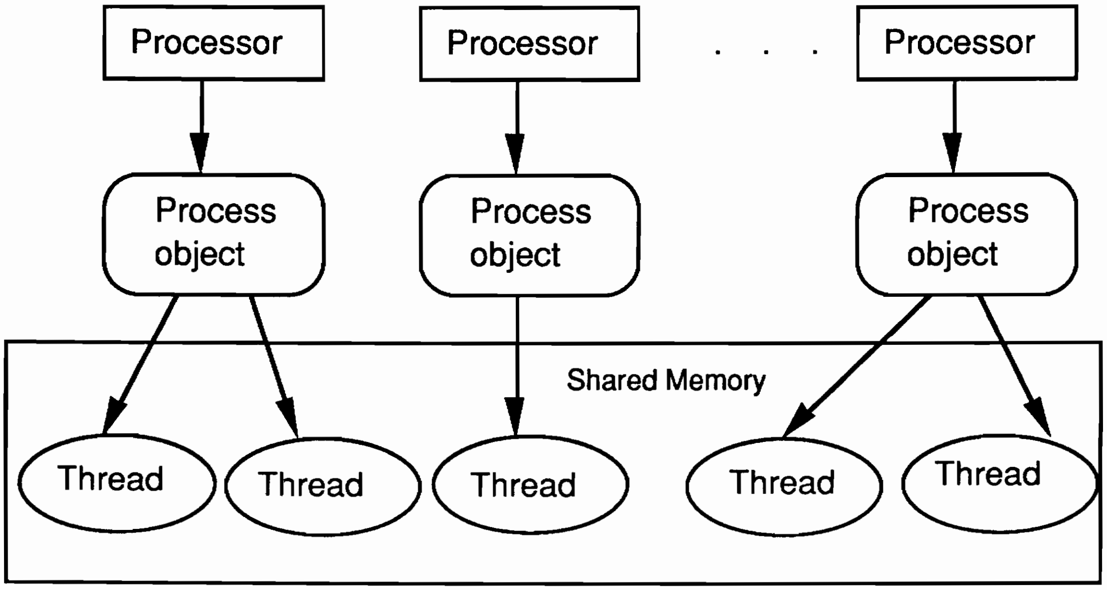


Figure 3: A process object in PRESTO is the abstraction of a process that runs on a physical processor. A process object executes only one thread at any time but a thread can run on many process objects in its lifetime. All threads reside in the shared memory.

To achieve concurrency, PRESTO uses a *thread*, instead of a process, as the unit of schedulable work. A *thread* is an object that is an instantiation of the **Thread** class. A *thread* has its own stack and models a light-weight process. *Threads* execute on *process objects* (**Figure 3**) which are abstractions of the underlying operating system processes. A *thread* can run on any *process object* in its lifetime. When the PRESTO run time system is initialized, only as many process objects are created as their are online CPUs. Thus PRESTO can run on both a uniprocessor and a multiprocessor system.

The current version of ACT++ has been implemented on the Sequent Symmetry shared memory multiprocesor. There are ten processors in this system and hence there can be ten process objects created in an application, one on each processor. Other than the process objects, all objects created in PRESTO and ACT++ reside in the shared memory of the system. This restriction is imposed so that a reference to any object by a *thread* would be valid irrespective of which *process object* the *thread* is executing on. This does not preclude the user from utilizing the feature of creating private objects on the Symmetry, but the management of those private objects would then be the responsibility of the user.

To achieve asynchronous method invocation, PRESTO uses a *callstate* object which is an instantiation of the **Callstate** class. A *callstate* object is used to store the *invocation state* of a method. An *invocation state* is represented by

- a pointer to the method being invoked,
- the number of arguments that are present in the invocation, and
- the actual arguments involved in the invocation.

Every *thread* object contains a *callstate* object as one of its private data members. The contents of the *callstate* object is used to initialize the stack of the *thread* before the *thread* starts execution.

2.1 Trace of an Asynchronous Method Invocation in PRESTO

In the following we will trace the events involved in the asynchronous invocation of a method in PRESTO. The classes that will be considered in the discussion are **Callstate**, **Thread** and **Scheduler**. Some of the data members and methods of these classes that play an important role in the asynchronous invocation of methods are listed below.

```
class Callstate    {
            PFany cs_func;   //method to be executed
            int         cs_argc;  //number of arguments
            int         cs_argvs[CS_MAXARGS]; //the args
public:
            void set(PFany f, int argc, int* argv);
            void call(int *sp = 0, Objany o = 0);
}

class Thread : public Object  {
protected:
            int *t_csp; //current stack pointer
            int *t_fp;  //frame pointer
            Callstate   t_callstate;      //initial call state
            Objany      t_boundobj; //what we are bound to
            virtual void t_start1(Objany obj);
public:
            virtual int start(Objany obj, PFany pf, ...);
            virtual int runrun();
            virtual int run();
            void swtch();
}


class Scheduler : public Object     {
protected:
            ThreadPool *sc_t_ready;  //threads wanting to run
public:
            virtual Thread* getreadythread();
            virtual void reasume(Thread* t);
            void begin(Thread* t)
                  {resume(t);}
}
```

Let us assume that from the *calling_meth* method of an object A, one wants to invoke the *test_meth* method in another object B. To initiate an asynchronous *thread* in *test_meth*, object A has to create a *thread* object first, and then invoke the *start* method in it. The code to do so is shown below.

```
Thread* test_th = new Thread("TEST", 0, 0);
test_th->start(B, (PFany)(B->test_meth), 1, 2.3);
```

The first argument to *start* specifies the object in which the *thread* is to be initiated. The second argument specifies the method which the *thread* has to execute. The third and fourth arguments are the actual arguments necessary to invoke *test_meth*.

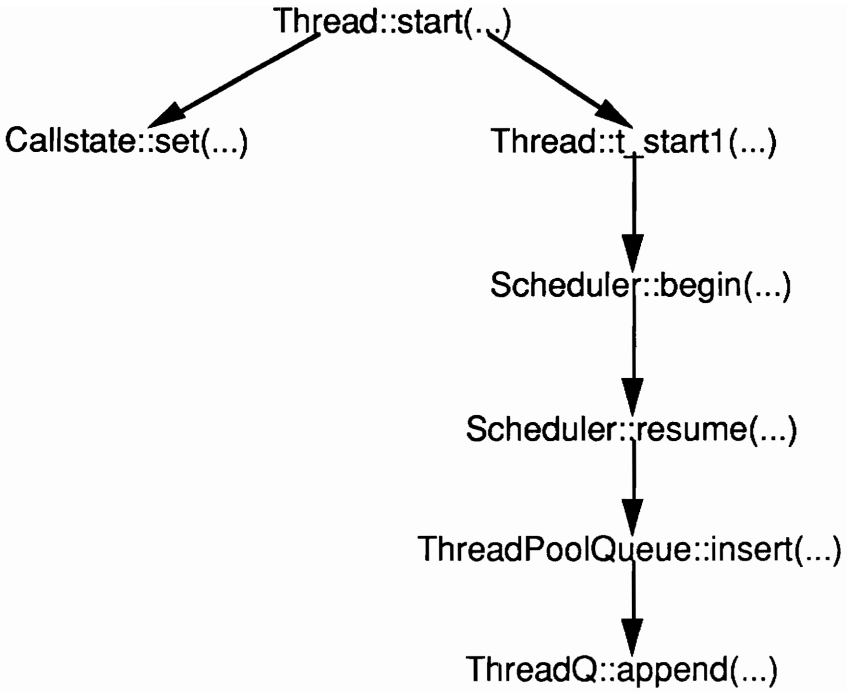


Figure 4: The call sequence resulting from a call to the start method of the Thread class. The methods are specified as class name :: method name. The ellipsis in the argument list are placeholders for the actual arguments.

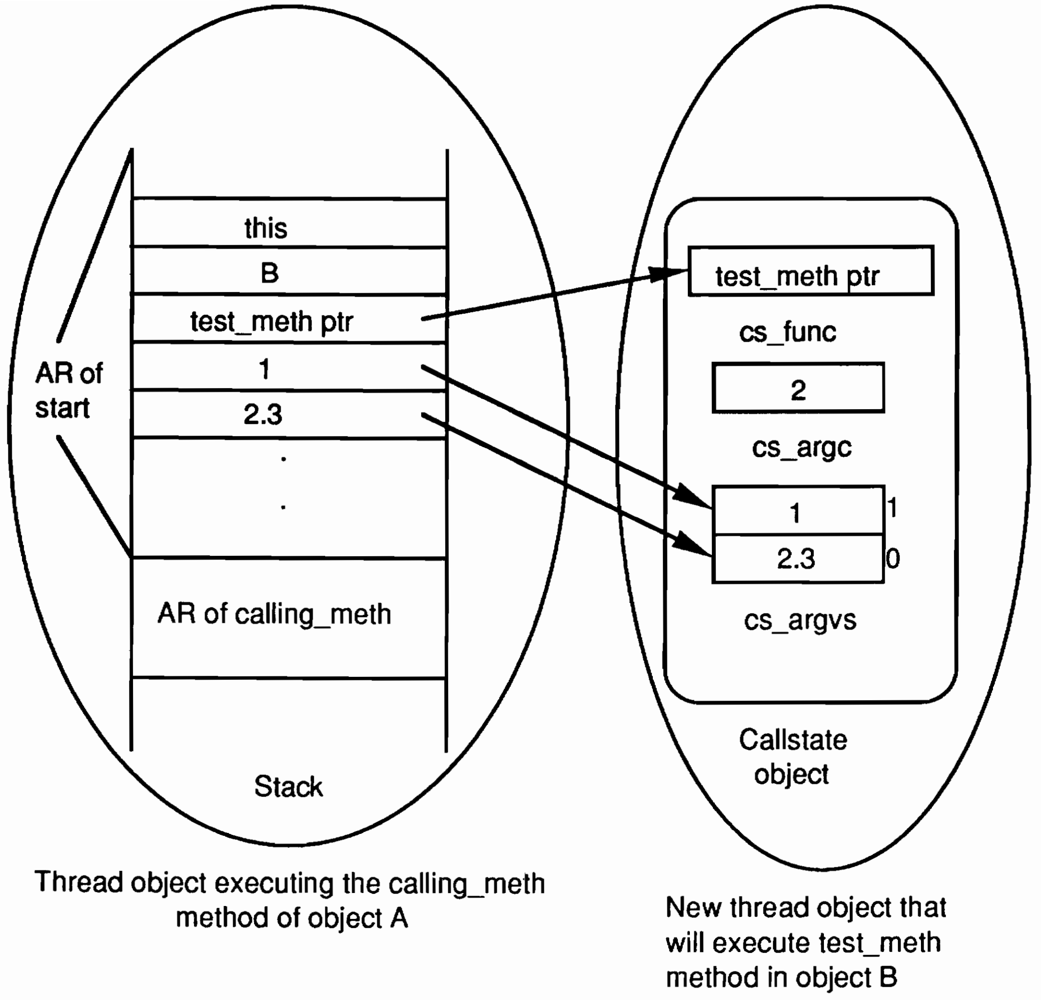

this
B
test_meth ptr
1
2.3
AR of start
AR of calling_meth
Stack
test_meth ptr
cs_func
2
cs_argc
1
2.3
1
0
cs_argvs
Callstate object
Thread object executing the calling_meth method of object A
New thread object that will execute test_meth method in object B

Figure 5: The figure shows how the callstate object of a new thread object is set from the start method of the Thread class.

The call to the *start* method initiates a sequence of method invocations that is shown in **Figure 4**. The first method to be invoked from *start* is the *set* method in the *callstate* object that is a private data member of the *thread* object. This method is responsible for setting up the *callstate* object. The invocation state of the callee is obtained from the **activation record** (AR) of the *start* method. In the case of our example, as a result of the call to *start*, the stack of the *thread* executing *calling_meth* will have the AR of *start* on its local stack as shown in **Figure 5**. The AR of the *set* method will be right on top of the AR of *start*. The *set* method is invoked with a pointer to the method to be invoked, the number of arguments, and a pointer to the beginning of the argument block in the AR of *start*. Figure 5 shows how these items are recorded in the relevant private data members of the *callstate* object. An important point to be noted here is that the arguments of the method to be invoked that are passed to *start* match the ellipsis in the argument list of *start* and as such the compiler is unable to do any sort of type checking on them. Therefore, the price paid for achieving asynchronous method invocation is the loss of type checking of the arguments.

The next method that is invoked from *start* is the *t_start1* method in the newly created *thread*. This method invokes the *begin* method in the *scheduler object*. The *Scheduler* class in PRESTO serves as the template for the *scheduler object* that is created at the time when the PRESTO run time system is initialized. There is only one shared *scheduler object* that stores ready *threads*. When a *thread* is started in the method of an object, it must be placed in the queue of the *scheduler object*.

To achieve the enqueueing of the new *thread*, *begin* invokes the *resume* method of the *scheduler object*. *Resume* invokes the *insert* method in *sc_t_ready*. *Insert* in turn invokes the *append* method of the *ThreadQ* object that the *ThreadPoolQueue* object is composed of, to place the new *thread* in the queue of the *scheduler object*.

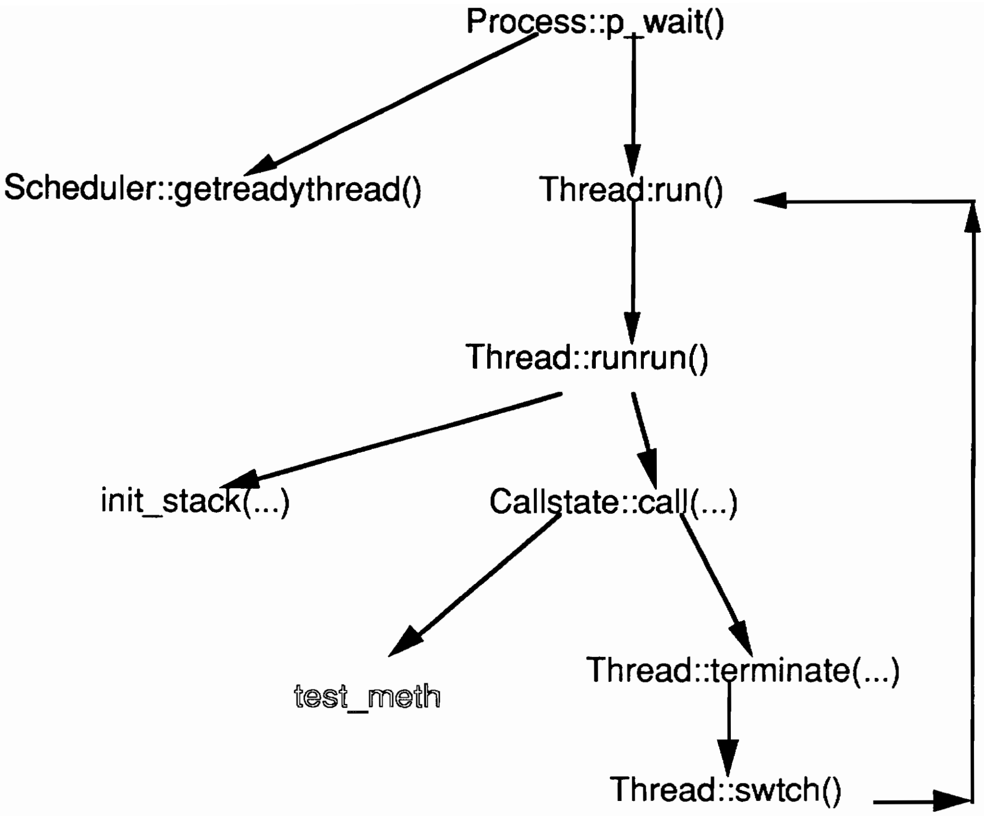


Figure 6: The call graph rooted at the p_wait method of the Process class that is executed by the scheduler thread. Note that instead of returning to terminate, swtch returns control to the run method.

The above sequence of events completes the journey of a new *thread* to the *scheduler object*. What remains is the actual execution of the *thread* by some *process object*. To understand that sequence of events, we have to look closely at the *thread* scheduling mechanism in PRESTO. When the PRESTO run time system is initialized, a unique *thread* is initiated on each *process object* that runs on it forever. This *thread* is responsible for extracting ready *threads* from the *scheduler object's* queue and executing them. Hence it is called the *scheduler thread*. The *scheduler thread* executes in the *p_wait* method of the *process object* that it is associated with. The call sequence rooted at *p_wait* is shown in **Figure 6**.

The *scheduler thread* executes an infinite loop in the *p_wait* method. The first major activity in the loop is to extract a ready *thread* from the queue of the *scheduler object*, if available. This is done by invoking the *getreadythread* method in the *scheduler object*. After extracting a *thread* the *run* method in that *thread* is invoked. The most important activity in *run* is to invoke the *runrun* method in the *thread*. In the *runrun* method, two alternative set of actions can take place depending on the state of the *thread* that was extracted. A *thread* extracted from the scheduler object can either be virgin (i.e., has not been scheduled before for execution) or is being re-scheduled for execution. Since in our example the *thread* that will execute *test_meth* is of the former type we will consider that case first.

If the *thread* is a virgin one the *init_stack* function is called. *Init_stack* is a global function which plays a key role in the scheduling of *virgin threads*. *Init_stack* does some housekeeping work and then switches to the local stack of the *thread* that was just extracted from the *scheduler object*. After the switching any subsequent method invocation takes place on the local stack of the new *thread* and not on the stack of the *scheduler thread*. Among the housekeeping work done in *init_stack*, the stack pointer and the base pointer of the *scheduler thread's* stack just before the switching is recorded in the *t_csp* and *t_fp* data members of the new *thread*. This information is used whenever the *thread* wants to switch back to the *scheduler thread*.

After switching stacks, control returns to *runrun* from *init_stack*. The next major event in *runrun* is to invoke the *call* method in the *callstate object* in the new *thread*. The sequence of events from hereon is shown pictorially in **Figure 7**. *Call* is the method that is responsible for laying out the AR of *test_meth* using the contents of the *callstate object.* After setting up the AR, *test_meth* is finally invoked which then executes to completion. After the method terminates, control returns to *call* which then invokes the *terminate* method in the *thread.* The most important activity in *terminate* is to invoke the *swtch* method in the *thread.* Using the *t_csp* and *t_fp* data members of the *thread* object in which *init_stack* had saved the return stack and frame pointers, *swtch* switches stacks once again and returns to the stack of the *scheduler thread.* Control returns to the *run* method instead of the *runrun* method in the *scheduler thread. Run* then deletes the *thread* and returns to the *p_wait* method in which the *scheduler thread* loops once again looking for the next work to perform.

If the *thread* is not a virgin *thread* then the *swtch* method is invoked in the scheduler *thread. Swtch* switches control to the local stack of the *thread* being re-scheduled and starts execution at the point where the *thread* was executing in its previous run. The return sequence on the completion of the method remains the same as that of a virgin *thread.*

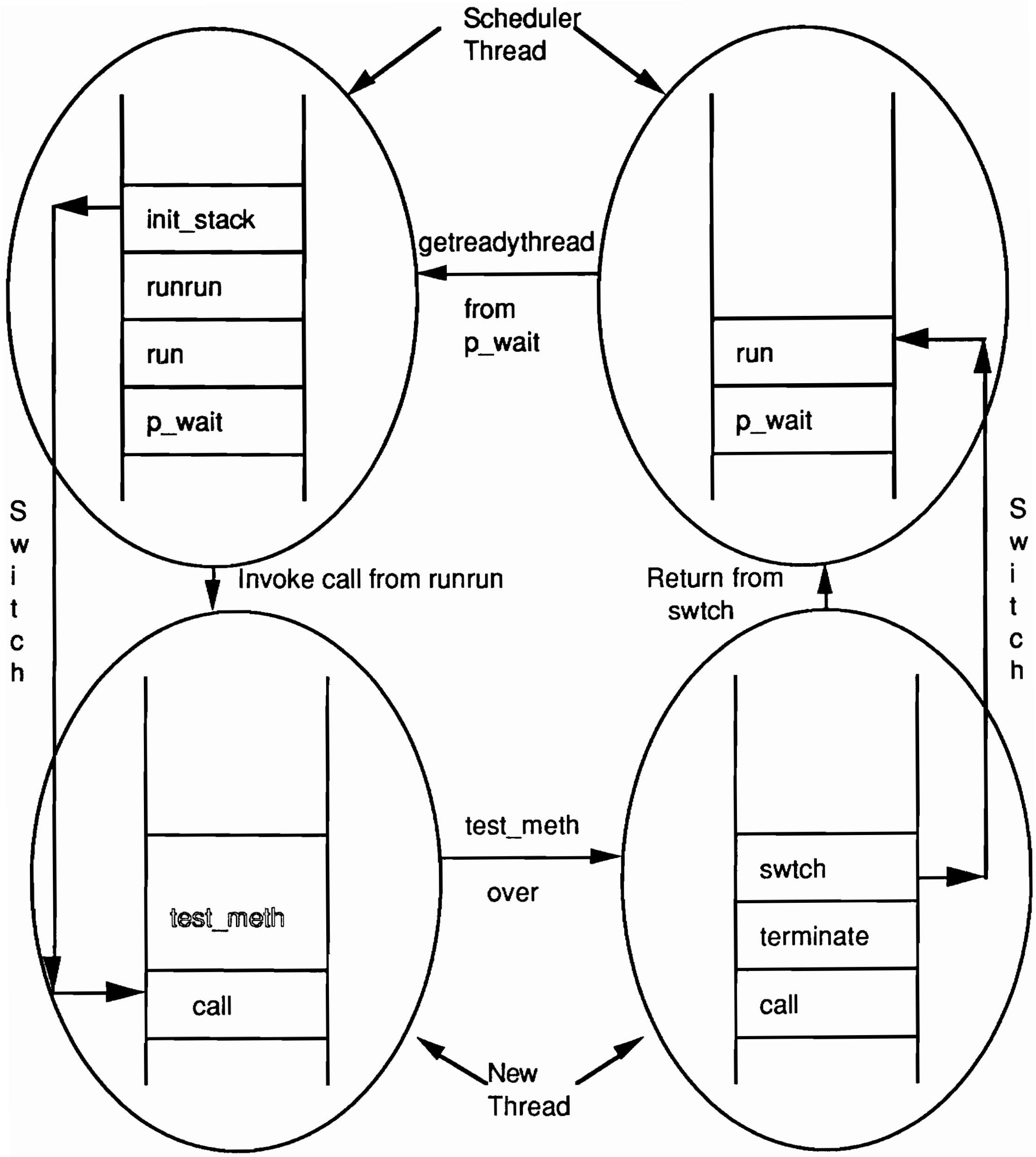


Figure 7: The execution of the asynchronously invoked method is achieved by switching between the stacks of the scheduler thread and that of the thread that is to execute the asynchronously invoked method. The ARs of the different methods participating in the process is shown in the figure.

2.2 Asynchronous Method Invocation in ACT++ and Messages

Asynchronous method invocation occurs in ACT++ when a message is sent to an actor. The message contains the method to be executed in the current behavior when the actor processes the message. The selection of a message object by the actor's current behavior object initiates a unit of activity, a *thread*, that will execute the requested method. The latter issue leads to the following questions.

- Is a new *thread* started in the requested method when the message object is created, and

- when and from where do we create the new *thread* object and invoke the *start* method in it?

One alternative is that we create a new *thread* inside the constructor of the message object and initiate it directly in the requested method of the replacement behavior of the target actor. This will work if the target actor has a replacement behavior available when the message object is created. If that is not so, what should happen to the message object? It must wait in the queue of the target actor and hence there must be an alternate way of scheduling a *thread* in the requested method when the replacement behavior is specified. This splitting of the method invocation process, once from the constructor and once from the actor is not an elegant approach.

Another drawback of the above strategy is the inability to perform the following set of activities once the *thread* has finished executing the requested method.

- Register in the actor that the current behavior has finished execution and increment the number of concurrently executable *thread* counter in the actor by one.

- Initiate the next activity in the actor by checking the availability of a message for the replacement behavior.

- Delete the message object.

- Delete the behavior object if it has not been reused as the replacement behavior.

The above activities are impossible to perform without adding code in the *call* method in the *Callstate* class because the *thread* is terminated immediately after returning from the requested method in *call*. To adhere to our goal of effecting the least possible change to PRESTO, we did not modify the *call* method. The design path that we followed instead was

- to centralize the *thread* creation and *thread* initiation activities and do them from special methods in the *Actor* class, and

- instead of initiating a *thread* directly in the requested method, it was initiated in a special method in the message object.

The latter enabled us to intervene in the *thread* scheduling sequence followed by PRESTO and regain control after the *thread* finished executing the requested method.

2.3 Trace of an Asynchronous Method Invocation in ACT++

This section traces the sequence of events involved in the asynchronous invocation of methods in ACT++. The discussion will be very similar to the one in section 2.1. Since the asynchronous invocation is not explicitly done by creating a *thread* and initiating in the requested method as in PRESTO, the exact sequence and the methods involved will be different in this case.

