ACT++ 3.0 - Implementation of the Actor Model Using POSIX Threads

by

Arjun Khare

Project Report submitted to the faculty of the

Virginia Polytechnic Institute and State University

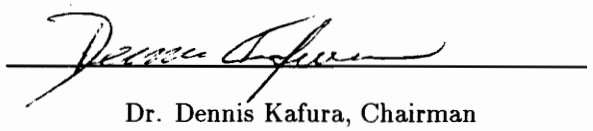
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

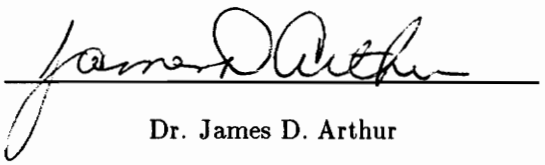
in

Computer Science

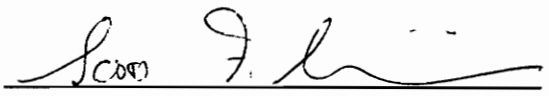
APPROVED:



Dr. Dennis Kafura, Chairman



Dr. James D. Arthur



Dr. Scott F. Midkiff

July, 1994

Blacksburg, Virginia

ACT++ 3.0 - Implementation of the Actor Model Using POSIX Threads

by

Arjun Khare

Committee Chairman: Dr. Dennis Kafura

Department of Computer Science

(ABSTRACT)

The actor model provides a framework for writing concurrent programs. ACT++ is an implementation of the actor model in C++, allowing concurrent programs to be written in an object-oriented style. In ACT++, each actor is an object possessing one or more independent threads of control. Version 2.0 of ACT++ uses the PRESTO threads package. As PRESTO threads are available only for certain architectures and operating systems, its use does not meet one of the goals of ACT++, namely portability among a variety of architectures. To facilitate portability, ACT++ 3.0 is written using the IEEE POSIX 1003.4a standard for threads (Pthreads). This project deals with the implementation of ACT++ 3.0, the testing of the implementation, and its performance.

ACKNOWLEDGEMENTS

I would first like to thank my advisor and committee chairman Dr. Dennis Kafura for his guidance and patience throughout the course of this project. I would also like to thank Dr. James Arthur and Dr. Scott Midkiff for serving on my committee, and also for their comments and suggestions pertaining to my report.

I would also like to thank the other members of the Actor project for their help, inspiration, and camraderie. Specifically, I would like to Rajiv Gandhi for his help in figuring out grad school in general and for his help in integrating some parts of the distributed system. I would also like to thank Jae-Woong Hwang for his help in building the shared-memory version of ACT++ 3.0 for general distribution. I owe Venkateswara Vykunta a debt of gratitude for his support during the entire defense process. Thanks also to Siva Challa for his advice on the presentation. Though not a member of the project, I would also like to thank Kaushal Dalal for his help in getting through my problems with LaTex.

Last, but certainly not least, I would like to thank my family for their inspiration and encouragement during all my academic years. Without them, I doubt I would have ever set my goals to come this far.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The actor model is a framework for concurrent computation, first developed by Hewitt, et al., [HBS73] and later refined by Agha [AGH86]. ACT++ 3.0 is an implementation of the actor model in C++ [STR91], using the IEEE POSIX 1003.4a standard for threads, referred to hereafter as Pthreads [IEE92]. By using an object-oriented language, ACT++ 3.0 promotes software reusability and models real-world entities better than other languages [KAL90]. The use of lightweight threads allows for better utilization of resources, as compared to multiple processes.

As shown in Figure 1.1, the implementation of ACT++ 3.0 involved both a shared memory and a distributed environment. This project involved the writing the core ACT++ 3.0 classes, which are used in both the shared memory and distributed versions. These core classes make up a significant part of the shared memory version and, to a lesser degree, a part of the distributed version.

While the new version of ACT++ is a derivative of ACT++ 2.0, it differs in significant ways. First, version 2.0 is based on the PRESTO threads package [BER88], [BER90], which greatly limits its portability. This version, based on Pthreads, provides greater transparent portability of the implementation. The differences in the implementation of the two versions comes from two sources. First, PRESTO threads and Pthreads do not offer the same set of primitives, both in terms of availability and use. Second, assumptions made in PRESTO threads are not always valid when using Pthreads and vice versa. These are explained further in Chapters 3 and 4. Another difference in versions is based on design. Version 2.0 does not offer strong static type checking, greatly degrading its "usability." Nor is it designed be extendable to a distributed version. ACT++ 3.0 achieves both static type