The classes in ACT++ that play a key role in this process are **Message**, **Actor**, and **Callstate**. Some of the data members and methods of the *Message* and *Actor* classes that will be considered in the discussion are listed below.

```
class Message : public Object {
      Callstate   my_cs;      //call state of async method
      Actor*      my_act;     //target actor
public:
      Message(PFany, ...);
      Message(Actor*, PFany, ...);
      void send(Actor*);
      void send();
      void run(Behavior*);
}

class Actor : public Object    {
      Behavior* getrdybeh();  //return the replacement beh
public:
      virtual void getsched_beh(Message*);
}
```

Let us consider an example similar to the one considered in section 2.1. From the *calling_meth* method of the current behavior of an actor A, the *test_meth* method in a behavior of actor B is invoked. We assume that the behavior of actor B is an instantiation of the *Repl_Beh* class. The first step is to create a message object and send it to the actor B.

The code to do so is shown below (refer to section 2.4 in chapter two for a discussion on the specific methods).

```
Message* new_mess = new Message((PFany)&Repl_Beh::test_meth,
                                                1, 2.3);
new_mess->send(B);
```

In the constructor of the message object, the *set* method of the private *callstate* object in the message is invoked and the *invocation state* of *test_meth* is recorded. This yields the same *callstate* object shown in Figure 5, except that instead of being a private member of the *thread*, this *callstate* is a private member in the message object. This signifies that when the requested method is actually executed by a *thread*, the AR of the requested method has to be set up using this *callstate* and not the *callstate* in the *thread.*

The invocation of the *send* method initiates the sequence of events shown in **Figure 8**. Using the actor pointer B specified in the *send* method, the *getsched_beh* method in B is invoked. The first major activity in *getsched_beh* is to extract the replacement behavior which is done by invoking the *getrdybeh* method in the actor. If a replacement behavior is not available, the message is put in the message queue and is processed later. That sequence of events will be discussed later in conjunction with the discussion on actors. Now we will assume that a replacement behavior is available. In that case, the next activity in *getsched_beh* is to create a *thread t* and then to invoke the *start* method in it in the following way:

```
t->start((Objany) m1, (PFany) &Message::run, b1)
```

where *m1* is a pointer to the message object being processed and *b1* is a pointer to the behavior in actor B. Thus, the new *thread* is initiated in the *run* method in the message object that is being processed by the actor. The invocation of *start* initiates the call sequence shown in Figure 4 and the same discussion applies here too.

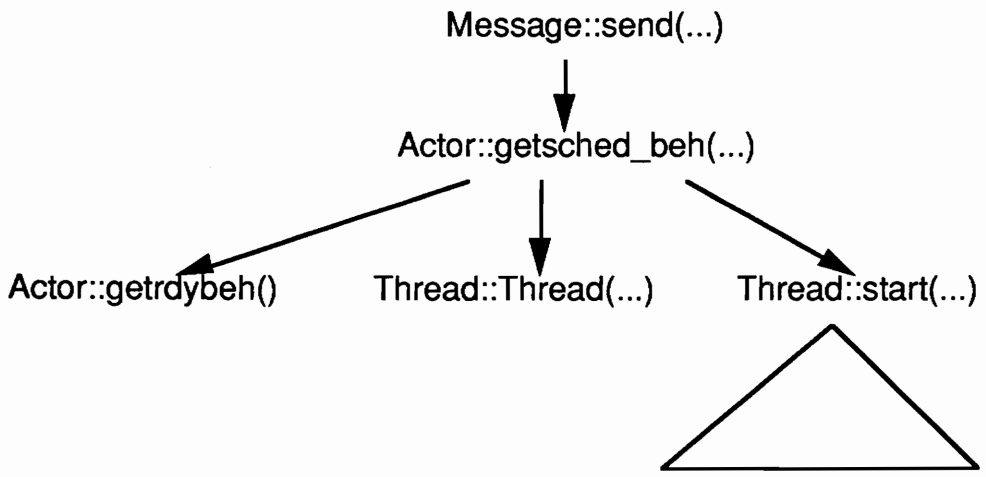


Figure 8: The call sequence resulting from a call to the send method of the Message class. The triangle below Thread::start signifies the presence of the call tree appearing in figure 4 before.

Run is the special method that is used to intervene in the *thread* execution sequence in PRESTO. The implementation of *run* is very similar to that of the *call* method of the *Callstate* class. *Run* uses the *callstate* object in the message to initialize the AR record of the requested method and then actually invokes that method. The call sequence rooted at the *p_wait* method of the *Process* class that appears in Figure 6 is reproduced in **Figure 9** including the slight modification introduced by the incorporation of *run*. The actual execution of the *test_meth* method by the actor B is shown pictorially in **Figure 10** which depicts the interaction of the *scheduler thread* and the *thread* executing *test_meth*.

The only difference between Figure 7 and Figure 10 is the additional AR between *call* and *test_meth*, that of the *run* method in the *Message* class. After *test_meth* finishes, control returns to *Message::run* and from there the behavior object in which *test_meth* was invoked is deleted. Then the message object in which *run* was invoked is deleted. Then control returns to *call* from where control gets switched back to the *scheduler thread* as before. This completes the trace of an asynchronous method invocation in ACT++.

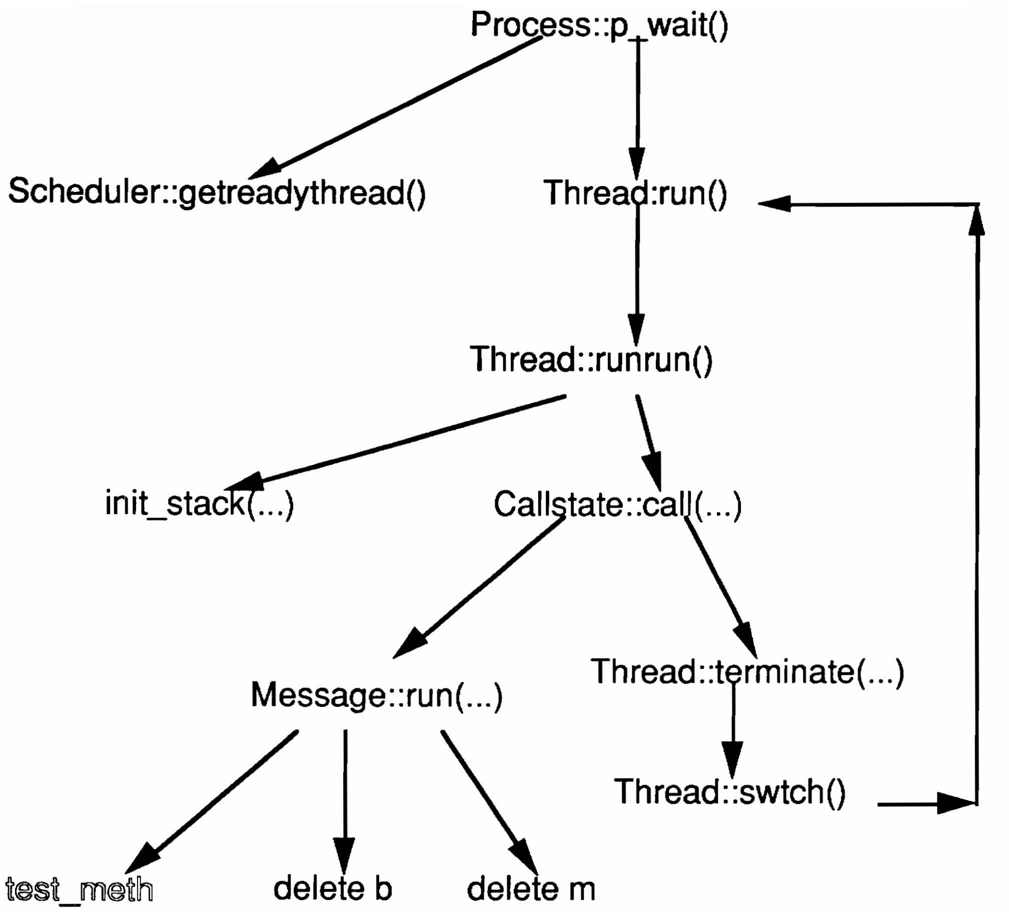


Figure 9: The modified call graph rooted at the p_wait method of the Process class that is executed by the scheduler thread in ACT++. Note that test_meth gets executed by the run method and not the call method as before. Also note that the behavior object and the message object are explicitly deleted in run.

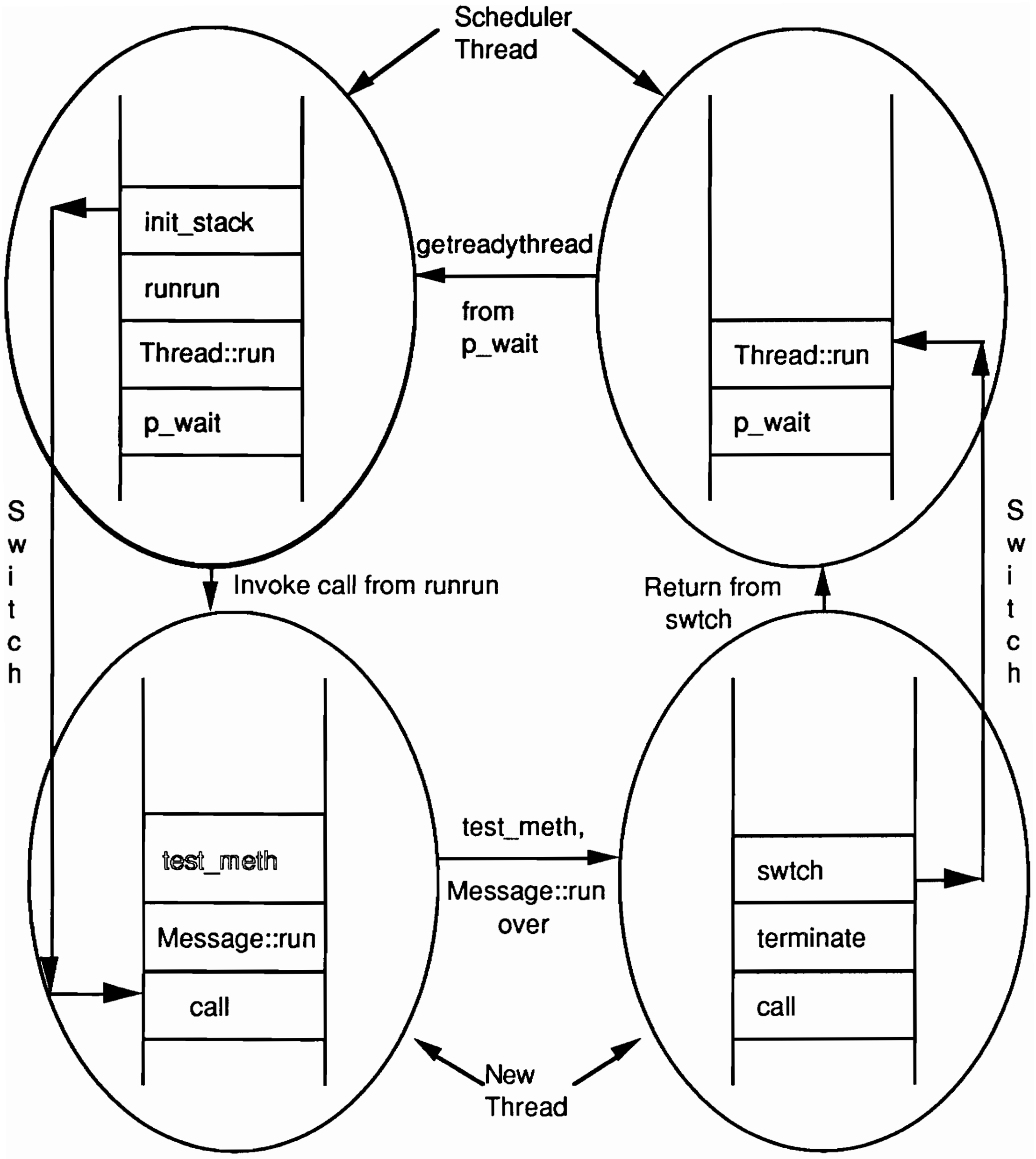


Figure 10: The modified execution sequence of the asynchronously invoked method in ACT++. The call method of the Callstate class no longer executes the test_meth. Instead it executes Message::run which in turn executes test_ meth.

3. The Implementation of Actors

An actor is the only type of object in ACT++ that is capable of scheduling independent *threads* in itself. We will discuss the different components of an actor and its *thread* scheduling ability in this section.

Some data members and methods in the *Actor* class that will appear in the discussion are listed below. A discussion of the individual components follows.

```
class Actor : public Object    {
      Spinlock actor_lock;
      Mailq mailq;
      int mailq_lngth;
      int max_threads;
      int curr_threads;
      Behavior* repl_beh;
      int got_repl;

      Behavior* getrdybeh();
      void put_mess(Message*);
      Message* get_mess();
      void put_beh(Behavior*);
      int test_beh();

public:
      virtual void getsched_beh(Message*);
      virtual void getsched_mess(Behavior*);
      virtual void upd_sched_next();
}
```

The *actor_lock* data member is a *Spinlock* that controls simultaneous access to an actor. Since there can be multiple independent *threads* accessing an actor simultaneously, this lock is necessary to ensure the consistency of the data structures inside the actor, especially the message queue.

The *mailq* data member, an instantiation of the *Mailq* class, implements the message queue. It is responsible for enqueueing request messages that wait for a replacement behavior to process them. The *mailq_lngth* data member is used to record the number of messages in the message queue. It is useful to determine whether a queue is empty without actually searching the queue.

The *max_threads* and *curr_threads* are integer data members that control the level of concurrent activity inside an actor. As noted before, an actor can schedule independent *threads* inside itself but the number of such concurrent *threads* is controlled by the user. Through an argument in the constructor of the actor, the user specifies the number of concurrent *threads* that will be allowed in an actor which is recorded in the *max_threads* data member. To ensure that the number of concurrent *threads* does not exceed this limit, the *curr_threads* data member is used. Whenever a new *thread* is scheduled in the actor, *curr_threads* is incremented by one and whenever a *thread* finishes executing, it is decremented by one.

The *repl_beh* data member is a pointer to the replacement behavior of an actor. It is a pointer to an instantiation of the *Behavior* class in ACT++. Whenever a behavior does a *become* operation and specifies a replacement behavior, if the replacement does not find a message to process, the behavior is recorded in the *repl_beh* data member and waits for the arrival of a message. Another situation in which the *repl_beh* is used to record a behavior is the time when the actor is constructed. An argument in the actor constructor specifies the *initial behavior* of the actor which is recorded in *repl_beh* which then awaits the arrival of a message.

The *got_repl* data member is a boolean variable that is used to indicate the presence of a replacement behavior in the actor. It is examined before initiating the next activity in the actor.

An actor's prime responsibility is to process messages by initiating an independent thread of control for each message processed. An independent thread of control in initiated by creating and scheduling a new *thread* whenever a behavior finds an appropriate message to process.

A new *thread* is created in an actor when any one of the three following conditions occur.

- A new message arrives to the actor and it finds a replacement behavior waiting to process it.
- A replacement behavior is specified and it finds a message waiting to be processed.
- The current behavior finishes execution and finds an appropriate message for the replacement behavior if it is still waiting.

The last case occurs if the actor had *max_threads* number of concurrent *threads* executing simultaneously in it, and as a result, although a message and a replacement behavior were available, the message could not be processed. In such a situation, it is the responsibility of a *thread* that finishes execution to schedule the next activity before terminating.

Corresponding to the three cases above there are three methods in the *Actor* class that schedule the next activity in the actor. The signatures of these methods, in the same order as the different cases listed above, are shown below.

- `virtual void getsched_beh(Message*)`
- `virtual void getsched_mess(Behavior*)`
- `virtual void upd_sched_next()`

We have referred to the *getsched_beh* method in connection with asynchronous method invocation in ACT++. This method is invoked from the *send* methods of a message object at the time when the message is sent to a target actor. The *getsched_mess* method is invoked from the *become* method of a behavior object. The *upd_sched_next* method is invoked from the *run* method of a message object after the asynchronously invoked method has finished execution and the *thread* is about to terminate.

In **Figure 11** we give the flowchart for the *getsched_beh* method. Some of the other methods in the *Actor* class are also specified in the flowchart along with the description of their function. The *getsched_mess* method is very similar to *getsched_beh* barring the following differences:

- instead of enqueueing a message, the replacement behavior is recorded in *repl_beh* using the *put_beh(Behavior*)* method, and

- instead of testing for a replacement behavior, the *mailq_lngth* data member is tested to see if any message is present in the message queue.

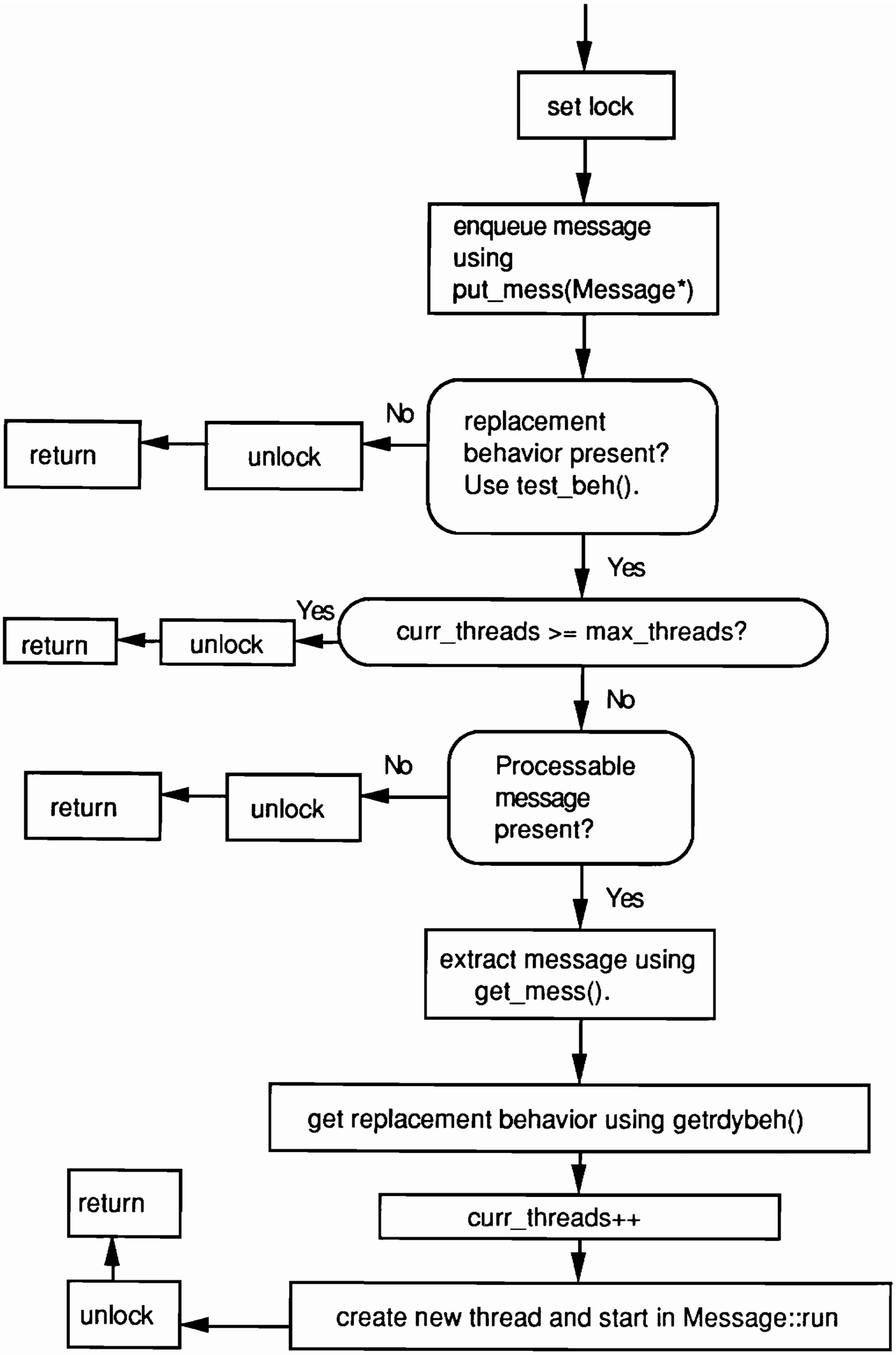

set lock
enqueue message using put_mess(Message*)
replacement behavior present? Use test_beh().
No
unlock
return
Yes
curr_threads >= max_threads?
Yes
unlock
return
No
Processable message present?
No
unlock
return
Yes
extract message using get_mess().
get replacement behavior using getrdybeh()
curr_threads++
create new thread and start in Message::run
unlock
return

Figure 11: The functionality of the getsched_beh method of the Actor class.

The *upd_sched_next* method is also very similar to the previous two except for the following differences.

- The *curr_threads* data member is decremented immediately after setting the lock.
- The criteria for extracting the next processable message are the presence of a message in the queue and the presence of a replacement behavior. The test on *curr_threads* is skipped because the termination of the current *thread* enables another *thread* to start execution in the actor.

4. The Implementation of Behaviors

User defined behaviors in an ACT++ program are subclasses of a predefined *Behavior* class. The data members and methods of the *Behavior* class that will be discussed in this section are shown below.

```
class Behavior : public Object      {
      Actor*      my_actor;
      int         beh_reused;
public:
      void        become(Behavior*);
      int         self_repl();
      void        set_actor(Actor*);
      Actor*      self();
}
```

The *my_actor* data member stores a pointer to the actor object that the behavior object is currently associated with. *My_actor* is assigned in two ways. The first way of setting it is when an actor object is constructed. The actor constructor is supplied a pointer to a behavior object that is the *initial behavior* of the actor. From the constructor of the actor, the *set_actor* method in the behavior is invoked which sets *my_actor* to the actor that is supplied as an argument to it. Another place from where *my_actor* is set is the *become* method. The *set_actor* method in the behavior object that is passed as an argument to *become* is invoked to set *my_actor* in the new behavior object.

The *become* method is used to specify replacement behaviors. The code for the method is shown below.

```
void Behavior::become(Behavior* b)
{
      if (this == b)
            beh_reused = 1;
                  //Same behavior reused as repl
      b->set_actor(my_actor);
                  //use actor pointer of current
                  //behavior to set my_actor of replacement
      my_actor->getsched_mess(b);
                  //initiate a new thread
                  //if message available for replacement.
}
```

As discussed in chapter 2, the *current behavior* can specify itself as the replacement behavior or create a new behavior object instead. If the same object is reused then *become(THISBEH)* is used to specify the replacement. *THISBEH* is a macro that expands to the *this* pointer. Hence, the first thing that is checked in *become* is whether the current behavior object is being reused as the replacement (in that case "*this* == *b*" will be true). If so, the *beh_reused* data member is set to true. This is important because in *Message::run,* just before deleting the behavior object, *beh_reused* is tested using the *self_repl* method, and if set, the behavior is not destructed.

The remaining actions in *become* are to set the *my_actor* data member in the replacement behavior and then to invoke the *getsched_mess* method in the actor which will schedule a new activity in the actor if a message is available for the replacement behavior.

Another useful method in the *Behavior* class is the *self* method. It returns the value of the *my_actor* data member. It is useful when a *behavior* wants to send a message to the actor it is associated with. The format to do that is *send(SELF)* where *SELF* is a macro that expands to a call to the *self* method.

5. The Queue Management Classes in ACT++

In this section, we will look at the class hierarchy that has been implemented to handle the different types of queues needed in ACT++.

The following features of a queue are necessary in ACT++ in connection with the implementation of behavior sets.

- The ability to remove an item from anywhere in a queue.
- The ability to non-destructively examine any item in a queue.
- The ability to search for a particular item in a queue.

PRESTO provides the *Oqueue* class for implementing queues that can store items that are instantiations of the *Object* class. Among the important data members and methods of the *Oqueue* class are the following.

```
struct Oqueue : public Object {
      Object*     oq_head;
      Object*     oq_tail;
      Oqueue(Object* head = 0);
      inline Object* get();   //from head of queue
      inline Object* lookat();      //return non-destructively
      inline void append(Object* ol);      //to end of queue
      int empty() {return oq_head == 0;}
}
```

Since the *Oqueue* class enables the removal of only the item at the head of a queue and does not allow a linear search of a queue, we introduced the *Genqueue* class to record the current item being examined and the last item that was examined. The class definition follows.

```
struct Genqueue : public Oqueue     {
      Object*     curr_pos;
      Object*     last_pos;
      Genqueue();
}
```

To implement the message queue of an actor the *Mailq* class was defined as follows.

```
class Mailq : public Genqueue {
public:
      Mailq();
      int lookat_next(); //non-destructive look at next
      Object* rem_curr(); //dequeue current element
      void reset();
      void shift_pos();
}

Mailq::Mailq() {;}

int Mailq::lookat_next()
{
      if (curr_pos != 0)      {
            Message* temp_ptr = (Message*) curr_pos;
            int i = temp_ptr->get_methodid();
            return i;
      }
      return 0;
}

Object* Mailq::rem_curr()
{
      if (oq_tail == curr_pos)
            oq_tail = last_pos;
      Object* i = curr_pos->o_next;
      (last_pos->o_next) = i;
```

```
        Object* save_curr_pos = curr_pos;
        curr_pos = i;
        return save_curr_pos;
}

void Mailq::reset()
{
        curr_pos = oq_head->o_next;
        last_pos = oq_head;
}

void Mailq::shift_pos()
{
        last_pos = curr_pos;
        curr_pos = curr_pos->o_next;
}
```

The message queue of an actor might be searched for a message that requests the execution of a method that is present in the current behavior set of the replacement behavior. The *lookat_next* method enables the non-destructive lookup of the next element in a linear search of the message queue. The search starts from the head of the queue and the first element is examined using the *lookat* method in the *Oqueue* class. Then, before searching from the second element onwards, the *reset* method is invoked to position the *curr_pos* pointer to the second element and the *last_pos* pointer to the first element of the queue. Then the *lookat_next* method is invoked which returns the content of the second element in the queue as its return value (in the case of a message queue the return value will be the physical address of a method). If the returned value is the one being searched for, then the corresponding element can be removed from the queue using the *rem_curr* method. The item removed is the one pointed to by the *curr_pos* pointer. If the item is not the one being searched for, the search has to continue. But before that, the *curr_pos* and the *last_pos* pointers are shifted one position to the right using the *shift_pos* method in order for the search to proceed correctly.

Although a behavior set is a set of addresses without any underlying ordering among them, we have represented the behavior set using a queue in ACT++. The set in a behavior set is an instantiation of the *BehSetq* class which has the following definition.

```
class BehSetq : public Mailq  {
public:
        BehSetq();
        int lookfor_id(int);
}

BehSetq::BehSetq() {;}

int BehSetq::lookfor_id(int i)
{
        int found = 0;
        int j;
        curr_pos = oq_head;
        while ((found == 0) && (curr_pos != 0))    {
                BehSetItem* temp_ptr = (BehSetItem*) curr_pos;
                j = temp_ptr->get_methodid();
                if (i == j)
                        found = 1;
                curr_pos = curr_pos->o_next;
        }
        if (found)
                return 1;
        else
                return 0;
}
```

A behavior set is constructed by enqueueing the individual items in the *BehSetq* object. A *BehSetq* object utilizes most of the methods in the *Mailq* class for satisfying operations on it. Since a behavior set has also to be searched just to determine the presence of a particular method's address before processing a message, an extra method needs to be defined in the *BehSetq* class that would return a yes/no answer without actually extracting any item from the queue. That is done by the *lookfor_id* method which takes as an argument the item being searched for and returns a 1 if the search is successful, a 0 otherwise.

6. The Implementation of Behavior Sets

As indicated in chapter 2, behavior sets have been introduced in ACT++ to overcome the Inheritence Anomaly. This anomaly - caused by conflicts between synchronization and inheritance - reduces the ability to reuse superclass methods that have data dependent synchronization information embedded in them. There are three classes relevant to the implementation of behavior sets in ACT++. These classes are the *BehSetItem*, *BehSetq*, and *Behavior_Set.* Since behavior sets regulate the interaction among behaviors, messages, and actors, methods were introduced in the *Behavior*, *Message,* and *Actor* classes to manipulate behavior sets. In the following, we specify the data members and methods of all these classes (except *BehSetq* which is discussed in the previous section) that will appear in the discussion in this section.

```
class BehSetItem : public Object    {
      int methodid;
public:
      BehSetItem(int);
      int get_methodid();
}

class Behavior_Set      {
      BehSetq* behsetq;
public:
      Behavior_Set(PFany, ...);
      Behavior_Set();
      int lookfor_id(int);
      friend BehSetq* operator +(Behavior_Set&, Behavior_Set&)
      friend void operator =(Behavior_Set&, Behavior_Setq*);
      friend void operator =(Behavior_Set&, Behavior_Set*);
      int behset_present();
}

class Actor : public Object    {
      int search_behset();
}

class Message : public Object {
public:
      int get_methodid();
}
```

```
class Behavior : public Object      {
protected:
      Behavior_Set curr_bset;
      virtual Behavior_Set NextBehavior_Set() { };
public:
      Behavior_Set* get_behset();
}
```

Together, all the above classes address three aspects related to behavior sets:

- representation of behavior sets,
- specification of behavior sets in ACT++ programs, and
- searching behavior sets.

We will discuss each aspect separately in the following, highlighting the methods that play important roles in realizing each of the above aspects.

6.1 Representation of Behavior Sets

Every behavior set consists of a queue of behavior set items. The behavior set items are instantiations of the *BehSetItem* class and the queue is an instantiation of the *BehSetq* class. Since behavior sets are collections of physical addresses of methods of a behavior subclass, a behavior set item is nothing but an object storing an integer, the integer value of the address of the method. Hence the constructor of *BehSetItem* takes the address of the method as an argument and sets the *methodid* data member equal to it. The *get_methodid* method returns the stored *methodid*. This method is useful when the behavior set is searched for the presence of a particular method.

6.2 Specification of Behavior Sets in ACT++ Programs

The correct way of specifying and using behavior sets in an ACT++ program is shown with an example in section 2.3 of chapter 2. The implementation of the different methods and operator overloadings that implement behavior sets assumes the suggested mode of usage. The salient features of the suggested scheme for using behavior sets are as follows.

- The behavior sets necessary for a user defined behavior class are declared as protected data members.
- The *NextBehavior_Set* method is declared as a protected member.
- The behavior sets are assigned in the constructor of the user defined behavior class.
- The *curr_bset* is assigned explicitly to the behavior set returned by the *NextBehavior_Set* method.

The first feature is realized by using the default constructor *Behavior_Set*. For each declared behavior set, this constructor is invoked which assigns a new empty behavior set queue to it.

The *NextBehavior_Set* method is null in the *Behavior* class. The user has the responsibility of defining this method and returning the proper behavior set.

The assignments of the behavior sets in the constructor of the user defined behavior class are achieved through the use of the *Behavior_Set(PFany, ...)* constructor and the three operator overloadings in the *Behavior_Set* class.

An existing behavior set X is assigned a new behavior set consisting of the methods A and B of a user defined behavior class *Test_Beh* , say, in the following way:

```
X = new Behavior_Set((PFany)&Test_Beh::A, (PFany)&Test_Beh::B)
```

This invokes the first constructor in which, a new behavior set queue is constructed. Then the values of the arguments are extracted and enqueued in the behavior set queue. In the above assignment the left hand side is a reference to a behavior set and the right hand side is a pointer to a behavior set. The goal is to assign the behavior set queue of the newly constructed item as the behavior set queue of X. That is achieved using the following overloading of the '=' operator.

```
void operator =(Behavior_Set& lhs, Behavior_Set* rhs)
{
      delete lhs.behsetq;
      lhs.behsetq = rhs->behsetq;
      rhs->behsetq = 0;
      delete rhs;
}
```

In our example, X becomes the first argument (*lhs*) and the newly constructed behavior set becomes the second argument (*rhs*) of the above overloading. The behavior set queue in X is deleted first. Then the behavior set queue of the new behavior set is assigned to the behavior set queue pointer in X. Then the pointer to the behavior set queue of the new behavior set is set to zero - this detaches the new behavior set from its behavior set queue. Then the new behavior set is deleted. Since the connection to its behavior set queue was severed, the behavior set queue does not get deleted when the behavior set is destructed.

Another way of assigning to a behavior set is to use the overloading of the '+' operator to form the union of two behavior sets. Note that the overloading returns a pointer to a behavior set queue and not a reference to a behavior set. Hence the '+' operator must be used to form the union of only two behavior sets at a time.

If X and Y are two behavior sets then we might assign a third behavior set Z as follows:

```
Z = X + Y;
```

The above first invokes the '+' opeartor overloading which is defined as below.

```
BehSetq* operator +(Behavior_Set& b1, Behavior_Set& b2)
{
      Behavior_Set* bnew = new Behavior_Set();
      if (!(b1.behsetq->empty()))   {
            BehSetItem* bs = (BehSetItem*) b1.behsetq->lookat();
            int i = bs->get_methodid();
            BehSetItem* b = new BehSetItem(i);
            bnew->behsetq->append(b);
            b1.behsetq->reset();
            i = b1.behsetq->lookat_next();
            while (i)   {
                  BehSetItem* b = new BehSetItem(i);
                  bnew->behsetq->append(b);
                  b1.behsetq->shift_pos();
                  i = b1.behsetq->lookat_next();
            }
      }
      if (!(b2.behsetq->empty()))   {
            bs = (BehSetItem*) b2.behsetq->lookat();
            i = bs->get_methodid();
            b = new BehSetItem(i);
            bnew->behsetq->append(b);
            b2.behsetq->reset();
            i = b2.behsetq->lookat_next();
            while (i)   {
                  BehSetItem* b = new BehSetItem(i);
                  bnew->behsetq->append(b);
                  b2.behsetq->shift_pos();
                  i = b2.behsetq->lookat_next();
            }
      }
      BehSetq* bsq = bnew->behsetq;
      bnew->behsetq = 0;
      delete bnew;
      return bsq;
}
```

The first thing done in the above overloading is to construct a new behavior set. Then the behavior sets passed as arguments are processed in the following way. Each item in the behavior set is extracted non-destructively, a behavior set item is constructed out of the extracted item and then it is enqueued in the behavior set queue of the new behavior set. This continues until there are no elements left in the behavior set. This search process is exactly the same as the one described in section 5 of this chapter about searching the message queue of an actor. After both the behavior sets are processed in this way, the behavior set queue in the new behavior set is returned. There are two things to be noted in this regard. First, the union operator does not destroy the behavior sets that are participating in the union. Second, the new behavior set queue returned might have the same element more than once. This implies that a multiset is created instead of an ordinary set. This might increase the search time of a behavior set but simplifies the union process.

After the behavior set queue is returned as the result of the union process, the same problem that occurred in the previous example arises - the left hand side is a reference to a behavior set whereas the right hand side is a pointer to a behavior set queue. A second overloading of the '=' operator is used to implement this assignment which is defined as follows.

```
void operator =(Behavior_Set& b1, BehSetq* bsq)
{
      delete b1.behsetq;
      b1.behsetq = bsq;
}
```

In our example Z = X + Y, the first argument to the above operator will be Z and the second argument will be a pointer to behavior set queue returned by '+'. The behavior set queue of Z is deleted and is reassigned to point to the new behavior set queue. Thus, both

X and Y will remain intact after the union is over and Z will be assigned the union of the behavior set items in X and Y.

The last issue involved in using behavior sets is to set the *curr_bset* data member of the behavior object to the current behavior set. That is achieved through the following assignment:

```
curr_bset = NextBehavior_Set().
```

The above assignment statement is available as a macro in ACT++ - the *CURRENT* macro. Note that no special overloading of the '=' operator is necessary for the above assignment - the predefined overloading of '=' in C++ suffices in this case. Section 2.3 in chapter 2 is referred to for a discussion on the placement of *CURRENT* in the methods of the user defined behavior classes.

6.3 Searching Behavior Sets

The last and most important aspect of using behavior sets is searching behavior sets to decide which message to process. Before processing a message in an actor, the behavior set of the replacement behavior, if any (there can be behaviors which do not use behavior sets in which case the very first message in the queue will be processed), must be searched to determine the first message in the queue that can be serviced by the replacement behavior. This search is realized by the *search_behset* method in the *Actor* class.

The incorporation of behavior sets slightly changes the implementation of the *getsched_beh*, *getsched_mess*, and *upd_sched_next* methods in the *Actor* class. Referring to Figure 11, the modification involves the expansion of the *Processable message present?* and the *extract message using get_mess* boxes as shown in **Figure 12**. The *search_behset* method returns a non-zero value if there is at least one message in the message queue that requests the execution of a method present in the behavior set. If a 1 is returned then the very first message in the message queue is processable and the *get_mess* method in the actor object is used to extract it. If a 2 is returned then it indicates that a message beyond the first one in the message queue is processable and that is extracted using the *rem_curr* method in the *mailq* object.

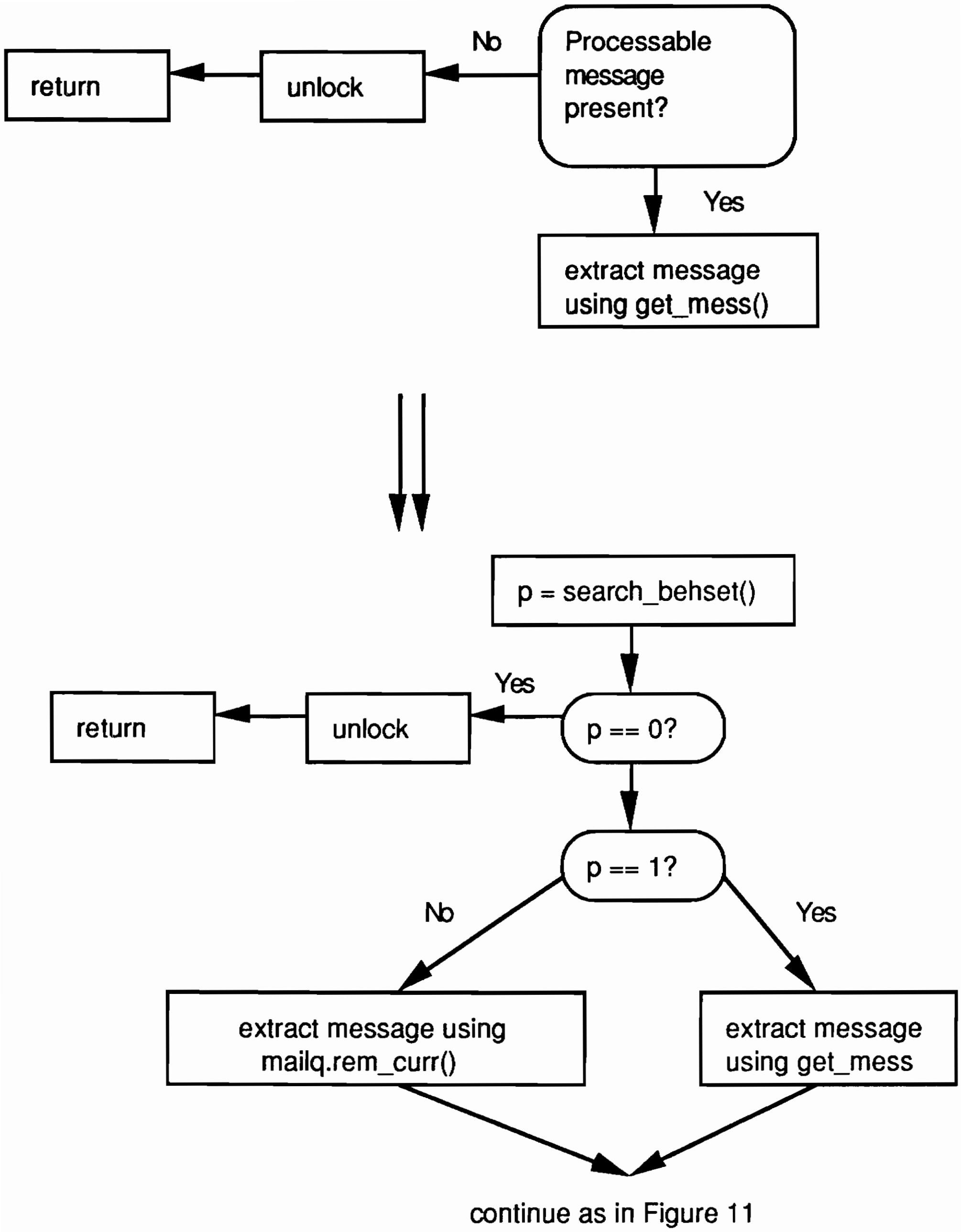

return
unlock
No
Processable message present?
Yes
extract message using get_mess()
p = search_behset()
Yes
return
unlock
p == 0?
p == 1?
No
Yes
extract message using mailq.rem_curr()
extract message using get_mess
continue as in Figure 11

Figure 12: The modification in the implementation of the getsched_beh, getsched_mess, and upd_sched_mess methods in the Actor class due to the incorporation of behavior sets.

The exact sequence of events involved in the *search_behset* method is as follows.

- Step 1: The first message in the message queue is extracted (non-destructively).
- Step 2: Using the *get_methodid* method in the message object the address of the requested method is extracted.
- Step 3: Using the *get_behset* method in the replacement behavior its behavior set is extracted. From the *get_behset* method, the *behset_present* method in *curr_bset* is invoked to determine whether a behavior set is present. The behavior set is returned if present. Otherwise a 0 is returned.
- Step 4: If no behavior set is present a 1 is returned. Note that if the behavior is not using a behavior set then the messages are processed in the default manner which is to process the very first message in the queue.
- Step 5: If a behavior set is present then the *lookfor_id* method in the extracted behavior set is invoked with the value returned by the *get_methodid* method as an argument. This in turn invokes the *lookfor_id* method in the behavior set queue of the behavior set. Inside the *lookfor_id* method, all the behavior set items are searched for the method in question and success is reported by returning a 1 and failure by returning a 0.
- Step 6: If a 1 was returned by *lookfor_id* a 1 is returned. Note that returning a 1 here indicates that the first message is processable by the replacement behavior since the requested method was present in the behavior set.
- Step 7: Otherwise prepare to search the remainder of the message queue by readjusting queue pointers. If all the queue has been searched then return a 0.

- Step 8: If all the messages have not been looked at then extract (non-destructively) the method address from the next message in the queue. Invoke the *lookfor_id* method in the behavior set with the extracted method address as an argument.
- Step 9: If a 1 was returned by *lookfor_id* in step 8 then return a 2. Note that returning a 2 indicates that a message other than the first one is to be processed by the replacement. If a 0 was returned by *lookfor_id* in step 8 then go to step 7.

7. The Implementation of Cboxes

Just as a *request message* is the means of invoking an operation in an actor, a *reply message* is the means of returning the result of an operation to an actor. Unlike the single possible return value of a function invocation in C++, an actor can send multiple reply messages. Before an actor can receive a reply message it has to create a Cbox matching the type of the information expected, pass a pointer to the Cbox to the replying actor, and invoke a data receiving operation on the Cbox. If the replying actor has already filled the Cbox with the reply message when the data extracting operation is invoked the requesting actor extracts the data. Otherwise the requesting actor blocks until the replying actor fills the Cbox.

From the intended usage of Cboxes outlined above, it is evident that the following capabilities must be available in order to use Cboxes.

- The ability to create Cboxes that will be able to store different types of data values.

- The ability to block the current *thread* as a result of a data receiving operation on a Cbox that is yet to be filled by the replying actor.

- The ability to unblock a *thread* that is waiting for a reply to arrive in a Cbox. This unblocking has to be done by the *thread* executing the replying actor at the time when the Cbox is filled with the data.

In addition, we define Cboxes that handle the reply messages in three different ways. The first type retains the very first reply message it receives and ignores all the subsequent ones. The second type retains only the most current reply message it receives and deletes all prior messages. The third type enqueues all reply messages. The common functionality among these three Cbox variations is captured by the *Cbox_sup* class. Three subclasses of *Cbox_sup*, namely, *Cbox_first*, *Cbox_last*, and *Cbox_queue*, capture the variations in reply message handling.

To free the user from handling the details of the three variations of Cboxes and in order to provide a homogeneous interface, we have provided the *Cbox* class. All user created Cboxes in an ACT++ program are instantiations of the *Cbox* class. The recording and extraction of reply messages in Cboxes are available as methods of the *Cbox* class as shown below.

```
class Cbox : public Object    {
            Cbox_sup* cb;
public:
            Cbox(int);
            void send(int&);
            void send(float&);
            void send(char&);
            void send(double&);
            void send(char*, int);
            void send(Cbox*);
            void send(Actor*);
            void receive(int&);
            void receive(float&);
            void receive(char&);
            void receive(double&);
            void receive(void*&);
            void receive(Cbox*&);
            void receive(Actor*&);
            void flush()
            { cb->flush();}
}

Cbox::Cbox(int i)
{
      if (i==FIRST)
            cb = new Cbox_first();
      if (i==LAST)
            cb = new Cbox_last();
      if (i==QUEUE)
            cb = new Cbox_queue();
}

void Cbox::send(int& i)
{
      cb->send(i);
}

void Cbox::receive(int& i)
{
      cb->receive(i);
}
```

The argument to the constructor is one of the predefined macros FIRST, LAST, and QUEUE. As the names suggest, FIRST denotes a Cbox that retains only the first message, LAST denotes a Cbox that retains only the most current message, and QUEUE denotes a Cbox that queues all the messages. Depending on the argument specified, an object of one of the three subclasses of *Cbox_sup* is instantiated in the *Cbox* constructor and that is remembered in the *cb* data member.

There are two major classes of methods in the *Cbox* class. The overloadings of the *send* methods are used to send reply messages of different types. Accordingly, a *send* method in the Cbox class invokes the corresponding *send* method in the actual Cbox object pointed to by the *cb* data member. Similarly, a *receive* method, which is used to extract a reply message from a Cbox, invokes the corresponding *receive* method in the Cbox object pointed to by *cb*.

In the following we discuss in detail the implementation of the different types of Cboxes.

7.1 The Implementation of the Cbox_sup Class

The *Cbox_sup* class implements the common features of Cboxes. Two other types of objects, Rboxes and Wboxes that are used in doing I/O in ACT++, also share some of the properties of Cboxes. As a result, some methods in *Cbox_sup* are used only in association with these objects. The definition of *Cbox_sup* is as follows.

```
class Cbox_sup : public Object      {
protected:
            union {
                  int int_elem;
                  char char_elem;
                  double double_elem;
                  struct_item* st_elem;
                  Cbox* cbox_elem;
                  Actor* actor_elem;
            }     cbs;
            int have_reply;
            SynchroObject* Cbox_lock;
            void thread_mgmt();
            virtual int get_int_elem();
            virtual char get_char_elem();
            virtual double get_double_elem();
            virtual struct_item* get_struct_elem();
            virtual Cbox* get_cbox_elem();
            virtual Actor* get_actor_elem();
            virtual void reset_have_reply();
public:
            Cbox_sup()
            {
                  have_reply = 0;
                  Cbox_lock = new SynchroObject(1);
            }
            virtual void send(int&);
            virtual void send(float&);
            virtual void send(char&);
            virtual void send(double&);
            virtual void send(char*, int);
            virtual void send(Cbox*);
            virtual void send(Actor*);
            void receive(int&);
            void receive(float&);
            void receive(char&);
            void receive(double&);
            void receive(void*&);
            void receive(Cbox*&);
            void receive(Actor*&);
            void wait();
            void wakeup_thread();
            virtual char* ret_buff();
            virtual void flush() {;};
}
```

The elements of the *cbs* union data member are used to store the data value that the Cbox is meant to carry. There are six elements of *cbs*. Of these six, three are for storing the basic scalar types - integer, character, float, and double. Note that the *double_elem* element is used to store data of both the *float* and *double* types. The remaining three are used to store a pointer to a Cbox, a pointer to an actor, and a pointer to a user defined structure. The *have_reply* data member indicates whether the Cbox has a reply message or not. The *Cbox_lock* data member is used to serialize access to the private data members of the Cbox.

The methods of the *Cbox_sup* class, except for *thread_mgmt, reset_have_reply, ret_buff, wakeup_thread,* and *flush* fall into the three following categories.

- The *send* methods are used to record the reply message into the Cbox. Since the handling of reply messages by the three types of Cboxes are different, these methods are declared in the *Cbox_sup* class but actually defined in the three subclasses that correspond to each of the types.

- The seven overloadings of the *receive* method that have one argument are used to extract the data value stored in the Cbox. Along with the *thread_mgmt* method, the *receive* methods implement *thread* blocking when an empty Cbox is interrogated for a reply message.

  The *wait* method is similar to the *receive* method and is used by Rboxes and Wboxes to implement *thread* blocking only. This is because no data is ever extracted out of a Wbox and Rboxes use a different operation to extract data stored in them. On invoking the *wait* method on a Wbox, a *thread* blocks if the content of the Wbox has not yet been written and is awakened when that is done. On invoking the *wait* method on a Rbox, a *thread* blocks if the Rbox has not yet received the data to be read and is awakened when that occurs.

- The methods of the form *get_X_elem* are used to extract the actual data value that the Cbox carries. These methods just return the relevant data member that stores the information corresponding to the type name denoted by *X*. Note that in the *receive* methods which are responsible for data extraction, one could have used the relevant data storing element (i.e. *X_elem* element of the *cbs* data member) to extract the data. But the extraction mechanism on queue type Cboxes varies from those of the other two types. In order to use the same set of *receive* methods for extracting data from queue type Cboxes, the *get_X_elem* methods have been defined. In the queue case, the *get_X_elem* mehods defined in the *Cbox_queue* class are actually used instead of the ones in the *Cbox_sup* class.

Let us consider the *thread_mgmt* method and one of the *receive* methods to see how data extraction from Cboxes occurs. The general structure of all the *receive* methods is exactly the same except that each is responsible for extracting a different type of information. We will consider the *receive* method that extracts an integer value from a Cbox. The definitions of the methods are as follows.

```
void Cbox_sup::receive(int& i)
{
	Cbox_lock->lock();
	thread_mgmt();
	i = get_int_elem();
	reset_have_reply();
	Cbox_lock->unlock();
	return;
}

void Cbox_sup::thread_mgmt()
{
	while (have_reply == 0)		{
		Cbox_lock->remember(thisthread);
		Cbox_lock->unlock();
		thisthread->sleep(Cbox_lock);
		Cbox_lock->lock();
	}
	return;
}
```

The first thing done in *receive* is to obtain exclusive access to the Cbox by locking *Cbox_lock*. Then the *thread_mgmt* method is invoked. In *thread_mgmt*, it is determined whether the Cbox has received the reply message yet. If so control returns immediately. Otherwise the following steps necessary to block the current *thread* are taken.

The first step in *thread_mgmt* is to register the current *thread* in the queue of the *Cbox_lock* object (instantiations of the *SynchroObject* class in PRESTO have the capability of remembering the *threads* that are waiting for the lock in the *SynchroObject* to be released). Next the *Cbox_lock* is unlocked enabling other *threads* to access the Cbox. Then the current *thread* is put to sleep by invoking the *sleep* method in the current *thread* object. Upon awakening the *thread* invokes the *lock* method in *Cbox_lock*. Then control loops back to the beginning of the while statement. The while statement ensures that if between the time when the *thread* is awakened and the time when it actually executes the *receive* method, another actor has extracted the contents of the Cbox (if the pointer to a Cbox is shared between behaviors and actors), then the awakened *thread* will be put to sleep once again.

On returning from *thread_mgmt*, the current *thread* is guaranteed that a reply message is available. In this case, it is extracted by invoking the *get_int_elem* method and assigned to the reference argument *i*. Then the *reset_have_reply* method is invoked which assigns a zero to the *have_reply* data member indicating that the reply message has been extracted. At the end, the *Cbox_lock* is unlocked and control returns from the method.

The same set of activities is repeated for extracting the remaining types of data members except for the pointer to the structure type. The *st_elem* element of the *cbs* data member is a pointer to an instantiation of the *struct_item* class which is defined below.

```
struct struct_item      {
      char* struct_item;
      int struct_len;
}
```

The *struct_item* member stores the pointer to the actual user defined structure that is sent by the replying actor. The *struct_len* member stores the length of the structure. The following code appears in the *receive* method for extracting the structure pointer element from a Cbox in the place where *i* = *get_int_elem()* appeared in the above example.

```
struct_item* y = get_struct_elem();
char* y1 = y->struct_elem;
int z = y->struct_len;
bcopy((char*) y1, (char*&) c, z);
```

After extracting the *struct_item* pointer into *y*, the actual *struct_elem* and its length are recorded in *y1* and *z* respectively. Then the memory block copy function *bcopy* is invoked to copy the contents of *struct_elem* into the reference structure-pointer argument.

The *wait* method has the following definition.

```
void Cbox_sup::wait()
{
        Cbox_lock->lock();
        thread_mgmt();
        reset_have_reply();
        Cbox_lock->unlock();
        return;
}
```

The above is similar to the *receive* methods in all respects except that no data extraction takes place.

The *wakeup_thread* method is invoked on Wboxes and Rboxes. It is defined as below.

```
void Cbox_sup::wakeup_thread()
{
        Cbox_lock->lock();
        have_reply = 1;
        Thread* t = Cbox_lock->recall();
        if (t != 0)
                t->wakeup(Cbox_lock);
        Cbox_lock->unlock();
        return;
}
```

The *wakeup_thread* method serves two major purposes. First, it sets the *have_reply* data member to 1. For Wboxes this signifies that the content of the Wbox has been written. For

Rboxes this signifies that data has been read into the Rbox. Second, any *thread* waiting on the Wbox or the Rbox is awakened. To do so, the *recall* method is invoked in *Cbox_lock* to obtain the first *thread* that is waiting. If no thread is waiting, *Cbox_lock* is unlocked and control returns. Otherwise the *thread* is awakened by invoking the *wakeup* method in it before returning.

The *ret_buff* method is an empty method declared in this class but is actually defined in the Wbox and Rbox subclasses. Its purpose is to return a pointer to the data buffer inside a Wbox or a Rbox.

The *flush* is also an empty method declared in this class but is defined in *Cbox_queue*. The idea behind the *flush* method is to delete the content of a Cbox and prepare it to be used for accepting a reply message once again. For the FIRST and LAST type Cboxes, extraction of the reply message renders the Cbox empty and thereby prepares it to be reused once again. But for a QUEUE type Cbox, extraction of one reply message does not imply that the Cbox is empty. If at any time it is required to ignore all the existing reply messages in a QUEUE type Cbox before receiving any further reply messages in it, the *flush* method must be used.

7.2 The Implementation of FIRST and LAST Cboxes

The FIRST and LAST types of Cboxes are implemented by the *Cbox_first* and *Cbox_last* classes respectively. The definition of the *Cbox_first* class is as follows.

```
class Cbox_first : public Cbox_sup  {
public:
            Cbox_first();
            void send(int&);
            void send(float&);
            void send(char&);
            void send(double&);
            void send(char*, int);
            void send(Cbox*);
            void send(Actor*);
}
```

The definition of the *Cbox_last* class is similar except for the name of the constructor. Both the constructors in *Cbox_first* and *Cbox_last* are empty.

Let us consider the *Cbox_first* class first. The overloadings of the *send* method are for sending reply messages to Cboxes. All the *send* methods except the one having two arguments are similar in structure. Let us consider the *send* method that is used to send an integer as the reply message.

```
void Cbox_first::send(int& i)
{
      Cbox_lock->lock();
      if (have_reply)    {
            Cbox_lock->unlock();
            return;
      }
      cbs.int_elem = i;
      have_reply = 1;
      Thread* t = Cbox_lock->recall();
      if (t != 0)
            t->wakeup(Cbox_lock);
      Cbox_lock->unlock();
      return;
}
```

The first action is to lock the Cbox. Then it is determined whether the Cbox already has a reply message. If so, this being a FIRST type Cbox in which all messages but the very first one are ignored, the current message is ignored, the Cbox unlocked, and control returned. Otherwise, the *int_elem* element of the *cbs* data member is set to the integer value appearing as the argument to *send*. Then the *have_reply* data member is set to register the fact that the

Cbox has received the reply message. Finally, any *thread* waiting for a reply message is unblocked. The first step in unblocking a *thread* is to invoke the *recall* method in the *Cbox_lock* object. This returns a pointer to the first *thread* object in the waiting queue of *Cbox_lock*. If no *thread* was waiting then the Cbox is unlocked and control returned. Otherwise, the *thread* is awakened by invoking the *wakeup* method in it before unlocking and returning. Note that there can be many threads waiting in the queue of *Cbox_lock* but only one is awakened per *send* operation.

The *send* method having two arguments is different from the rest of the overloadings in its implementation. It is used to send a user defined structure as a reply message. The user does not invoke this overloading directly. Instead the *SENDSTR* macro is used. The macro casts its first argument (which is a pointer to the user defined structure being passed) into a character pointer and applies the *sizeof* function to its second argument (which is the name of the user defined structure type) to obtain the size of the structure being passed. The difference in the implementation is in the following four lines which appear in place of *cbs.int_elem* = *i* that appeared in the above *send* method.

```
void Cbox_first::send(char* c, int i)
{
                  .
                  .
      cbs.st_elem = new struct_item;
      (cbs.st_elem)->struct_elem = malloc(i);
      (cbs.st_elem)->struct_len = i;
      bcopy((char*) c, (char*) ((cbs.st_elem)->struct_elem), i);
                  .
                  .
}
```

The first argument *c* is a pointer to the structure being passed in the reply message. The size of the structure is passed as the argument *i*. Enough memory to store the structure is obtained through a call to the *malloc* function. The length of the structure is also recorded in the *struct_len* item of *st_elem*. Then the block copy function *bcopy* is used to initialize the *struct_elem* item of *st_elem*.

The Cbox of the LAST type is implemented by the *Cbox_last* class. The *send* methods in this class are similar to the *send* methods in *Cbox_first* except that they handle the next reply message differently. Let us consider the *send* method which is used to send an integer as a reply message.

```
void Cbox_last::send(int& i)
{
      Cbox_lock->lock();
      if (have_reply)    {
            cbs.int_elem = i;
            Cbox_lock->unlock();
            return;
      }
      cbs.int_elem = i;
      have_reply = 1;
      Thread* t = Cbox_lock->recall();
      if(t!=0)
            t->wakeup(Cbox_lock);
      Cbox_lock->unlock();
      return;
}
```

The only difference of the above with *Cbox_first::send* that serves the same purpose is the way the reply message is handled after detecting that the Cbox is full. Since this is the LAST type Cbox in which the most current reply message is retained, the previous *int_elem* is overwritten by the new integer message.

The overloading for sending a structure as a reply message has the same differences as for the *Cbox_first* class except that if the Cbox already has a structure when a new reply message arrives, the old *st_elem* element in the *cbs* data member is deleted and a new structure item is assigned to it.

7.3 The Implementation of the QUEUE Cbox

The QUEUE type Cbox is implemented by the *Cbox_queue* class as defined below.

```
class Cbox_queue : public Cbox_sup  {
            Oqueue Cbox_csqueue;
            int num_items;
protected:
            virtual int get_int_elem();
            virtual char get_char_elem();
            virtual double get_double_elem();
            virtual struct_item* get_struct_elem();
            virtual Cbox* get_cbox_elem();
            virtual Actor* get_actor_elem();
            virtual void reset_have_reply();
public:
            Cbox_queue();
            void send(int&);
            void send(float&);
            void send(char&);
            void send(double&);
            void send(char*, int);
            void send(Cbox*);
            void send(Actor*);
            void flush();

}
```

The queue that stores reply messages is implemented as an instantiation of the *Oqueue* class of PRESTO. The objects that represent individual reply messages are instantiations of the *Cboxq_Item* class in ACT++ which has the following definition.

```
class Cboxq_Item : public Object    {
            union {
                  int int_elem;
                  char char_elem;
                  double double_elem;
                  struct_item* st_elem;
                  Cbox* cbox_elem;
                  Actor* actor_elem;
            }       cbqs;
public:
            Cboxq_Item()        {;};
            void setint(int);
            void setchar(char);
            void setdoub(double);
            void setstruct(char*, int);
            void setcbox(Cbox*);
            void setactor(Actor*);
            int getint();
            double getdouble();
            struct_item* getstruct();
            Cbox* getcbox();
            char getchr();
            Actor* getactor();
}
```

We will first discuss the *Cboxq_Item* class and then explain the *Cbox_queue* implementation.

The private data member *cbqs* of *Cboxq_item* is used to store the data value of a reply message and is similar in function to those in the *Cbox_sup* class. Note that although the *Cbox_queue* class is a subclass of *Cbox_sup*, the reply messages in a queue type Cbox do not use the data storage members of *Cbox_sup*. Instead, the members of *Cboxq_Item* are used for that purpose.

The remaining methods in *Cboxq_Item* fall in two categories - those for setting the reply message with the proper data item and those for extracting the data value from a reply message. The former is done using the *setX* set of methods and the latter is done using the *getX* set of methods. The *setint* and *getint* methods are shown below and all the remaining methods in each type are similarly defined.

```
void Cboxq_Item::setint(int i)
{
      cbqs.int_elem = i;
}
```

```
int Cboxq_Item::getint()
{
      return cbqs.int_elem;
}
```

The methods in the *Cbox_queue* class except for *reset_have_reply* and *flush* fall in two categories - those for sending a reply message to a queue type Cbox and those for extracting a reply message from a queue type Cbox. The former is done using the overloadings of the *send* method and the latter is done using the *get_X_elem* set of methods. As before, all the *send* methods are similar in structure except for a minor deviation in the overloading for sending a structure. As for the *get_X_elem* methods, they are all similar in structure. As a result, we will discuss the overloading for sending an integer as a reply message, how the overloading for sending a structure differs from the rest, and the *get_int_elem* method that is used to extract an integer reply message.

The integer overloading of *send* is defined as follows.

```
void Cbox_queue::send(int& i)
{
      Cbox_lock->lock();
      Cboxq_Item* curr_item = new Cboxq_Item();
      curr_item->setint(i);
      Cbox_csqueue.append((Object*) curr_item);
      have_reply = 1;
      num_items++;
      Thread* t = Cbox_lock->recall();
      if (t != 0)
            t->wakeup(Cbox_lock);
      Cbox_lock->unlock();
      return;
}
```

After locking the Cbox, an instantiation, *curr_item*, of the *Cboxq_Item* class is obtained which stores the reply message. Next, the *setint* method in *curr_item* is invoked to set the integer data member of the reply message. Then the reply message is appended to the reply message queue, *Cbox_csqueue* and *have_reply* is set to 1, indicating that the Cbox has reply messages in it. Then the *num_items* data member is incremented to account for the arrival of the new reply message. The rest of the code pertains to *thread* awakening and was explained earlier.

The overloading of *send* to pass a structure has additional code to realize structure copying as shown below.

```
void Cbox_queue::send(char* s, int i)
{
                        .
                        .
      Cboxq_Item* curr_item = new Cboxq_Item();
      char* st = malloc(i);
      bcopy((char*) s, (char*) st, i);
      curr_item->setstruct(st, i);
      Cbox_csqueue.append((Object*) curr_item);
                        .
                        .
}
```

As before, a new instantiation of the *Cboxq_Item* class is obtained to store the reply message. A block of memory, large enough to store the structure, is obtained through *malloc* and the *bcopy* function is invoked to copy the structure into the new memory area. Then the *setstruct* method is invoked in the new reply message item to set the data member storing the structure. Finally, the reply message is appended to the *Cbox_csqueue*.

The *get_int_elem* method is defined as follows.

```
int Cbox_queue::get_int_elem()
{
      Cboxq_Item* cbqi = (Cboxq_Item*) Cbox_csqueue.get();
      int j = cbqi->getint();
      delete cbqi;
      num_items--;
      if (num_items == 0)
            have_reply = 0;
      return j;
}
```

The first action is to extract the reply message at the front of the queue. Then the *getint* method is invoked in the extracted message to obtain the actual integer content of the reply message. Then the reply message is deleted, *num_items* is decremented to account for the extraction, *have_reply* is set to indicate whether any reply messages remain in the queue, and the integer value just extracted is returned.

The *reset_have_reply* method in the *Cbox_queue* class is a specialization of the same method in the *Cbox_sup* class. It is defined as a null method in *Cbox_queue*. The purpose of *reset_have_reply* in *Cbox_sup* is to set the *have_reply* data member so that it indicates that the Cbox is empty. But for queue type Cboxes the *have_reply* data member is set from the *get_X_elem* methods. Since *reset_have_reply* is invoked from the *receive* methods in *Cbox_sup* which are used to extract reply messages for queue type Cboxes too, it is defined to do nothing for queue type Cboxes. This specialization lets us use the same set of *receive* methods for the three variations of Cboxes.

As discussed before, the *flush* method is applicable to queue type Cboxes only. It can be used to delete all reply messages in the Cbox at any point in time. The method is useful in the case when all reply messages existing in the Cbox at a certain point in the computation must be ignored and must be removed from the Cbox before any new reply message arrives. The method is defined as follows.

```
void Cbox_queue::flush()
{
      int i;
      Cboxq_Item* cbqi;
      have_reply = 0;
      for (i = 1; i < num_items; i++)     {
            cbqi = (Cboxq_Item*) Cbox_csqueue.get();
            delete cbqi;
```

```
        }
        num_items = 0;
}
```

The above method extracts a reply message and deletes it. This is done for all the reply messages in the queue. It also sets *have_reply* and *num_items* to zero indicating that the Cbox is empty.

8. Handling I/O in ACT++

The issue of I/O in a concurrent object oriented system has been discussed at length in chapter 2. In this section we will discuss the classes that realize I/O in ACT++. We begin by assuming that there is only one processor in the system and we will discuss the facilities available to do I/O in such a system. Then we will extend our discussion to a multi-processor system, highlight the problems that are introduced, and explain how we have tackled them in our implementation.

Before discussing the major classes that handle I/O we will consider some of the classes that play a supporting role in the implementation in the following section.

8.1 Wboxes, Rboxes, I/O Queues and Replypoints

Wboxes and Rboxes are the media through which data to be written and read is transferred respectively. Wboxes are implemented by the *Wbox* class and Rboxes are implemented by the *Rbox* class. Both Wboxes and Rboxes are special types of Cboxes and are derived from the *Cbox_sup* class. Some methods pertaining to the Rbox and Wbox classes have already been discussed under the discussion on *Cbox_sup*. The class definitions follow.

```
class Wbox : public Cbox_sup  {
            char* buffer;
            int buf_size;
public:
            Wbox();
            Wbox(char*);
            Wbox(Rbox*);
            void refill(char*);
            void refill(Rbox*);
            int size();
            void refresh();
            char* ret_buff();
            void wait()
            { Cbox_sup::wait();}
}


class Rbox : public Cbox_sup  {
            char* buffer;
            int buf_size;
public:
            Rbox();
            Rbox(int);
            void refresh();
            void refresh(int);
            int size();
            char* ret_buff();
            char get(int);
            void wait()
            { Cbox_sup::wait();}
}
```

The private data members of both the above classes are the same. There is a *buffer* data member that stores the characters that are read or are meant to be written. The *buf_size* data member stores the length of the character buffer.

The *Wbox()* constructor sets both *buffer* and *buf_size* to zeros. The *Wbox(char*)* constructor is used to construct a new Wbox and copy the null-terminated string of characters passed as an argument to the constructor into *buffer*. Its definition follows.

```
Wbox::Wbox(char* buff)
{
      int i = strlen(buff);
      buffer = new char [i + 1];
```

```
        buf_size = i + 1; //Add 1 for the null-terminator
        bcopy(buff, buffer, buf_size);
}
```

The length of the string passed as an argument is determined first in the above constructor and stored in *i*. Then *buffer* is assigned a new string of characters of size *(i + 1)*.Then the length is recorded in *buf_size* and the string in the argument is copied into *buffer* using the *bcopy* function.

The *Wbox(Rbox*)* constructor is used to construct a Wbox from a Rbox. Such a constructor is useful in the case when information read from one place needs to be displayed at some other place without modifying it in any way. In order to do so, the Wbox constructor is given a pointer to the Rbox which contains the information to be displayed and the Wbox is constructed by extracting the information from the Rbox. The constructor is defined as follows.

```
Wbox::Wbox(Rbox* rb)
{
        buf_size = rb->size();
        buffer = new char [buf_size];
        bcopy(rb->ret_buff(), buffer, buf_size);
}
```

In this case, the length of the buffer is obtained from the argument *rb* by invoking the *size* method in it. Then a new blank *buffer* is created and the data buffer in *rb* is copied into *buffer*. The *ret_buff* method is invoked in *rb* to obtain a pointer to its *buffer* data member.

The overloaded *refill* methods in the Wbox class are used to reinitialize an existing Wbox with new data. The new data is either supplied as a null-terminated string of characters or it is extracted from a Rbox. The definition of *refill(char*)* follows.

```
void Wbox::refill(char* buff)
{
      delete buffer;
      have_reply = 0;
      buf_size = strlen(buff) + 1;  //Add 1 for the null_terminator
      buffer = new char [buf_size];
      bcopy(buff, buffer, buf_size);
}
```

The existing *buffer* is deleted and *have_reply* is set to zero indicating that the contents of the Wbox has not yet been written. Then *buf_size* is determined from the string argument *buff* and a new *buffer* of the appropriate size is created. Then *buffer* is initialized from *buff.*

At this point let us digress to consider the role the *have_reply* data member plays in determining the blocking criterion on a Wbox. Recall that *have_reply* is a private data member in the *Cbox_sup* class which indicates whether a Cbox has a reply message. If *have_reply* is zero then there is no reply message in the Cbox and a *thread* blocks and vice versa. The blocking semantics for Wboxes is different. A *thread* blocks on a Wbox if its content has not yet been written and vice versa. Hence, in order to use the same methods for implementing blocking for both Cboxes and Wboxes, *have_reply* is set to zero when a Wbox is constructed or refilled indicating that the contents of the box has not yet been written and a *thread* must block on it. When the content is written, *have_reply* is set to 1 from the *wakeup_thread* method defined in *Cbox_sup* indicating that the contents have been written and a blocked *thread* can be awakened.

The differences between the above *refill* method and *refill(Rbox*)* are that the *buf_size* is set by invoking the *size* method in the Rbox and the data to be copied into *buffer* is obtained by invoking the *ret_buff* method in the Rbox.

Among the remaining methods in the *Wbox* class, *size* returns the *buf_size* data member, *ret_buff* returns the *buffer* data member, and *refresh* deletes *buffer* and sets *have_reply* and *buffer* to zero. Note that one of the *refill* methods must be used immediately after using the *refresh* method in order to fill the Wbox with data.

The *Rbox()* constructor in the *Rbox* class sets both *buffer* and *buf_size* to zero. The *Rbox(int)* constructor is defined as follows.

```
Rbox::Rbox(int size)
{
      buffer = new char [size];
      buf_size = size;
      bzero(buffer, buf_size);
}
```

The integer argument to the constructor specifies the size of the character buffer that is to be created in the Rbox. Accordingly, *buffer* is assigned a new character array of the requested size and *buf_size* records that size. Note that I/O using Wboxes and Rboxes handle only null-terminated strings and hence the *size* argument must include space for the null terminator. The *bzero* function assigns null characters to the buffer thereby ensuring that any string of characers that are read in will be null terminated. Choosing *size* to be at least one character larger than the maximum number of characters that could be read ensures that the string is always null-terminated.

The two overloadings of the *refresh* method are used to reinitialize an existing Rbox. The two are defined as follows.

```
void Rbox::refresh()
{
      delete buffer;
      have_reply = 0;
      buffer = new char [buf_size];
      bzero(buffer, buf_size);
}

void Rbox::refresh(int new_size)
{
      delete buffer;
      have_reply = 0;
      buf_size = new_size;
      buffer = new char [new_size];
      bzero(buffer, buf_size);
}
```

In the first *refresh* method, first, the old *buffer* is deleted and *have_reply* set to zero indicating that the Rbox is empty. Then a new *buffer* is created which is of the same size as the previous *buffer* (because the existing *buf_size* value is used). The second overloading of *refresh* differs only in the way the size of the new *buffer* is determined - the integer argument to the method is treated as the new size and a *buffer* of that size is created.

Note that unlike the blocking semantics on Wboxes, Rboxes use the same blocking semantics as Cboxes. Hence, when a Rbox is constructed or refreshed, *have_reply* is set to zero indicating that the the Rbox is empty and a *thread* must block on it. When data is read into the Rbox, *have_reply* is set to 1 from the *wakeup_thread* method defined in *Cbox_sup* indicating that the Rbox is full and any blocked *thread* must be awakened.

Among the other methods in the *Rbox* class, *size* and *ret_buff* are exactly the same as in the *Wbox* class. The *get* method is used for extracting individual characters from the Rbox and is defined as follows.

```
char Rbox::get(int i)
{
      if (i < buf_size)
            return buffer[i];
      else
            return NULL;
}
```

The *get* method indexes into the *buffer* data member in the Rbox and returns the character corresponding to the integer value supplied in the argument. If the index falls outside the buffer size then a null character is returned.

Instantiations of both the *Io_queue* and *Replypoint* classes are used in connection with performing I/O in ACT++. An *Io_queue* object is a queue that stores *replypoints* which are instantiations of the *Replypoint* class. Since multiple file descriptors might be waiting for I/O to become ready, such file descriptors are remembered by the ACT++ run time system by enqueueing them in a I/O queue. A *replypoint* is a collection of a file descriptor and a Cbox. When I/O is ready on the file descriptor, the *replypoint* containing the file descriptor is extracted from the I/O queue and a reply message is sent to the corresponding Cbox. On receipt of this reply message the *thread* waiting for the completion of I/O is awakened. The definitions of the *Replypoint* and *Io_queue* classes follow.

```
struct Replypoint : public Object    {
private:
            Cbox* dest;
            int fd;
public:
            Replypoint(Cbox*, int);
            int get_fd();
            Cbox* get_dest();
}

class Io_queue : public Mailq {
public:
            Io_queue();
            int lookat_next();
}
```

The constructor of the *Replypoint* class sets the *dest* and *fd* data members to the first and second arguments respectively. The *get_fd* method returns the *fd* data member and the *get_dest* method returns the *dest* data member.

The *Io_queue* class has an empty constructor. The operations defined in the *Mailq* class for performing linear searches on a queue and extracting elements from anywhere in a queue are utilized by the *Io_queue* class. The only method that had to be specialized is *lookat_next.* The *Mailq* class implements the message queue in an actor and hence the item returned by *Mailq::lookat_next* is the method id inside the next message object in the message queue. Since *Io_queue* implements a queue of *replypoint* objects, the *lookat_next* method has to return the file descriptor inside the next *replypoint* object in the I/O queue. As a result, the method has been specialized as follows.

```
int Io_queue::lookat_next()
{
      if (curr_pos != 0)      {
            Replypoint* temp_ptr = (Replypoint*) curr_pos;
            int i = temp_ptr->get_fd();
            return i;
      }
      return -1;
}
```

If all the elements in the queue have not been looked at (indicated by *curr_pos != 0*), then the *curr_pos* pointer (which points to the next element to be examined in the queue) is cast to point to a *Replypoint* object and the transformed pointer is stored in *temp_ptr.* Then the *get_fd* method is invoked in *temp_ptr* to obtain the file descriptor inside the *replypoint.* Then this value is returned. If all the elements in the queue have been examined (indicated by *curr_pos = 0*), -1 is returned indicating that the search is over.

The casting of *curr_pos* to a *Replypoint* pointer is necessary to avoid declaring a virtual *get_fd* method in the *Object* class in PRESTO. Since *Io_queue* is a subclass of the *Oqueue* class which implements queues that store pointers to instantiations of the *Object* class, *curr_pos* actually is an *Object* pointer. Invoking *get_fd* directly in *curr_pos* would require us to declare a virtual *get_fd* method in the *Object* class. Since our goal was to minimize the changes made to PRESTO in order to implement ACT++, we avoided that option and did the casting instead.

8.2 I/O In a Single Processor System

I/O in ACT++ has been implemented as an actor operation. This means that instead of issuing I/O requests directly to files one must direct such a request to an actor that is responsible for that file. Actors which handle I/O to files are called *interface actors* (IAs)and the *IActor* class in ACT++ implements such actors. The definition of the class follows.

```
class IActor : public Actor   {
public:
            IActor(char* fname, Tty_Beh* init_beh, char* name = 0);
            ~IActor();
}
```

The behaviors of an IA are instantiations of the *Tty_Beh* class which is defined as follows.

```
class Tty_Beh : public Behavior      {
            int fd;
            int oldflags;
public:
            Tty_Beh();
            int Open(char* fname);
            void Read(Rbox* r, int nbytes);
            void Write(Wbox* w, int nbytes);
}
```

To use an IA, the user has to create one using the *IActor* constructor and assign an initial behavior object to it by using the TTYBEH macro. As a result of using the TTYBEH macro, an instantiation of the *Tty_Beh* class is created using the *Tty_Beh* constructor that does not have any arguments. The only activities in *Tty_Beh::Tty_Beh()* are to set the *fd* data member to -1 and the *oldflags* data member to 0. The *fd* data member stores the file descriptor of the device for which the IA is the server. Since file descriptors start from 0, a -1 indicates that no valid file descriptor is stored in *fd*. The *oldflags* data member is used to store the old status of a file descriptor.

The definition of the *IActor* constructor is as follows.

```
IActor::IActor(char* fname, Tty_Beh* init_beh, char* name)
                    : ((Behavior*) init_beh, 1, name)
{
        int fd = init_beh->Open(fname);
}
```

The above is an incomplete definition of the *IActor* constructor. As we introduce different issues related to doing I/O in ACT++, we will embellish the above definition.

The first argument to *IActor* is the name of the file for which the IA will act as a server. The second argument is a pointer to the initial behavior object. The third argument could be used to assign a name to the IA. The assignment is optional because a null name is assigned by default.

The initializer list associated with the *IActor* constructor is used to initialize the arguments of the *Actor* constructor. The first argument in the list specifies the initial behavior object and the third argument specifies the optional name. The second argument specifies the number of *threads* that can be simultaneously active inside an IA. There should be only one *thread* executing inside an IA at any point in time. Hence this argument has a default value of 1. The significance of this arrangement is that an IA will serialize all I/O requests to a file because it will never process the next request message until the current request message has been serviced completely. This ensures that multiple I/O requests are not activated simultaneously on the same file descriptor.

The only activity in the *IActor* constructor is to invoke the *Open* method in the *init_beh* object which is an instantiation of the *Tty_Beh* class.This way of constructing an "interface actor-behavior" pair is different from the construction process of an ordinary "actor-behavior" pair. In the latter case, a behavior object is constructed first and then it is passed as an argument to the actor constructor. The entire behavior object is constructed and ready to be used before it is passed to the actor constructor. But in the former case, the behavior object constructed as a result of using the TTYBEH macro is an incomplete object and not ready for use. It is the invocation of the *Open* method from the *IActor* constructor that completes the definition of the initial behavior object and renders it useful.

The reason behind this deviation is related to doing multiprocessor I/O in ACT++. Although multiprocessor I/O is discussed in a later section, we will discuss the rationale behind the "interface actor-behavior" construction now. The construction of an initial *Tty_Beh* object and the construction of an IA are two independent activities that are executed by the same physical *thread*. On a single processor system, the *thread* is guaranteed to execute on the same process object. But, on a multiprocessor that is running a preemptive ACT++ run time system, the *thread* might execute on different process objects before it can complete both the construction activities. Thus if the functionality of the *Open* method, which includes obtaining a file descriptor that the IA will use for I/O, was done in the constructor of the *Tty_Beh* class, then, the file descriptor could be obtained on process object A and the IA could be constructed on process object B. The underlying assumption by the ACT++ run time system is that all *threads* executing the request messages to an IA are executed on the process object on which the IA was constructed. As a result, the construction of the *Tty_Beh* object is completed as a part of the IA construction process.

The *Open* method is defined as follows.

```
int Tty_Beh::Open(char* fname)
{
      static struct sigvec svec_async = {sigasyncio_alrm,
                              sigmask(SIGIO), 1};
      int fcntl_arg = 000;
      fd = open(fname, O_RDWR, 0);
      if (sigvec(SIGIO, &svec_async, (struct sigvec*) 0) < 0)
      {
            perror("\n sigvecsigio, in Tty_Beh::Open \n");
            fatalerror();
      }
      if ((oldflags = fcntl(fd, F_GETFL, 0) < 0)      {
            perror("\n fcntl FGETFL setup error in
                        Tty_Beh::Open \n");
            fatalerror();
      }
      thisaiom->register_fd_stat(fd, oldflags);
      fcntl_arg = fcntl_arg | (FASYNC | FNDELAY);
      if (fcntl(fd, F_SETFL, fcntl_arg) < 0)    {
            perror("\n fcntl FASYNC setup error in
                        Tty_Beh::Open \n");
            fatalerror();
      }
      if (fcntl(fd, F_SETOWN, getpid()) < 0)    {
            perror("\n fcntl FSETOWN setup error in
                        Tty_Beh::Open \n");
            fatalerror();
      }
      return fd;
}
```

The first important activity in *Open* is to invoke the *open* system call and obtain the system provided file descriptor corresponding to the file name argument *fname* supplied by the user. This is an important step because all subsequent I/O to the file will use this file descriptor and not the name of the file.

The next important activity in *Open* is to specify the signal handler for the SIGIO signal. This is the signal that the operating system will send when I/O is ready i.e. when the I/O operation on the file can be completed without blocking. The specification is done using the *sigvec* system call. The first argument to *sigvec* specifies the signal for which the signal handler is being specified - in our case that is SIGIO. The second argument to *sigvec* specifies a reference to a structure that contains, among other things, the name of the signal handler function. In our case, the signal handler function is called *sigasynio_alrm* .

After setting up the signal handler, the next activity in *Open* is to obtain the current status flags associated with the file descriptor *fd*. This is done to assure proper restoration of the status of the file descriptor at the end of the ACT++ application. The current status is obtained by using the F_GETFL option of the *fcntl* system call and this status is recorded in the *oldflags* data member.

The significance of the invocation of the *register_fd_stat* method in *thisaiom* will be discussed in section 8.3.4. The next activity in *Open* is to change the status of the file descriptor *fd* to enable both asynchronous I/O and I/O without delay on it. That is done by first setting the *fcntl_arg* variable to the disjunction of the FASYNC and FNDELAY flags. Setting the FASYNC flag denotes that asynchronous I/O can be done on the device. Setting the FNDELAY flag denotes that a *read* or a *write* system call will never wait for the file to become ready for I/O when the calls are made. With the FNDELAY flag set, if a I/O call returns with a zero return value, then, it indicates that the file is not ready for I/O and asynchronous I/O must be performed. To set the new status, the *fcntl* system call is invoked using the F_SETFL option and providing the *fcntl_arg* variable as an argument.

The next step in *Open* is to identify the current process as the recipient of the SIGIO signal. This is an important step because it informs the operating system which process to deliver the SIGIO signal to for a file on which asynchronous I/O is being performed. The *fcntl* system call is used once again, this time with the F_SETOWN option. The process id of the current process is obtained using the *getpid* system call and it is passed as an argument to *fcntl*.

The final action in *Open* is to return the file descriptor *fd*. Note that *perror* prints a message on the terminal and is used in the above method if any one of the system calls fails. The *fatalerror* function is a predefined function in PRESTO that gracefully terminates the PRESTO application.

8.2.1 Reading and Writing Using an Interface Actor

To read from and write to a file for which an IA has been created, one has to send read and write request messages respectively. The read request message invokes the *Read* method and the write request message invokes the *Write* method in the current behavior of the IA. These methods are defined as follows.

```
void Tty_Beh::Read(Rbox* r, int nbytes)
{
        int ret_val = 0;
        Cbox* cb = new Cbox(FIRST);
        int async_io = thisaiom->init_async_io(cb, fd,
                            (Cbox_sup*) r, nbytes, READ);
        if (async_io)       {
                cb->receive(ret_val);
                read(fd, r->ret_buff(), nbytes);
        };
        r->wakeup_thread();
        delete cb;
        become(THISBEH);
}
```

```
void Tty_Beh::Write(Wbox* w, int nbytes)
{
      int ret_val = 0;
      Cbox* cb = new Cbox(FIRST);
      int async_io = thisaiom->init_async_io(cb, fd,
                        (Cbox_sup*) w, nbytes, WRITE);
      if (async_io)     {
            cb->receive(ret_val);
            write(fd, w->ret_buff(), nbytes);
      };
      r->wakeup_thread();
      delete cb;
      become(THISBEH);
}
```

An object with which an IA interacts in order to perform I/O is the asynchronous I/O manager (AIOM). There is only one AIOM object per process object which manages the low level aspects of doing I/O, especially asynchronous I/O. The AIOM is the central manager of I/O involving all the file descriptors associated with IAs. It keeps track of all the file descriptors that have been used by the IAs, determines which file descriptor is ready for I/O when a SIGIO signal occurs, and acts as an intermediary through which the operating system and the ACT++ application communicate in order to do I/O. The interaction between the current behavior of an IA and the AIOM is captured in **Figures 13(a) and 13(b)** which depict the cases of synchronous and asynchronous I/O respectively. In this section we will explain some aspects of the AIOM. A more complete explanation is given in the next section.

We consider the operations of the *Read* method in Figure 13. The operations of the *Write* method are similar and will not be considered separately.

The first argument passed to the *Read* method is a pointer to a Rbox object in which the read data is to be stored. The second argument represents the number of bytes that is to be read from the device.

In the *Read* method, first, a Cbox is created. This is the Cbox on which the current *thread* might block if the read request cannot be satisfied synchronously. After that, the *init_async_io* method in *thisaiom* is invoked. *Thisaiom* is a private pointer that is maintained by a process object to keep track of the AIOM with which it is associated.

The first argument to *init_async_io* is the pointer to the Cbox created in *Read.* The AIOM sends a reply message to the Cbox when I/O is ready on the device in the case of asynchronous I/O. The second argument is the file descriptor on which the I/O is requested. The third argument is the Rbox pointer in which the data is to be recorded. The fourth argument specifies the number of bytes to be read. The fifth argument specifies that this invocation of *init_async_io* is associated with a read operation.

The reason behind sending all the information to *init_async_io* is that if the device is found to be ready, then a synchronous read operation is performed in *init_async_io*. This case is shown in **Figure 13(a)**. To do the synchronous read operation the file descriptor, the Rbox pointer, and the number of bytes is required. The Cbox is not used in this case because the *thread* executing the *Read* method never blocks. After the Rbox has been filled, a zero is returned to *Read* indicating that the I/O could be completed synchronously. Noting that, the next step in *Read* is to wakeup any *thread* waiting in the Rbox by invoking the *wakeup_thread* method in the Rbox. Note that it is assumed that there can be at most one *thread* waiting for the Rbox to be filled. The assumption has an important impact on the way Rboxes are to be used because if more than one *thread* is waiting inside the Rbox, then only the first one will be awakened and the rest will never be awakened. Therefore it is mandatory to allow only one *thread* to block on a Rbox at any point in time.

If the I/O cannot be completed synchronously in *init_async_io* then the file descriptor is prepared for asynchronous I/O. The operations involved in this case is shown in **Figure 13(b)**. A 1 is returned to *Read* by *init_async_io* indicating that asynchronous I/O has to be performed. This causes the *thread* executing the *Read* to invoke the *receive* operation (step 4 in Fig 13(b)) on the Cbox that was created before. The *thread* blocks on the Cbox awaiting a reply message from the AIOM.

When the device is ready for I/O, a SIGIO signal is sent (step 5) by the system which in turn invokes the SIGIO signal handler. The signal handler invokes the *propagate_signal* method (step 6) in the AIOM. After doing some file descriptor related operations, *propagate_signal* sends a reply message to the Cbox (step 7) on which the *thread* executing the *Read* method blocked. This awakens the *thread* (step 8) which then executes the *read* system call to read the data from the device (step 9) into the Rbox. Then the *wakeup* method is invoked in the Rbox to awaken any waiting *thread*.

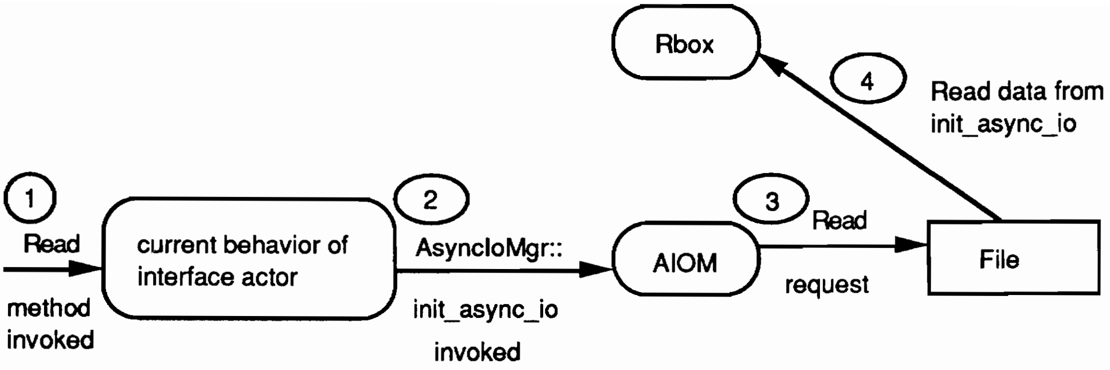


Figure 13(a) : The steps involved in a synchronous read operation initiated from the Tty_Beh::Read method. The numbers in circles denote the step number. Note that this sequence of events does not block the thread executing the Read method.

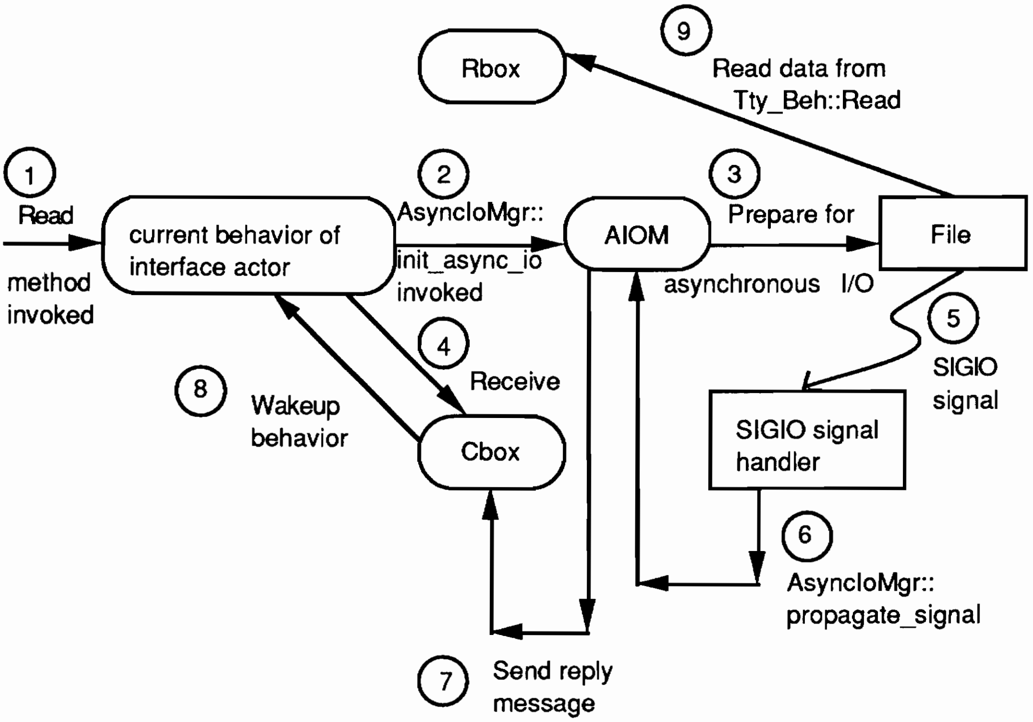


Figure 13(b) : The steps involved in an asynchronous read operation initiated from the Tty_Beh::Read method. The numbers in circles denote the step number. Note that the thread executing Tty_Beh::Read blocks on a Cbox until awakened by the propagate_signal method of the AIOM. On waking up, the thread does a read from Tty_Beh::Read into the Rbox.

8.2.2 The Operation of the AIOM

As discussed in the previous section, the AIOM plays a major role in realizing I/O operations in ACT++. In this section we will discuss the role of the AIOM in detail. AIOM objects are instantiations of the *AsyncIoMgr* class in ACT++ which is defined as follows.

```
class AsyncIoMgr  {
            int fdtablesize;
            fd_set io_mask;
            Io_queue* io_pend_list;
public:
            AsyncIoMgr();
            int init_async_io(Cbox*, int, Cbox_sup*, int, int);
            void propagate_signal();
}
```

The *fdtablesize* data member records the size of the file descriptor table. This member is used when the set of file descriptors have to be searched to find out which ones are ready for I/O. The *io_mask* data member records the file descriptors that have been set up for asynchronous I/O. *Fd_set* is a system defined type that is used to declare bit masks corresponding to the bit masks used by the system to manipulate file descriptors. There are predefined macros available that operate on *fd_set* variables to set specific positions in the bit mask, to determine whether specific bits are set etc.

The *io_pend_list* data member is a pointer to an instantiation of the *Io_queue* class. The queue stores instantiations of the *Replypoint* class. Recall that a *replypoint* object stores a file descriptor and a Cbox pointer. When a particular file descriptor which was set up for asynchronous I/O is found to be ready for I/O, the AIOM searches for that descriptor in the *io_pend_list* queue, and if found, sends a reply message to the associated Cbox. If the file descriptor is not found in the queue then it indicates that I/O has not yet been requested on the file descriptor. As a result, when I/O is requested on that file descriptor at a later time, it will be serviced synchronously.

The constructor for the class is defined as follows.

```
AsyncIoMgr::AsyncIoMgr()
{
      fdtablesize = getdtablesize();
      FD_ZERO(&io_mask);
      io_pend_list = new Io_queue();
}
```

The *getdtablesize* is a system call that returns the size of the file descriptor table. *FD_ZERO* is a system defined macro that sets all the bits of *io_mask* to zero. Finally, *io_pend_list* is assigned a new I/O queue.

The *init_async_io* method is defined as follows.

```
int AsyncIoMgr::init_async_io(Cbox* cb, int fd_arg,
            Cbox_sup* r, int nbytes, int io_type)
{
      int oldmask;
      thisthread->nonpreemptable();
      oldmask = sigblock(sigmask(SIGIO));
      if ((iotype == READ) && (read(fd_arg, r->ret_buff(),
                                             nbytes) > 0))
      {
            sigsetmask(oldmask);
            thisthread->preemptable();
            return 0;
      }
      if ((iotype == WRITE) && (write(fd_arg, r->ret_buff(),
                                             nbytes) > 0))
      {
            sigsetmask(oldmask);
            thisthread->preemptable();
            return 0;
      }
      FD_SET(fd_arg, &io_mask);
      Replypoint* rp = new Replypoint(cb, fd_arg);
      io_pend_list->append((Object*) rp);
      sigsetmask(oldmask);
      thisthread->preemptable();
```

```
        return 1;
}
```

Init_async_io is very sensitive to interruptions. There are two ways in which the execution of *init_async_io* can be interrupted:

- the scheduler can preempt the *thread* executing *init_async_io* (if PRESTO has been configured to run in a preemptive fashion), and

- the SIGIO signal handler can start executing due to a SIGIO signal from the operating system.

There are three reasons for not interrupting *init_async_io*. First, if interrupted before the synchronous *read* and *write* calls could be made then, although the file may be ready, the I/O will be unnecessarily delayed. Second, if the file is not ready then the file descriptor should be enqueued in the *io_pending_list* queue as soon as possible to prevent unnecessary delay in sending a reply message.

The third and the most important reason is the possibility of deadlock. The *thread* executing *init_async_io* can be interrupted by the execution of the SIGIO signal handler at any point in time. If this interruption occurs after the *read* or *write* system call is over but before the *replypoint* object is enqueued in *io_pend_list,* then deadlock will be imminent. This is because, after the I/O calls are executed for a file descriptor, I/O must be initiated by the SIGIO signal for that descriptor. Since the "file descriptor - Cbox pointer" pair could not be enqueued when the *thread* was interrupted, the signal handler will not be able to send a reply message to the Cbox. Deadlock may be avoided because after the signal handler executes, the *thread* executing *init_async_io* will be resumed and the pair will be enqueued. If the signal handler is invoked subsequently due to some other I/O event, then the I/O queue will be searched again and a reply message will be sent to the Cbox corresponding to the file descriptor that was skipped last time. But the possibility of deadlock remains.

To prevent the interruption of *init_async_io*, we used the *thisthread->nonpreemptable()* call at the very beginning. This ensures that the *thread* will not be preempted by the scheduler. *Thisthread* is a pointer to a *thread* that is maintained by each process object in PRESTO to remember the current *thread* that is executing on the process object. *Nonpreemptable* is a method in the *Thread* class that marks a *thread* as nonpreemptable.

To address the deadlock issue we introduced the call to the *sigblock* function. *Sigblock* is a system call that blocks the signal which appears in its argument. What is actually sent as an argument to *sigblock* is a mask generated by the *sigmask* system call. *Sigmask* declares a bit mask and sets in it the bit corresponding to the signal that appears in its argument. Blocking a signal means that if the signal occurs during the period in which it is blocked, then it is held by the system and delivered to the process when the signal is enabled once again. The call to *sigblock* returns the signal mask that existed before the *sigblock* executed. This old status can be used to restore the state of the signal masks (thereby enabling the signals that were blocked by *sigblock*). The old status is remembered in the *oldmask* variable.

The next action in *init_async_io* is to execute the code relevant to the read operation. If the current invocation of *init_async_io* was from *Tty_Beh::Read* then *io_type* will match the value of the READ macro. If the *read* call returns more than zero bytes, then the file is ready and the read request will be satisfied. Note that the second argument to *read* is the buffer obtained by invoking the *ret_buff* method in the Rbox. As a result, after the *read* is over, the Rbox will contain the read data. After completing a successful synchronous read operation, the signal mask is restored to its original state, the current *thread* marked as preemptable, and a zero is returned indicating that the I/O could be completed synchronously. The same explanation applies for the write operation.

For either the read or the write operation, if the *read* or the *write* returns zero bytes then the file is not ready and preparation for asynchronous I/O has to be made. Note that if the file is not ready, the I/O system call returns immediately because the FNDELAY option of the file descriptor was set in the *Tty_Beh::Open* method.

The first step in preparing for asynchronous I/O is to set the bit corresponding to the current file descriptor in the the *io_mask* data member. This is done using the system defined *FD_SET* macro. Then a *replypoint* object is created using the current file descriptor and the Cbox pointer to which a reply message is to be sent when the file is ready. Then the *replypoint* is enqueued in the *io_pend_list* queue. After that, the system signal mask is restored to its original state, the current *thread* is marked as preemptable, and a 1 is returned indicating that the I/O will be performed asynchronously.

The *propagate_signal* method is the one that is invoked from the SIGIO signal handler. The signal handler relies on this method for the proper notification of all the behaviors that are waiting for a I/O request to complete. The SIGIO signal handler is called *sigasyncio_alrm* and is defined as follows.

```
int sigasyncio_alrm(int sig, int code,  struct sigcontext * scp)
```

```
{
      thisaiom->propagate_signal();
      return 0;
}
```

The *propagate_signal* method is defined as follows.

```
void AsyncIoMgr::propagate_signal()
{
      struct timeval temp;
      int remove_head = 0;
      int remove_curr = 0;
      fd_set tempr_mask, tempw_mask;
      int j = howmany(FD_SETSIZE, NFDBITS);
      for (int i = 0; i < j; i++)    {
            tempr_mask.fds_bits[i] = io_mask.fds_bits[i];
            tempw_mask.fds_bits[i] = io_mask.fds_bits[i];
      }
      temp.tv_sec = 0;
      temp.tv_usec = 0;
      int nfdbits = select(fdtablesize, &tempr_mask,
                  &tempw_mask, (fd_set*) 0, &temp);
      if (nfdbits == 0)
            return;
      Replypoint* rp = (Replypoint*) io_pend_list->lookat();
      int fd = rp->get_fd();
      if (FD_ISSET(fd, &tempr_mask) || FD_ISSET(fd,
                                                &tempw_mask))
      {
            Cbox* cb = rp->get_dest();
            cb->send(1);
            FD_CLR(fd, &io_mask);
            remove_head = 1;
      }
      io_pend_list->reset();
      fd = io_pend_list->lookat_next();
      while (fd > -1)    {
            if (FD_ISSET(fd, &tempr_mask) || FD_ISSET(fd,
                                                &tempw_mask))
            {
                  rp = (Replypoint*) io_pend_list->
                                          rem_curr();
                  Cbox* cb = rp->get_dest();
                  cb->send(1);
                  FD_CLR(fd, &io_mask);
                  removed_curr = 1;
                  delete rp;
            }
            if (removed_curr)
                  removed_curr = 0;
            else
                  io_pend_list->shift_pos();
            fd = io_pend_list->lookat_next();
```

```
        }
        if (remove_head)  {
               rp = (Replypoint*) io_pend_list->get();
               delete rp;
        }
}
```

The first major event in *propagate_signal* is to prepare to make the *select* system call. The first step is to determine the size of the bit masks that are used to manipulate the file descriptors. The size varies depending on the number of file descriptors supported by the system on each process. The number is determined by using a system defined macro called *howmany*. Then the contents of the *io_mask* data member is copied into two temporary bit masks of type *fd_set* called *tempr_mask* and *tempw_mask*. Then the elements of the *temp* structure are initialized to zero.

The *select* system call is used to determine which file descriptors are ready for I/O. The first argument specifies the number of file descriptors that will be checked in the bit masks that are supplied as the second and third arguments. In our case we intend to search all the file descriptors i.e. all the bits in the bit masks. *Select* examines the file descriptors specified by these bit masks to see if they are ready for I/O and returns, in place, a mask of those descriptors which are ready. This implies that the mask corresponding to the second argument will be set for all the file descriptors that are ready for reading, and, the mask corresponding to the third argument will be set for all the file descriptors that are ready for writing. Note that since an IA serializes I/O requests, no file descriptor can be ready for both reading and writing and so the same bit should not be set in both the returned masks. We are not interested in the fourth argument which is used to determine which descriptors have an exceptional condition pending. The fifth argument enables the setting of a timeout limit for which *select* would poll the file descriptors. If the fifth argument points to a zero valued *timeval* structure then the call returns as soon as all the descriptors have been polled for once; that is the option used here.

Select returns the number of file descriptors that are ready for I/O. If no descriptors are ready then control is returned immediately. Otherwise the *io_pend_list* queue is scanned to determine which file descriptors waiting in the queue are ready for I/O. The search process starts by examining (without removing from the queue), the file descriptor in the first *replypoint*. Using the *FD_ISSET* macro it is determined whether the bit corresponding to this file descriptor is set in either of *tempr_mask* or *tempw_mask*. If so, the file is ready for I/O. Therefore, the Cbox pointer is extracted from the *replypoint* and a reply message is sent to the Cbox. Then the bit corresponding to the descriptor is cleared in the *io_mask* indicating that the request has been serviced. Due to the way the queue searching mechanism is implemented in ACT++, the head of a queue cannot be removed until the whole queue has been searched. We remember that the head has to be removed later by setting the *remove_head* variable.

To search the remainder of the *io_pend_list*, we invoke the *reset* method in it to initialize the *curr_pos* and *last_pos* pointers of the *Genqueue* class. Then we invoke the *lookat_next* method in *io_pend_list* which returns the file descriptor in the next *replypoint*. We determine whether it is ready for I/O in the same way as described above. The only difference in this case is that, if it is ready, the *replypoint* is extracted from *io_pend_list*. Removal of an item from the middle of a queue sets *curr_pos* to the next item to be examined. As a result, the *shift_pos* method need not be invoked in *io_pend_list* in order to shift the pointers. This is implemented by setting the *removed_curr* variable to 1 when an item is removed from the middle and invoking *shift_pos* only if *removed_curr* is zero. The search continues until a -1 is returned by the *lookat_next* method which indicates all members of the queue have been examined.

Finally, if *remove_head* was set before, the head of the *io_pend_list* is removed by invoking the *get* method in it.

8.3 I/O In a Multi-Processor System

The most important issue to be addressed to implement I/O in a multi-processor system is that of *file descriptor consistency* across processes. In the single processor case, only one process object is created. As a result, a file descriptor obtained in the *Tty_Beh::Open* method could be used by any *thread* executing the application because all the *threads* execute within the single process. But, in a multi-processor system, *threads* created on one process object may execute on other process objects during its lifetime. However, UNIX file descriptors are process specific entities which are undefined across process boundaries. As a result, the file descriptor obtained by the *thread* when it executed *Tty_Beh::Open* on one process object would be meaningless when the same *thread* runs on a different process object.

The solution to the file descriptor consistency problem is to selectively bind *threads* to process objects. The specific types of *threads* that are bound to process objects are those involved in the execution of methods in the behaviors of an IA. The *thread* that executes the constructor of an IA obtains a file descriptor. This file descriptor is stored in the initial behavior object, an instantiation of the *Tty_Beh* class, of the IA. Later, when the IA processes messages, *threads* are created which use the stored file descriptor to do I/O.

These *threads* have to be executed on the same process object on which the IA was created for the descriptors to be valid.

Since PRESTO does not support selective binding of *threads* to process objects we have implemented that feature in PRESTO. In order to do so, we had to modify the *Thread* class and subclass the *Scheduler* class in PRESTO. In the following, we will first discuss the operation of the scheduler in PRESTO and then discuss the enhancements made to PRESTO to implement selective *thread* binding.

8.3.1 The Operation of the Scheduler in PRESTO

The scheduler in PRESTO handles *thread* scheduling. It is an instantiation of the *Scheduler* class. The following partial definition of the *Scheduler* class contains the items that will be discussed.

```
class Scheduler : public Object     {

protected:
            ThreadPool *sc_t_ready;
                                    //threads wanting to run
            Process* sc_p_procs[NUMPROCS];
                                    //workers live here
            int sc_p_numschedulers;
                                    //max# which can be active
            int sc_p_activeschedulers;
                                    //# which are active
            int sc_quantum;    // in ms., if 0, no preemption
            int sc_p_busybits;      //32 is the proc limit
            Spinlock* sc_lock;
            inline int busybits(int on);

public:
      Scheduler(int nschedulers, int quantum = DEFQUANTUM);
      virtual int invoke();
      virtual void resume(Thread* t);
                                    //put t back on ready queue
      virtual Thread* getreadythread();
      int readyqlen()
```

```
            {return sc_t_ready->size();}
}
```

There is a single queue in the scheduler object that stores *threads* that are ready for execution. This queue is represented by the *sc_t_ready* data member. The *sc_p_procs* data member stores a pointer to all the process objects that are executing the application. The *sc_p_numschedulers* data member stores the maximum number of process objects that can be used in the application. The number of process objects desired to run an application can be set by the user as an argument to the constructor of the *Scheduler* class.

The *sc_p_activeschedulers* records the number of scheduler *threads* that are currently active in the application. During the initialization phase of PRESTO, this data member starts at 1 and reaches the value of the maximum number of process objects created in the system. The *sc_quantum* data member records the quantum size for which a *thread* will run if the system is configured to run in a preemptive fashion. A zero value for *sc_quantum* indicates that no preemption is to occur.

The *sc_p_busybits* keeps track of which process objects are busy executing a *thread* other than the scheduler *thread*. This data member is used to determine the termination condition in PRESTO - if all process objects are idle (i.e. executing the scheduler *thread*) then the system is idle and can be shut down. Note that the size of *sc_p_busybits* (32 bits) limits the number of processors that can be active in the system. The *sc_lock* data member is used to serialize access to the queue of ready *threads* in the scheduler object. It is also used to serialize access to the *sc_p_busybits* data member.

The *busybits* method is used for two purposes. First, if a non-zero argument is passed to the method, then, the bit corresponding to the current process object is set in the *sc_p_busysits* data member. Second, if a zero argument is passed to the method, the contents of the *sc_p_busybits* data member is returned. The value returned helps to determine the termination condition in PRESTO.

The constructor for the *Scheduler* class has two arguments. The first one signifies the number of process objects (and in turn the number of scheduler *threads*) that must be spawned in the application. Since the scheduler object is responsible for creating all but the first process object in a multi-processor application, this information is provided to the constructor. The second argument specifies the quantum size. If no quantum size is specified by the user, then the value of the predefined macro *DEFQUANTUM* is assigned to the *sc_quantum* data member. Since *DEFQUANTUM* is defined to be 0, not providing a non-zero value for this argument initializes PRESTO for non-preemptive execution. Some of the primary actions in the constructor are to properly initialize the different data members of the class and to invoke the functions that initialize the preemption handling facilities, if necessary.

The *invoke* method is a key method in the *Scheduler* class that plays an important role in the initialization process of the PRESTO run time system. Since we have not modified any step involved in the initialization process, we will not discuss the *invoke* method in detail. Among the primary actions taken in the method are the invocations of the functions that initialize the signal handling and preemption handling routines in PRESTO, the creation of as many process objects as there are available processors in the system, setting the corresponding bit in the *sc_p_busybits* data member as each process object is created, and finally returning the number of process objects that were created in the method.

The *resume* method is used to put a *thread* into the queue of the scheduler object either after its creation or after it exits a blocked state. The definition of the method follows.

```
void Scheduler::resume(Thread* t)
{
        if (t->flags()&TF_SCHEDULER)
                t->error("Can't resumr a scheduler thread\n");
        t->isready();
        sc_t_ready->insert(t);
}
```

The *thread* pointer, *t*, represents the *thread* that is being resumed. The first thing that is determined is whether *t* is a scheduler *thread* or not. Since scheduler *threads* always execute on the process object they are bound to, they must never be scheduled by the scheduler. If such a condition ever arises then it is an error and a message is displayed. Otherwise, the state of the *thread* is marked ready by invoking the *isready* method. Then the *thread* is inserted into the ready queue of the scheduler by invoking the *insert* method of the *sc_t_ready* data member.

The *getreadythread* method is used to extract a ready *thread* from the queue of the scheduler object. It is defined as follows.

```
Thread* Scheduler::getreadythread()
{
        sc_lock->lock();
        Thread* t = sc_t_ready->get();
        if (t)          {
                (void)busybits(1);
                sc_lock->unlock();
        } else {
                int bust = busybits(0);
                sc_lock->unlock();
                if (busy == 0 && thisproc->isroot() )
                        this->halt();
        }
        return t;
}
```

The *getreadythread* method is invoked by the scheduler *thread.* This method is invoked to determine the next piece of work for the process object on which the scheduler *thread* is executing. The first action in the method is to ensure exclusive access to the scheduler object by locking the *sc_lock* data member. Then the *get* method is invoked in *sc_t_ready* to extract the next ready *thread* in the queue, if any. If a ready *thread* is found, the *busybits* method is invoked with an argument of 1. This sets the bit in the *sc_p_busybits* data member that corresponds to the process object on which the scheduler *thread* is executing. Finally, *sc_lock* is unlocked and a pointer to the extracted *thread* is returned. If no *thread* is available for scheduling, the *busybits* method is invoked with an argument of zero. This invocation returns the contents of the *sc_p_busybits* data member. After assigning the return value to the *busy* variable, *sc_lock* is unlocked. If *busy* is found to be zero it indicates that no process objects are busy and along with the fact that the queue in the scheduler is empty, it implies that the application program has finished.

The responsibility for making the preparations to shut down the PRESTO run time system lies with the "root" process that executed *Scheduler::invoke* and spawned all the remaining process objects. A process object determines if it is the "root" by using the *isroot* method. This method is invoked using *thisproc* which is a private variable that is maintained by each process object to store a pointer to itself. If the current process object is at the root of the process object hierarchy, the *halt* method in the scheduler object is invoked which makes the necessary preparations for shutting down the PRESTO run time system. Then a null *thread* pointer is returned.

The *readyqlen* is an inline method that returns the size of the current ready queue in the scheduler object by invoking the *size* method in the *sc_t_ready* data member.

8.3.2 Implementing Selective Thread Binding in PRESTO

To implement selective *thread* binding, the four following features had to be incorporated in PRESTO.

- The ability to mark a *thread* that has been created in an IA and to keep a record of the process object on which it must be executed.

- The addition of process specific queues that store *threads* which must be run on a specific process object.

- The ability to selectively enqueue *threads* in the process specific queues.

- The ability to dequeue *threads* from process specific queues.

To implement the *thread* marking, new data members and methods were added to the *Thread* class in PRESTO. The additions are listed below.

```
class Thread : public Object  {
protected:
            int t_iactor;     //thread created inside int. actor
            Process* t_runonlyon;
                              //process on which to run
            int t_processq; //process queue in which thread
                              //will reside in scheduler
public:
            int get_proc()
                  {return (int)t_runonlyon;}
            int set_proc(Process* p)
                  {t_runonlyon = p;}
            void set_iactor()
                  {t_iactor = 1;}
            void set_processq(int i)
                  {t_processq = i;}
            int get_processq()
                  {return t_processq;}
            int test_iactor();
}

int Thread::test_iactor()
{
      if (t_iactor)
            return 1;
      else
            return 0;
}
```

The *t_iactor* data member in the *Thread* class is used to mark a *thread* that has been created inside an IA (henceforth referred to as an IA *thread*). If it is set to 0 then the *thread* is an oridinary *thread*. If set to 1 the *thread* is an IA *thread*. The *t_runonlyon* data member is used to keep a record of the process object on which the *thread* was created and on which it must always execute. The *t_processq* data member is used to speed up the enqueueing process for IA *threads*. It stores the position in the array of queue pointers in the scheduler object of the process specific queue in which this *thread* will be enqueued when ready to execute. The methods added to the *Thread* class operate on these three data members in order to set, reset, or return them and are self explanatory.

To implement the queue related features we replaced the scheduler object in PRESTO by a new scheduler object which is an instantiation of the *Prty_Scheduler* class in ACT++. The partial structure of the new scheduler object is shown in **Figure 14**.

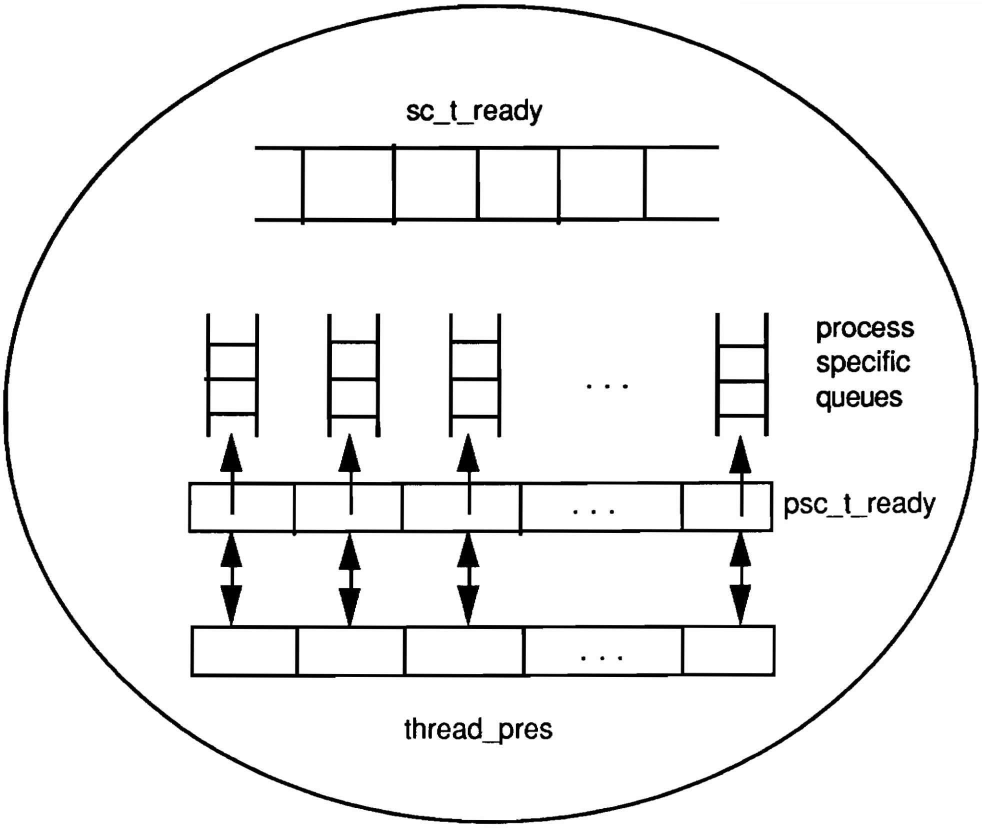


Figure 14: The partial structure of the new scheduler object in ACT++. The sc_t_ready queue is used to store threads that have not been created in interface actors. The queues pointed to by the elements of the psc_t_ready array are used to store threads that are created in interface actors and must execute on specific process objects. The elements of the thread_pres array keep track of the availability of threads in process specific queues.

The *Prty_Scheduler* class is defined as follows.

```
class Prty_Scheduler : public Scheduler   {
protected:
            ThreadPool* psc_t_ready[NUMPROCS];
            int thread_pres[NUMPROCS];
            virtual int get_prty_qptr(int i);
            virtual void incr_thread_pres(int);
            virtual void decr_thread_pres(int);
public:
            Prty_Scheduler(int numschedulers,
                              int quantum = DEFQUANTUM);
            virtual ~Prty_Scheduler();
            virtual Thread* getreadythread();
            virtual void resume(Thread* t);
            virtual int invoke();
            virtual int readyqlen();
            virtual int Scheduler_qlen();
            virtual int process_qlen(int);
}
```

Among the data members of the *Prty_Scheduler* class, *psc_t_ready* is an array of pointers to queues as shown in Figure 14. There are as many pointers in the array as there can be process objects in the system. Each element of the array can store a pointer to a queue that represents the queue of ready IA *threads* that will execute only on that process object. That is, *psc_t_ready[i]* is the pointer to the ready queue for the process object *i*. Recall that the data member *sc_p_procs[i]* in the *Scheduler* class stores a pointer to the *ith* process object. As a result, to locate the pointer to the queue for process object *i*, it is sufficient to know the location in *sc_p_procs* which stores the pointer to this process object. The same index in *psc_t_ready* contains the process specific queue pointer for the process object. The *NUMPROCS* macro is predefined in PRESTO as 16 which indicates that there can be at most 16 process objects in the system.

The *thread_pres* data member in *Prty_Scheduler* is an array of integers that stores the status of the IA *thread* queues. If *thread_pres[i]* is set to 1 then the queue pointed to by *psc_t_ready[i]* has at least one ready *thread* in it. If the *ith* queue is empty, then, *thread_pres[i]* stores a 0.

As discussed above, the position of the process specific queue pointer in the *psc_t_ready* array has a one-to-one correspondence to the position in *sc_p_procs* that stores the pointer to the process object. To do process specific enqueueing and dequeueing of IA *threads* efficiently, the pointer to the process specific queue is stored permanently in an IA *thread*. To do so, the *get_prty_qptr* method is used which is defined as follows.

```
int Prty_Scheduler::get_prty_qptr(int i)
{
      int found = 0;
      while (!found)
            for (int j = 0; j < sc_p_numschedulers; j++)
            {
                  int p = (int) sc_p_procs[j];
                  if (i == p)
                        return j;
            };
      return -1;
}
```

The argument to the above method is actually the integer value of a pointer to a process object. That integer value is searched among the process object pointers stored in the *sc_p_procs* array. The number of elements in *sc_p_procs* that are searched depends on the number of process objects that are executing the application. Recall that *sc_p_numschedulers* stores that number. After locating the process object pointer in *sc_p_procs*, the corresponding index value is returned. If the process object pointer is not found in *sc_p_procs* (which should never happen), then, it indicates a fatal error and a -1 is returned. The significance of the *while* statement will be explained at the end of this section.

The *incr_thread_pres* method is defined as follows.

```
void Prty_Scheduler::incr_thread_pres(int i)
{
      thread_pres[i] = 1;
}
```

The argument to the above method represents the position of the IA *thread* queue pointer for a specific process object in the *psc_t_ready* array. The position in *thread_pres* corresponding to argument is set to one indicating that the queue has at least one ready *thread*.

The *decr_thread_pres* method is defined as follows.

```
void Prty_Scheduler::decr_thread_pres(int i)
{
      if ((psc_t_ready[i]->size() == 0) && (thread_pres[i]))
            thread_pres[i] = 0;
}
```

The above is very similar to the *incr_thread_pres* method. If the IA *thread* queue at the *i* th location in the *psc_t_ready* array has no *threads* (determined by invoking the *size* method in *psc_t_ready*) and *thread_pres[i]* is not zero, then, *thread_pres[i]* is set to zero indicating that the queue is empty.

The *Prty_Scheduler* constructor is defined as follows.

```
Prty_Scheduler::Prty_Scheduler(int numschedulers,
      int quantum = DEFQUANTUM) : (numschedulers, quantum)
{
      psc_t_ready[0] = new ThreadPoolQueue;
      for (int j = 0; j < NUMPROCS; j++)
            thread_pres[j] = 0;
}
```

The initializer list in the above constructor is for the constructor in the superclass, *Scheduler*. Among the actions in *Prty_Scheduler*, the first is to create a new queue object which is an instantiation of the *ThreadPoolQueue* class in PRESTO. A pointer to this new queue is stored in *psc_t_ready[0]*. This queue would store ready IA *threads* that must execute on the process object pointed to by *sc_p_procs[0]*. The second action is to assign zeros to all the elements of the *thread_pres* array.

The reason why only one IA *thread* queue is created in the *Prty_Scheduler* constructor is that the time at which the scheduler object is created during the initialization of the PRESTO run time system, only one process object exists in the system. All subsequent process objects, if any, are created at a later time and hence the creation of the corresponding IA *thread* queues are also delayed.

The *Prty_Scheduler* destructor is defined as follows.

```
Prty_Scheduler::~Prty_Scheduler()
{
	for (int i = 0; i < sc_p_activeschedulers; i++)
		delete psc_t_ready[i];
}
```

The only activity in the destructor is to delete all the IA *thread* queues in the scheduler object. Note that *sc_p_activeschedulers* and *sc_p_numschedulers* store the same value once the PRESTO run time system has been initialized; both store the number of process objects that are executing the PRESTO application.

The *getreadythread* in *Prty_Scheduler* is a generalization of the *getreadythread* method in the *Scheduler* class. It is defined as follows.

```
Thread* Prty_Scheduler::getreadythread()
{
	int i = (int) thisproc;
	int k = get_prty_qptr(i);
	Thread* t;
	if (k == -1)	{
		cerr << "\n -1 RETURNED BY GET_PRTY_QPTR IN
				GETREADYTHREAD \n";
		this->abort(SIGKILL);
		kill(getpid(), SIGILL);
		//NOT REACHED
	}
	if (psc_t_ready[k]->size() != 0)
	{
		t = psc_t_ready[k]->get();
		decr_thread_pres(k);
		sc_lock->lock();
```

```
                (void) busybits(1);
                sc_lock->unlock();
        } else
                t = Scheduler::getreadythread();
        return t;
}
```

The above method is used to extract a ready *thread* from the scheduler object by the scheduler *thread* executing on an idle process object. The functionality of the method is the same as *Scheduler::getreadythread* except that code has been added to search process specific queues before searching the general queue. That is achieved by first converting the value of the *thisproc* pointer of the idle process object into an integer and then invoking the *get_prty_qptr* using that integer value as an argument. *Get_prty_qptr* returns the position of the IA *thread* queue pointer in *psc_t_ready* corresponding to the idle process object. If the returned value is a -1 then it is a fatal error and the PRESTO run time system is aborted. Otherwise, if the size of the IA *thread* queue of the idle process object is not zero, then, a ready IA *thread* is extracted, *busybits* is invoked inside a critical section, and a pointer to the extracted *thread* is returned.

If the IA *thread* queue has no ready *threads* for the idle process object, then, the general queue is searched by invoking *Scheduler::getreadythread.* Then, whatever is returned by *Scheduler::getreadythread* is returned by *Prty_Scheduler::getreadythread.* Note that the reason why we do not need to access the *psc_t_ready* queue in a critical section is that no two scheduler *threads* would ever try to extract a ready *thread* from the same IA *thread* queue because there is a unique queue for each process object, only one *thread* executes on a process object at any time, and a scheduler *thread* is non-preemptable. Yet, many *threads* might try to invoke *busybits* simultaneously. Hence it is invoked in a critical section.

Note that searching the process specific queues before searching the general queue implicitly assigns a higher priority to IA *threads* over ordinary *threads*. This prioritizing is reasonable because it ensures that I/O activities, which are usually time consuming, will always be attended to first in an ACT++ application. This also makes ACT++ especially suitable for real time applications in which I/O activities play a significant role and must be completed under tight time constraints.

The *resume* method in *Prty_Scheduler* is used to enqueue a *thread* that has just been created or was blocked and has become ready for execution once again. The method is defined as follows.

```
void Prty_Scheduler::resume(Thread* t)
{
	int index;
	if (t->flags()&TF_SCHEDULER)
		t->error("Can't resume a scheduler thread\n");
	t->isready();
	if (t->test_iactor())
	{
		if ((t->get_processq()) == -1)		{
			index = get_prty_qptr(t->get_proc());
			t->set_processq(index);
		} else
			index = t->get_processq();
		psc_t_ready[index]->insert(t);
		incr_thread_pres(index);
	} else
		sc_t_ready->insert(t);
}
```

The above method is very similar to the *resume* method in the *Scheduler* class. In fact, the test for the scheduler *thread*, marking the *thread* ready, and inserting the *thread* in the *sc_t_ready* queue are exactly the same. The additional code is to tackle the enqueuing of an IA *thread* in the proper IA *thread* queue. That is achieved by testing whether the *thread* is an IA *thread* by invoking the *test_iactor* method in the thread *t*. *Test_iactor* returns a 1 if the *t_iactor* data member is set, or a 0 if not. If a 1 is returned then *t* is an IA *thread* and so has to be enqueued in a process specific queue instead of the general queue.

To determine which IA *thread* queue to enqueue *t* to, first, the *get_processq* method is invoked in *t*. If the *thread* has just been created then the queue pointer has not been recorded in the *thread* yet. *Get_processq* indicates this by returning a -1. If a -1 is not returned, then, the *index* records the location of the IA *thread* queue pointer in *psc_t_ready* for the *thread*. Otherwise, the *get_proc* method is invoked in *t* to obtain the process object on which *t* must execute. Then, *get_prty_qptr* is invoked with the integer value of the process pointer as the argument. This invocation returns the location of the IA *thread* queue pointer in *psc_t_ready* for *t* which is recorded in *t* once and for all by invoking the *set_processq* method in it.

After the correct IA *thread* queue for *t* has been determined and stored in *index*, *t* is placed on the queue by invoking the *insert* method in *psc_t_ready[index]*.Then the *incr_thread_pres* method is invoked with *index* as the argument to record the presence of a IA *thread* in the queue.

The *invoke* method is a generalization of the *Scheduler::invoke* method. It is defined as follows.

```
int Prty_Scheduler::invoke()
{
      int j;
      if (sc_p_numschedulers < 0)
            return sc_p_activeschedulers;
      for (j = 1; j < sc_p_numschedulers; j++)
            psc_t_ready[j] = new ThreadPoolQueue;
      int i = Scheduler::invoke();
      return i;
}
```

As noted before, *Scheduler::invoke* is a method that plays a significant role in the initialization of the PRESTO run time system. We have not altered the sequence of events taking place during the initialization. Due to the addition of the process specific queues in the new scheduler we needed to add the event of creating those queues to the initialization process. That has been done in the above method before invoking *Scheduler::invoke.*

The check on *sc_p_numschedulers* ensures that the number of process objects to be created has not been mistakenly specified to be a negative number. If so, single process execution is assumed, the number of active scheduler *threads* already created is returned, and the rest of the initialization process is skipped.

If *sc_p_numschedulers* is a positive number, those many IA *thread* queues are created and pointers to those queues are stored in *psc_t_ready*. Then *Scheduler::invoke* is invoked to complete the rest of the initialization process.

Now the significance of the *while* loop in the *get_prty_qptr* method in the *Prty_Scheduler* class will be explained. It is related to the creation of new process objects from the *invoke* method in the *Scheduler* class. The following statement is used to create and record the pointer to a new process object in the *Scheduler::invoke* method:

```
sc_p_procs[pid] = thisproc->newprocess(pname, pid+thisproc->id()).
```

In the above, the *newprocess* method is invoked on the process object pointed to by the *thisproc* pointer which returns a pointer to a new process object. That pointer is recorded in *sc_p_procs[pid]* where *pid* is an index variable. But before the new process object is recorded in *sc_p_procs*, it may so happen that the scheduler *thread* in the new process object invokes the *getreadythread* method in the scheduler object in order to extract a ready *thread*. The very first action in *getreadythread* is to invoke the *get_prty_qptr* method on the scheduler object in order to obtain the pointer to the process specific queue for the process object on which the scheduler *thread* is executing. The way *get_prty_qptr* determines the location of the process specific queue pointer in the *psc_t_ready* array is to search the *sc_p_procs* array for the process object under consideration. Since the newly created process object is not yet recorded in *sc_p_procs*, the search will fail thereby causing a fatal error. To prevent that from happening, the *while* loop has been introduced in *get_prty_qptr*. Due to the loop, the search continues until the assignment is completed and a match is found. Note that control returns immediately after the match is found in *get_prty_qptr*. Hence *found* is never assigned to true.

The remaining methods of the *Prty_Scheduler* class are operations on the different queue structures in the scheduler object. These methods are used to do preemptive scheduling in ACT++. On the expiration of a time quantum the preemptive scheduler determines the overall load of ready *threads* in the scheduler and depending on the load, decides how many active *threads* to preempt.

The *readyqlen* method is used to determine the total number of ready *threads* in the scheduler object. It is defined as follows.

```
int Prty_Scheduler::readyqlen()
{
      int i = Scheduler::readyqlen();
      for (int j = 0; j < NUMPROCS; j++)
            i = i + thread_pres[i];
      return i;
}
```

The first action in the above method is to invoke *Scheduler::readyqlen* to determine the size of the general queue. Then each of the IA *thread* queues is considered in turn to check for the presence of any IA *thread*. Since a process object can execute only one *thread* at a time, it does not matter how many IA *threads* are waiting to execute on a process object; only one need be counted towards determining the total size. Hence the *i* th element of the *thread_pres* array is added to *i* in the *for* loop because *thread_pres[i]* can be at most 1 if there is at least one *thread* in the queue, or it will be zero. At the end of the scan over the *thread_pres* array, the total number of ready *threads* in the scheduler is returned.

The *Scheduler_qlen* method is used to determine the total number of ready *threads* in the general queue alone. To do so, the *Scheduler::readyqlen* method is invoked and the number returned by it is returned as the result.

The *process_qlen* method is used to determine whether the IA *thread* queue for a specific process object has any ready IA *threads* in it. It is used by the preemptive scheduler to look for a ready *thread* in the general queue if the process specific queue is found to be empty for a particular process object. It is defined as follows.

```
int Prty_Scheduler::process_qlen(int i)
{
      if (thread_pres[i] > 0)
            return 1;
      else
            return 0;
}
```

The argument to the above method is the position in *psc_t_ready* of the IA *thread* queue pointer for the process object in question. If *thread_pres[i]* is 1 then the queue has at least one ready *thread* and a 1 is returned. Otherwise a 0 is returned.

8.3.3 Extending ACT++ To Handle Multi-Processor I/O

With the selective *thread* binding capability in place, we can now discuss the changes necessary in the existing class definitions of ACT++ to use an IA in a multi-processor ACT++ application. The first changes, involving the *Actor* class and the *IActor* constructor, are shown below.

```
class Actor : public Object   {
      int iactor;
      Process* my_process;
public:
      void set_iactor();
      void set_proc(Process*);
}

IActor::IActor(char* fname, Tty_Beh* init_beh, char* name)
                  : ((Behavior*) init_beh, 1, name)
{
      thisthread->nonpreemptable();
      Actor::set_proc(thisproc);
      Actor::set_iactor();
      int fd = init_beh->Open(fname);
      thisthread->preemptable();
}
```

Two new data members have been added to the *Actor* class. The *iactor* data member is set when an IA is created and is used to take special scheduling actions on the *threads* created inside the IA. The *my_process* data member stores a pointer to the process object on which the *threads* created in the IA must execute. Note that these two pieces of information have to be stored inside the IA object because at the time an IA is created no *threads* are created inside it. Later, when the IA processes request messages, the *iactor* data member is used to mark the *threads* created to process the messages and the *my_process* data member is used to set the *t_runonlyon* data member in the *threads*.

The *set_iactor* method sets the *iactor* data member to 1. The *set_proc* method sets the *my_process* data member to the process object pointer sent as an argument to the method.

Two of the additions in the *IActor* constructor are invocations to the *set_iactor* and *set_proc* methods in the *Actor* class. The *thisproc* pointer of the process object executing the *IActor* constructor is passed as an argument to *set_proc*. This signifies that all *threads* created in the IA must execute on this process object.

The invocations of the *nonpreemptable* and *preemptable* methods in *thisthread* play a very significant role. Since PRESTO can be configured to run as a preemptive system, a *thread* executing the *IActor* constructor can be preempted at any point during its execution. After being resumed, the *thread* might execute on a different process object. If such a thing happens between the invocation of the *set_proc* method and the invocation of the *Open* method, then, the old process object would be recorded as the site of execution of all subsequent *threads* in the actor, and yet, the file descriptor obtained in *Open* will be on the new process object. To avoid this file descriptor consistency problem we mark the *thread* as nonpreemptable at the very beginning and mark it as preemptable at the very end.

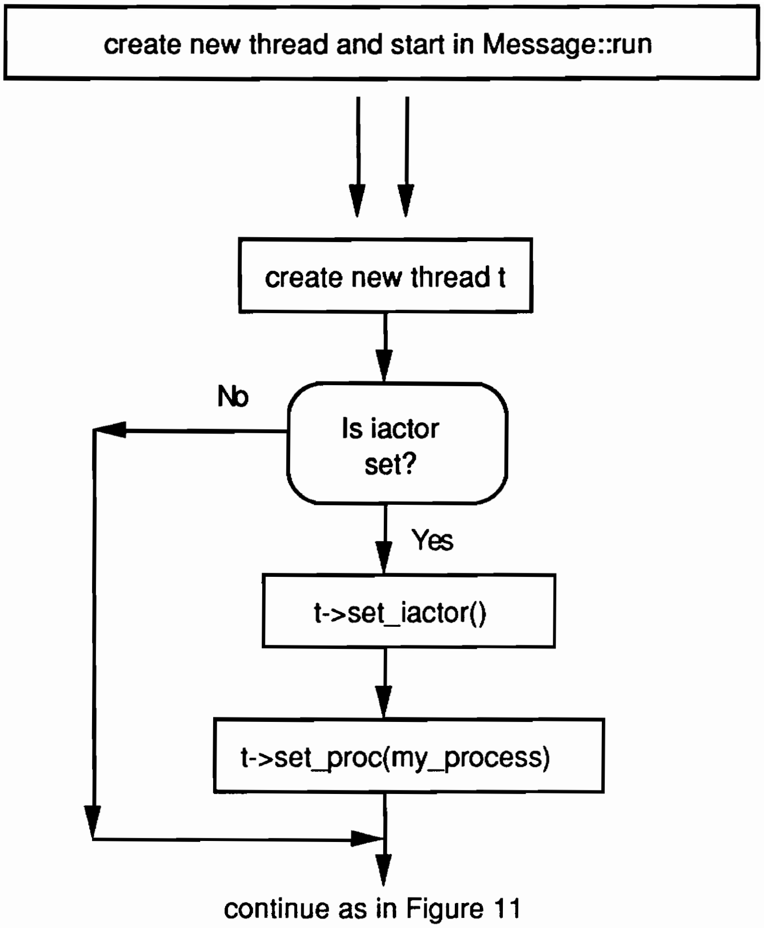


Figure 15: The modification in the implementation of getsched_beh, getsched_mess, and upd_sched_mess methods in the Actor class in order to handle I/O on multiple processors.

The next addition involves the three key methods in the *Actor* class, namely, *getsched_beh*, *getsched_mess*, and *upd_sched_next*. Referring to Figure 11 in this chapter, the "create new thread and start in Message::run" box in the flowchart is expanded as shown in **Figure 15**. The addition involves checking whether the *iactor* data member is set after creating a new *thread*. If so, the *t_iactor* data member is set in the new *thread* by invoking the *set_iactor* method on the *thread*. Also, the *t_runonlyon* data member is set in the new *thread* by invoking the *set_proc* method and passing the *my_process* data member in the *Actor* class as the argument.

The last addition involves modifying the VTALRM signal handler in PRESTO which is called *sigpreempt_alrm*. This signal handler is executed at the expiration of the allowed time quantum when PRESTO runs in a preemptive fashion. This signal handler is responsible for determining which process objects are running *threads* that are preemptable, and in such a case, signalling those process objects to execute the next ready *thread* in the queue of the scheduler. Note that only the process at the root of the process hierarchy that executes a PRESTO application is configured to receive the VTALRM signal. The addition of the process specific queues in the scheduler warranted modifications to the signal handler. The signal handler along with the additions is shown below. Each addition is identified by a comment.

```
int sigpreempt_alrm(int sig, int code, struct sigcontext* scp)
{
      register *sp = (int*) scp->sc_sp;
      Process* p;
      Thread* t;
      int numtopreempt;
      int i;
      int numinscheduler;     //Addition
      numtopreempt = sched->readyqlen();
      numinscheduler = sched->Scheduler_qlen(); //Addition
      numalarms++;
      double q = ((double) sched->quantum()) / 1000.0;
      for (i = 0; numtopreempt &&
            i < sched->sc_p_activeschedulers; i++) {
            p = sched->sc_p_procs[i];
            int r = sched->process_qlen(i);      //Addition
            if ((!r) && (!numinscheduler))
                  continue;                            //Addition
            t = p->runningthread();
            if (t && t->canpreempt())      {
                  if (p==thisproc)
                        (void)sigpreempt_notify(sig,code,scp);
                  else
                        kill(p->pid(), SIGPREEMPT_NOTIFY);
                  numtopreempt--;
                  if ((!r) && (numinscheduler > 0))
                        numinscheduler--;       //Addition

            }
      }
```

```
        return 0;
}
```

We will not discuss the signal handler in great detail. We will first discuss what the operation of the handler was before effecting the changes. Then we will discuss the effects of the additions. The general scheme of operation of the original handler was to determine the number of ready *threads* in the queue of the scheduler by invoking the *readyqlen* method in *sched*, the shared pointer to the scheduler object. This number gave it some idea about how many process objects to examine for possible *thread* preemption. If there were more ready *threads* than process objects, then, all the process objects were examined. If there were fewer *threads* than process objects, then, all the process objects were not examined. This was the basic idea behind the condition that drove the *for* loop in the handler. Note that the signal handler never extracts any *thread* from the queue of the scheduler object. The process objects that are signalled are responsible for extracting the *threads*. The signal handler just ensures that enough *threads* are preempted and the corresponding process objects relieved so that the waiting *threads* in the scheduler object get a chance to execute.

To interrogate the status of the *threads* running on the process objects, the process object pointers stored in the *sc_p_procs* data member of the scheduler were used. Starting from 0, each process object pointer in *sc_p_procs* was obtained and a pointer to the *thread* running on that process object was retrieved. Then it was determined whether the *thread* could be preempted by invoking the *canpreempt* method in the *thread*. If preemptable, a signal was sent to the process which would eventually result in the execution of another signal handler, *sigpreempt_notify*, which would prepare the current *thread* for preemption and enable the switchover to the scheduler *thread*. If the process object being interrogated happened to be the one on which the *sigpreempt_alrm* function was executing, the *sigpreempt_notify* function was invoked directly instead of sending a signal.

To account for the process specific queues added to the scheduler object the handler was modified as shown above. The modifications have resulted in the following changes to the scheduling algorithm.

- The process specific queues are searched for a ready *thread* before searching the general queue. Therefore IA *threads* are given higher priority than general *threads*.

- If a particular process specific queue is found empty only then is the general queue searched for a ready *thread*. If the general queue is empty then the *thread* on the current process object is not chosen for preemption and hence the process is not signalled.

- An empty general queue does not mark the end of the scheduling algorithm. The algorithm terminates only after all the process specific queues have been searched.

In the modified *sigpreempt_alrm* function, due to the modification of the *readyqlen* method in *Prty_Scheduler*, the *numtopreempt* variable receives the total number of ready *threads* available in the scheduler which includes both IA and general *threads*. Then code has been added to determine the number of *threads* available in the general queue alone which is assigned to the *numinscheduler* variable. The latter variable is used to keep track of the number of ready *threads* in the general queue that are available for execution on process objects that do not have any IA *thread* waiting for them. As each *thread* is scheduled for execution from the general queue, this variable is decremented. When the value of this variable reaches zero, only IA *threads* can be scheduled for execution; if a process does not have any IA *thread* in its process specific queue then it must not be signalled. Thus, *numinscheduler* helps to identify this condition.

Inside the *for* loop, after retrieving the *i* th process object pointer from *sc_p_procs*, the *process_qlen* method is invoked on *sched* to determine if there are any ready IA *threads* in the IA *thread* queue for this process. If there is no ready IA *thread* and there are no ready *threads* in the general queue, then, the process object need not be signalled and hence the next iteration of the loop is executed. Note that although the general queue is found to be empty we still have to execute the next iteration of the *for* loop and examine the next process object in *sc_p_procs* because there might be a ready IA *thread* for it in its process specific queue.

Then, as before, it is determined whether the current *thread* on the process can be preempted. If so, the same actions as before is taken along with the following additional action. If the IA *thread* queue for the current process object is empty and there is at least one ready *thread* in the general queue, then, the *numinscheduler* variable is decremented. This is done to ensure that if the ready *thread* in the general queue is the last *thread* to be scheduled for execution on the process object just signalled, then, in the next iteration of the *for* loop, no process would be unnecessarily signalled if its IA *thread* queue is empty.

8.3.4 Restoring File Descriptor Status

The issue of restoring file descriptor status is independent of the issues of performing single or multi-processor I/O in ACT++ but relates to the issue of I/O in general. The issue to be addressed is quite simple. The status of a file descriptor obtained in *Tty_Beh::Open* is changed by setting the FASYNC and FNDELAY flags. This setting of the descriptor is useful as long as the ACT++ application is running, but after the program terminates, the descriptor must be restored to its original status. Otherwise, after the ACT++ program terminates, the user would be automatically logged out. This happens because the *read* done by the shell program on the file descriptor will return immediately (since FNDELAY is set) with a null character. This null character would be interpreted as the end of session request (equivalent to control-D used to logout from the current session on many UNIX systems) and the user would be logged out.

The restoration of file descriptor status must be either done explicitly by the user or the ACT++ runtime system has to do it before termination. In ACT++ both the options are available. There are operations that the user can invoke explicitly to close and restore the status of file descriptors. The descriptors that are not handled by the user are restored by the ACT++ run time system. The file descriptor status restoration process involves the following steps.

- Before changing the status of a descriptor saving its existing status.

- Recording the saved status at a place where it would be available for use at the end of the program.

- Adding the step of file descriptor status restoration to the termination process of the run time system.

The old status of a file descriptor is saved in an object that is an instantiation of the *Fd_Stat* class which is defined as follows.

```
class Fd_Stat : public Object {
      int fd;
      int oldflags;
public:
      Fd_Stat(int fd_arg, int oldflags_arg);
      int ret_fd();
      int ret_oldflags();
}
```

The *fd* data member stores the file descriptor whose status is being changed. The *oldflags* data member stores the old status of the file descriptor.

The first argument to the constructor *Fd_Stat* is a file descriptor and the second argument is its old status. In the constructor, these two items are recorded in *fd* and *oldflags* respectively. The *ret_fd* method returns the value of the *fd* data member. The *ret_oldflags* method returns the value of the *oldflags* data member.

After obtaining a file descriptor in *Tty_Beh::Open*, the *oldflags* variable is set to the existing status of the descriptor by invoking the *fcntl* function with the *F_GETFL* option. Then the *register_fd_stat* method is invoked to record the 'file descriptor - old status' pair in the AIOM object for the current process which is pointed to by the *thisaiom* pointer.

There are several reasons for choosing the AIOM object as the place to record the old status of file descriptors. First, it is the only object in the ACT++ run time system of which there is always one per process object and which can be accessed by any *thread* executing on the process object through the *thisaiom* pointer. The fact that there is one distinct AIOM per process object is very important because the restoration of file descriptors has to be done on a per process basis since file descriptors are undefined across process boundaries. This implies that descriptor status has to be stored in an object that is unique per process and the AIOM is the only such object in ACT++. Second, the AIOM is involved in doing I/O using any file descriptor and as such is the logical choice for storing information pertaining to them. Third, the invocation of a method that actually does the restoration of the file descriptor status can be very easily incorporated into the termination process of the run time system of ACT++ if the AIOM is used for recording the status.

To handle descriptor status restoration the *AsyncIoMgr* class had to be embellished as follows.

```
class AsyncIoMgr  {
      Fd_Stat_queue fd_stat_q;
      int fd_stat_cnt;
public:
      void register_fd_stat(int, int);
      void remove_fd_stat(int);
      ~AsyncIoMgr();
}
```

Since there can be multiple IAs created on a particular process object, each having its own file descriptor, the AIOM might have to restore the status of more than one file descriptor. The *fd_stat_q* data member, an instantiation of the *Fd_Stat_queue* class, is used to store multiple instantiations of the *Fd_Stat* class. The *fd_stat_cnt* data member stores the number of elements in the *fd_stat_q* queue.

The *Fd_Stat_queue* class, like the *Io_queue* and *BehSetq* classes defined before, has a lot of common functionality with the *Mailq* class and is defined as follows.

```
class Fd_Stat_queue : public Mailq  {
public:
      Fd_Stat_queue() {;};
      int lookat_next();
}
```

The constructor for the class is empty and the *lookat_next* method returns the file descriptor in the next *Fd_Stat* object in the queue.

The *register_fd_stat* method in the *AsyncIoMgr* class is used to record a descriptor and its status in the AIOM. The method is defined as follows.

```
void AsyncIoMgr::register_fd_stat(int fd, int oldflags)
{
      Fd_Stat* fds = new Fd_Stat(fd, oldflags);
      fd_stat_q.append(fds);
      fd_stat_cnt++;
}
```

The first thing done in the above method is to create an instantiation of the *Fd_Stat* class out of the file descriptor and the status information passed as arguments. Then the new *Fd_Stat* object is appended to the *fd_stat_q* queue. Finally, the *fd_stat_cnt* data member is incremented to register the addition to the queue.

For the file descriptors that are not restored explicitly by the user, the destructor of the *AsyncIoMgr* class is responsible for the actual restoration of the status of all the file descriptors in the *fd_stat_q* queue. The destructor is invoked through a 'delete *thisaiom*' statement from the destructor of the *Process* class in PRESTO. The idea is to destruct the

AIOM object on each process object before destructing the process object itself and as a side effect, close and restore the status of all open file descriptors. The AIOM destructor is defined as follows.

```
AsyncIoMgr::~AsyncIoMgr()
{
      for (int i = 0; i < fd_stat_cnt; i++)       {
            Fd_Stat* fds = (Fd_Stat*) fd_stat_q.get();
            if (fds)    {
                  if (fcntl(fds->ret_fd(), F_SETFL,
                              fds->ret_oldflags()) < 0)      {
                        perror("\n fcntl F_SETFL error in
                                    ~AsyncIoMgr \n");
                        fatalerror();
                  }
            }
            close(fds->ret_fd());
      }
}
```

The destructor loops through all the elements in the *fd_stat_q* queue, extracting each element and invoking the *fcntl* function with the file descriptor and its old status as arguments. The *ret_fd* and *ret_oldflags* methods are invoked on the *Fd_Stat* object extracted from the queue to obtain the file descriptor and its status respectively. The *F_SETFL* option of *fcntl* is used to restore the descriptor to its old status. Then the file descriptor is closed using the *close* function. The above is done *fd_stat_cnt* times.

To implement operations that can be explicitly invoked to close and restore the status of file descriptors the following additions were made.

- A *Close* method was introduced in the *Tty_Beh* class that must be invoked by the user to do file descriptor status restoration.

- Since a file descriptor and its old status are recorded from *Tty_Beh::Open* at the time of creation of an IA, file descriptors restored explicitly must be removed from the *fd_stat_q* in the AIOM. To do so, the *remove_fd_stat* method was introduced in the *AsyncIoMgr* class.

- IAs on which the *Close* operation is invoked are rendered useless because the file descriptor for doing I/O is no longer available. Any message sent to such a "dead" IA must not have any effect and must not be unnecessarily enqueued in the message queue. A special *become* operation in the *Behavior* class, *become(DEAD_ACTOR)*, along with additions to the *Actor* class have been introduced to delete such messages to a "dead" IA and prevent the creation of any *threads* to process such messages. As a result, it is the responsibility of the user to ensure that the computation never blocks expecting reply messages from "dead" IAs.

  Note that the implementation to mark actors as "dead" is not limited to IAs only. The *become(DEAD_ACTOR)* operartion is a general operation and can be used from the methods of any user defined behavior. As a result, any actor can be marked as "dead" and the semantics discussed above would be applicable to it. The reason for introducing this feature is to assist the garbage collector which will be implemented in a future version of ACT++.

The specific additions to the different classes are discussed next.

The following addition has been made to the *Tty_Beh* class.

```
class Tty_Beh : public Behavior      {
public:
      void Close();
}

void Tty_Beh::Close()
{
	if (fcntl(fd, F_SETFL,oldflags) < 0)	{
		perror("\n fcntl F_SETFL error in Tty_Beh::Close \n");
		fatalerror();
	}
	close(fd);
	thisaiom->remove_fd_stat(fd);
	fd = -1;
	oldflags = 0;
	become(DEAD_ACTOR);
}
```

The first action in the *Close* method is to use the *fd* and *oldflags* data members in the *Tty_Beh* object to restore the old status of the file descriptor. For that, the F_SETFL option of the *fcntl* system call is used. Then the file descriptor is closed using the *close* system call. Then the *remove_fd_stat* method is invoked on the AIOM object associated with the process object to remove the element corresponding to the file descriptor from the queue of descriptors in the AIOM. Then the *fd* and *oldflags* data members are restored to their initial values and the *become(DEAD_ACTOR)* operation is invoked to specify that the actor is "dead".

The *remove_fd_stat* method in the *AsyncIoMgr* class that is used to remove file descriptors that are restored by the user is defined as follows.

```
void AsyncIoMgr::remove_fd_stat(int fd_arg)
{
	Fd_Stat* fds = (Fd_Stat*) fd_stat_q.lookat();
	int fd_next = fds->ret_fd();
	if (fd_next==fd_arg)	{
		fds = (Fd_Stat*) fd_stat_q.get();
		delete fds;
		fd_stat_cnt--;
		return;
	}
	fd_stat_q.reset();
	fd_next = fd_stat_q.lookat_next();
	while (fd_next > -1)	{
		if (fd_next==fd_arg)	{
```

```
                  fds = (Fd_Stat*) fd_stat_q.rem_curr();
                  delete fds;
                  fd_stat_cnt--;
                  return;
            }
            else  {
                  fd_stat_q.shift_pos();
                  fd_next = fd_stat_q.lookat_next()
            }
      }
}
```

The above method is invoked with the file descriptor that is being restored by the user as an argument. That descriptor is first tested for a match with the descriptor in the first object in the *fd_stat_q* queue. If a match is found, the object is extracted, deleted, and the *fd_stat_cnt* data member is decremented to record the removal of an item from the queue. If a match is not found the rest of the queue is searched and the same action is taken when a match is found. The search process for the rest of the queue is very similar to the search performed on the *io_pend_list* queue in *AsyncIoMgr::propagate_signal* and hence will not be explained here.

The following additions have been made to the *Actor* class.

```
class Actor : public Object   {
      int dead_actor;
public:
      void set_dead_actor();
}

void Actor::set_dead_actor()
{
      dead_actor = 1;
}
```

The *dead_actor* data member is assigned to 0 in the *Actor* constructor and is set to 1 by the *set_dead_actor* method to record that the actor is "dead".

Since messages sent to a dead actor must be deleted and no *threads* must be created to process them, the *getsched_beh*, *getsched_mess*, and *upd_sched_next* methods in the *Actor* class have been modified as follows.

```
void Actor::getsched_beh(Message* m)
{
        actor_lock.lock();
        if (dead_actor)    {
                delete m;
                actor_lock.unlock();
                return;
        }
                .
                .
                .
}

void Actor::getsched_mess(Behavior* b)
{
        actor_lock.lock();
        if (dead_actor)    {
                actor_lock.unlock();
                return;
        }
                .
                .
                .
}

void Actor::upd_sched_next()
{
        actor_lock.lock();
        if (dead_actor)    {
                actor_lock.unlock();
                return;
        }
                .
                .
                .
}
```

The *getsched_beh* is executed on the arrival of a new message object. Hence, if the actor is found to be "dead", the message is deleted and control returned. The check on *dead_actor* in *getsched_mess* is to prevent the situation when, after a *become(DEAD_ACTOR)*, a new replacement behavior is created from the current behavior which could then find an existing message in the queue and process it. The check on *dead_actor* in *upd_sched_next* is to ensure that after completing the execution of the behavior that specified a "dead" replacement actor, the *thread* does not perform the operations to schedule the next activity in the actor.

The following additions have been made to the *Behavior* class.

```
class Behavior : public Object      {
public:
      void become(char);
}

void Behavior::become(char c)
{
      if (c==DEAD_ACTOR)
            my_actor->set_dead_actor();
}
```

The above *become* operation is invoked as a result of *become(DEAD_ACTOR)*. The *my_actor* data member in the *Behavior* class is used in this overloading of the *become* operation to invoke the *set_dead_actor* method in the *Actor* class and thereby set the *dead_actor* data member.

9. The Root Actor and the Actorprog Behavior

The last issue to be discussed is how the user specifies the main program in ACT++ and how the system starts its execution. In PRESTO the user specifies the main program as a method of the *Main* class. That method is called *main*. So a PRESTO program has the declaration for *Main::main* which the system considers to be the main program. The user can also specify two other methods of the *Main* class, *init* and *done*. In *init* one can reassign several of the private data members of the *Main* class thereby altering the values of system parameters and do application specific initializations. In *done*, the user can specify application specific actions pertaining to system termination.

We will explore the sequence of events that take place in the PRESTO run time system from bootstrapping until termination by looking at the partial call tree shown in **Figure 16**. Understanding this will help the explanation of the changes made in ACT++.

PRESTO has a predefined *main* function in which the first major activity is to create an instantiation of the *Main* class. This invokes the constructor *Main::Main* (step 1 in Fig. 15). All the activities in a PRESTO program take place from the constructor of the *Main* class. When control returns from *Main::Main* the user program has already been executed and the system is about to terminate.

The next major activity is to create an instantiation of the *Thread* class (step 2) which will be used subsequently to create other *threads*. The next activity is to invoke the *init* method in the *Main* class (step 3). If the user has supplied an *init* then it is executed at this point. Otherwise, the default *init* method is executed. The next step is to create the scheduler object (step 4) if it has not been created by the user in *init*. The next activity (step 5) is to create the *thread* that will be the very first one to be executed by PRESTO. This *thread* is scheduled to execute the *invoke* method in the scheduler object (step 6). The next two actions (steps 7 and 8) are performed *nummainthreads* (a private data member of the *Main* class) times which create new *threads* and schedule them to execute the *main* method in the *Main* class. This indirectly means that these two actions repeatedly schedule *threads* that will execute the program supplied by the user. *Nummainthreads* is assigned 1 by default, but, the user can change its value in *init*. Note that assigning *nummainthreads* to be more than 1 means that all these *threads* would simultaneously execute all of *Main::main*. The user, then, has to carefully control the concurrency in the program in order to perform a meaningful computation.

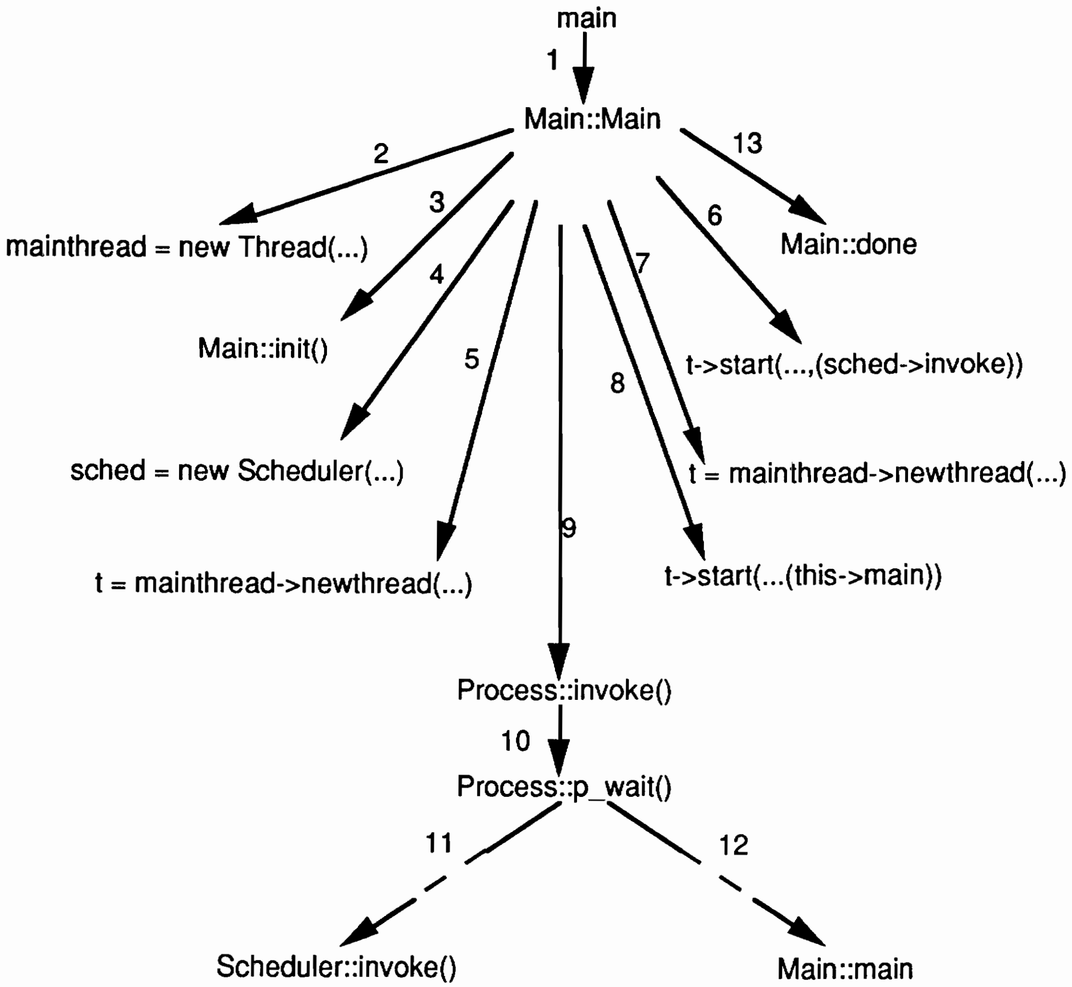

main
1
Main::Main
2
mainthread = new Thread(...)
3
Main::init()
4
sched = new Scheduler(...)
5
t = mainthread->newthread(...)
13
Main::done
6
t->start(...,(sched->invoke))
7
t = mainthread->newthread(...)
8
t->start(...(this->main))
9
Process::invoke()
10
Process::p_wait()
11
Scheduler::invoke()
12
Main::main

Figure 16: The partial call tree showing the major method invocations involved in PRESTO from bootstrapping until termination.

The next action (step 9) is to execute the *invoke* method in the process object on which the *Main* constructor is executing. This in turn invokes the *p_wait* method in the process object (step 10). The *p_wait* is the method that the scheduler *thread* executes to wait for work.

When the scheduler *thread* gets a ready *thread* in the queue of the scheduler, a series of actions is triggered that finally culminates in the execution of the very first *thread* in PRESTO, the *thread* that was scheduled to execute the *invoke* method in the scheduler (step 11). The *invoke* method completes the bootstrapping process in PRESTO.

After executing *invoke,* control returns to *p_wait.* Assuming that there is only one process object, the next *thread* in the scheduler object is executed (step 12) by it. This is the *thread* that executes the *main* method in *Main.* This is the point at which the user program really starts executing. After the user program has been executed and no more *threads* are available for execution, control returns to *Main::Main* once again. Then the major action is to invoke the *Main::done* method (step 13). If *done* has not been defined by the user, the default *done* method is executed. A few more actions completes the termination of PRESTO.

In ACT++ the user specifies the main program as the *Execute* method of the *Actorprog* class which is defined as follows.

```
class Actorprog : public Behavior   {
public:
            Actorprog() {;}
            void Setup();
            void Execute();
            void Terminate();
}
```

The *Actorprog* constructor is null. The *Setup* method is the counterpart of *Main::init.* The default definition of *Setup* assigns an instantiation of the *Prty_Scheduler* class to *sched,* the shared pointer to the scheduler object. The default arguments supplied to the *Prty_Scheduler* constructor sets up PRESTO to run on a single processor as a non-preemptive system. If the user wants to change the default setup, an *Actorprog::Setup* method must be defined and the *Prty_Scheduler* constructor must be invoked with the new values (the SCHEDULER macro can be used for this purpose, refer to section 3 in chapter 2).

The default definitions of *Execute* and *Terminate* are null. *Execute* is the counterpart of *Main::main*. *Terminate* is the counterpart of *Main::done*.

Although the three new methods appear to model the three methods in the *Main* class, the change does not merely involve a renaming of the methods. The new strategy is to implement the execution of the user defined main program as an actor operation in order to be homogeneous with the ACT++ model of computation. As a result, an actor, called the *Root Actor*, is created from *Main::Main* to which an instantiation of the *Actorprog* class is assigned as the initial behavior. Then a message is sent to the *Root Actor* requesting the execution of the *Execute* method. When the request message is processed, the user program is executed. Completion of the user program terminates the processing of the message. The changes in the call tree structure as a result of this new strategy is shown in **Figure 17**.

The first new action in the above call tree is step 3 in which an instantiation of *Actorprog* is assigned to the *actprog* variable. In step 4 the *Setup* method is invoked in *actprog* which in turn assigns an instantiation of the *Prty_Scheduler* class to *sched* (step 5). Steps 6 and 7 are as before but the next three steps are new. In step 8 the *Root Actor* is created. In step 9 the message that contains the request for the execution of the *Execute* method is created. In step 10 the message is sent to the *Root Actor*. The sending of this message creates the *thread* that will execute the *Execute* method in *Actorprog* and schedules it for execution. Steps 11 through 13 are once again as before. At step 14 the ACT++ main program starts executing, not as Main::main, but as Actorprog::Execute. In step 15 the *Terminate* method is invoked in *actprog*. The remaining actions involved in terminating PRESTO remain unchanged.

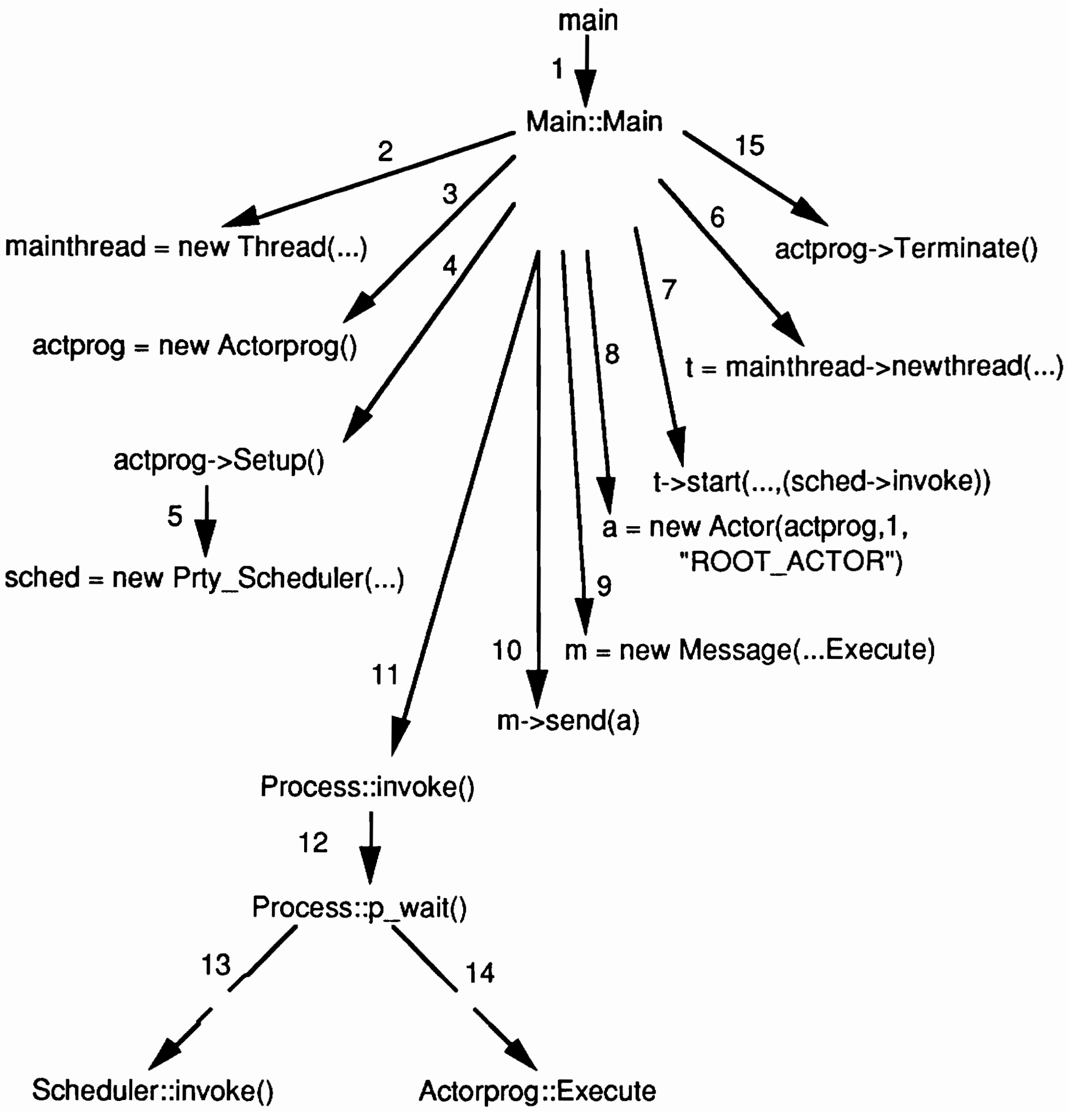

main
1
Main::Main
2
3
4
15
6
7
8
9
10
11
12
13
14
mainthread = new Thread(...)
actprog = new Actorprog()
actprog->Setup()
5
sched = new Prty_Scheduler(...)
actprog->Terminate()
t = mainthread->newthread(...)
t->start(...,(sched->invoke))
a = new Actor(actprog,1, "ROOT_ACTOR")
m = new Message(...Execute)
m->send(a)
Process::invoke()
Process::p_wait()
Scheduler::invoke()
Actorprog::Execute

Figure 17: The partial call tree showing the major method invocations involved in ACT++ from bootstrapping until termination.

Chapter Four

Conclusion

1. From C++ to ACT++

The goal of this project was to implement a concurrent object-oriented programming environment that would enable users of C++ to utilize the modeling features of object-based programming and take advantage of the computational power associated with parallelism. That goal has been achieved. We have successfully implemented the ACT++ programming environment on the Sequent Symmetry multiprocessor using the PRESTO *threads* package. The ACT++ programming environment allows programmers to write concurrent programs in C++ utilizing the features of the Actor model of concurrent computation . The programs in ACT++ can be executed on single or multiple processors on the Symmetry.

One of the salient features of the ACT++ programming environment is the passing of request messages to achieve asynchronous invocation of operations. Actors have message buffering capability and process messages using agents called behaviors. Each behavior of an actor is responsible for specifying a replacement behavior responsible for processing the next available request message. Creation of actors which execute simultaneously and the specification of replacement behaviors which execute in parallel are the two ways of introducing concurrency in an ACT++ program.

To implement asynchronous invocation of methods in concurrent objects the type checking of method invocations in C++ has been sacrificed. This happened due to the rigid definition of the method invocation semantics in the C++ compiler. On seeing the presence of a method invocation statement in the program, the C++ compiler not only generates code to set up the activation record of the callee and type check the arguments but it also generates code to start the execution of the callee. To invoke methods asynchronously, the last action of the compiler must be delayed. But the rigidity of the invocation mechanism does not allow any intervention and hence we had to write functions that replaced the compiler's invocation handling. In so doing, we lost the ability to type check arguments because our mechanism relies on the runtime aspects of method invocation only. Type checking in C++ being static, we could not accomplish that.

A second salient feature of the ACT++ programming environment is its ability to successfully handle the Inheritance Anomaly which arises from the interference between the inheritance mechansim in C++ and the specification of synchronization constraints in the methods of a class. For this purpose we have introduced behavior sets that are used to selectively process messages thereby obviating the explicit specification of synchronization constraints in the methods.

A third salient feature of ACT++ is the introduction of the ability to perform I/O as an actor operation. Instead of accessing files directly from an ACT++ program, the user can create separate I/O servers for different files. These servers are called interface actors. I/O is performed by sending request messages to an interface actor. The blocking of processes due to I/O requests for which the data is not ready when the request is made has been resolved by incorporating the ability of doing asynchronous I/O. Interface actors hide all the details of setting up a file descriptor for asynchronous I/O and transparently deliver the result of asynchronous I/O to the user. The problem of file descriptor consistency due to migration of *threads* has been addressed by incorporating a new scheduler in PRESTO.

2. ACT++ and Beyond

The ACT++ programming environment is by no means a complete environment. Many useful extensions to the system are possible which give rise to issues that are yet to be studied. Such issues and the possible extensions will be discussed in this section.

One source of concurrency in an ACT++ program is the simultaneous execution of actors. The other source of concurrency is the simultaneous execution of behaviors within a single actor. The latter is similar to the notion of *intra-object concurrency* in object-based systems and hence its concern is similar too - the coherence of private data members of an actor in the face of simultaneous accesses. The easy solution to the coherence problem would be to introduce data-based synchronizations in the operations of an actor. But such synchronizations would be in direct conflict with the inheritance mechanism in C++ leading to the Inheritance Anomaly. The adequacy of behavior sets in controlling intra-object concurrency has to be studied and the ability to utilize intra-object concurrency has to be introduced in ACT++.

The introduction of a garbage collector that operates in parallel with an ACT++ application is a very useful extension to the environment. The main barrier to the realization of such a garbage collector is the handling of acquaintances in ACT++. Since an actor can create new actors, know about one or more of the other actors that are executing simultaneously with it (its acquaintances), and pass pointers to actors in messages, the garbage collection problem becomes impossible without assigning a visible identity to the acquaintances of an actor. The most natural way of representing acquaintances in ACT++ is yet to be determined and a garbage collector has to be implemented.

Interface actors in ACT++ can be used to perform null-terminated string I/O only. The introduction of the ability to perform typed I/O will be a very useful extension to the ACT++ environment. The classes for performing typed stream I/O in C++ 2.0 can be utilized to implement such a capability in ACT++. The main obstacle to exploring the possibility of typed I/O has been the dependence of PRESTO on a version of C++ lower than 2.1 that lacks the stream I/O classes for doing typed I/O.

Finally, the deficiencies and strengths of any programming environment are determined by writing extensive application programs using it. It is the next important step that must be undertaken to determine the adequacy of the different features of ACT++. After determining the usefulness of the system, the writing of a compiler for the *ACT++ programming language* must be undertaken which would remove the current dependence on the idiosyncracies of the C++ compiler and would enable a cleaner implementation of the language.

References

[Agha 86] Agha, Gul, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[Agha and Hewitt 87] Agha, Gul, and Hewitt, Carl, "Concurrent Programming Using Actors," In: *Object_Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (eds.), MIT Press, Cambridge, MA, 1987, 37-53.

[America 87] America, Pierre, "Inheritance and Subtyping in a Parallel Object-Oriented Language," *ECOOP '87 Proceedings*, 234-242, Springer-Verlag, 1987.

[Barry et al. 87] Barry, Brian M., Altoft, John, Thomas, D.A., and Wilson, Mike, "Using Objects to Design and Build Radar ESM Systems," *OOPSLA '87 Conference Proceedings*, *SIGPLAN Notices, ACM*, 1987.

[Bershad 88] Bershad, B.N., Lazowska, E.D., and Levy, H.M., "PRESTO: A System for Object-Oriented Parallel Programming," *Software Practice and Experience*, 1988.

[Bershad 90] Bershad, B.N., "The PRESTO User's Manual," Report, Department of Computer Science, University of Washington, Seattle, Washington, 1990.

[Briot 89] Briot, Jean-Pierre, "Actalk: A Testbes for Classifying and Designing Actor Languages in the Smalltalk-80 Environment," *ECOOP '89 Proceedings*, July, 1989.

[Briot and Yonezawa 90] Briot, J-P, Yonezaea, A., "Inheritance and Synchronization in Obhject-Oriented Concurrent Programming," in *ABCL: An Object-Oriented Concurrent System*, (ed. A. Yonezawa), MIT Press, Cambridge, MA, 1990.

[Delagi and Saraiya 88] Delagi, B.A., and Saraiya, N.P., "ELINT in LAMINA: Application of a Concurrent Object Oriented Language," *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, 1988, *SIGPLAN Notices*, 24, 4, April 1989.

[Goldberg and Robson 83] Goldberg, A., and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Joshi 92] Joshi, N., *A Distributed Implementation of ACT++,* M.S. Thesis (in preparation), Departmenet of Computer Science, Virginia Tech, Blacksburg, VA, 1992.

[Kafura and Lee 89] Kafura, D., and Lee, K.H., "Inheritance in Actor Based Concurrent Object-Oriented Languages," *ECOOP '89 Proceedings*.

[Kafura and Lee 90] Kafura, D., and Lee, K.H., "ACT++: Building a Concurrent C++ with Actors," *Journal of Object-Oriented Programming*, Vol. 3, No. 1, 25-37, May/June, 1990.

[Lalonde et al 86] Lalonde, W.R., Thomas, D.A., and Pugh, J.R., "An Exemplar Based Smalltalk," *OOPSLA '86 Conference Proceedings*, ACM, 1986.

[Lavender and Kafura 90] Lavender, G., and Kafura, D., "Specifying and Inheriting Concurrent Behavior in an Actor-Based Object-Oriented

Language," Technical Report TR 90-56, Department of Computer Science, Virginia Tech, Blacksburg, VA, 1990.

[Lee 90] Lee, K.H., *Designing A Statically Typed Actor-Based Concurrent Object-Oriented Programming Language*, Ph. D. Dissertation, Department of Computer Science, Virginia Tech, June 1990.

[Maekawa et al 87] Maekawa, M., Oldehoeft, A. E., and Oldehoeft, R. R., *Operating Systems: Advanced Concepts*, Benjamin/Cummings, CA, 1987.

[Matsuika et al 90] Matsuoka, S., Wakita, K., and Yonezawa, A., "Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages," *OOPSLA '90 Conference Proceedings*, *SIGPLAN Notices*, 1990.

[Reed and Kanodia 79] Reed, D. P., and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers," *CACM*, Feb 1979, v22 no 2, pp 115-123.

[Stroustrup 86] Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, Menlo Park, CA, 1986.

[Yonezawa 90] Yonezawa, A., (ed), *ABCL: An Object-Oriented Concurrent System*, MIT Press, Cambridge, MA, 1990.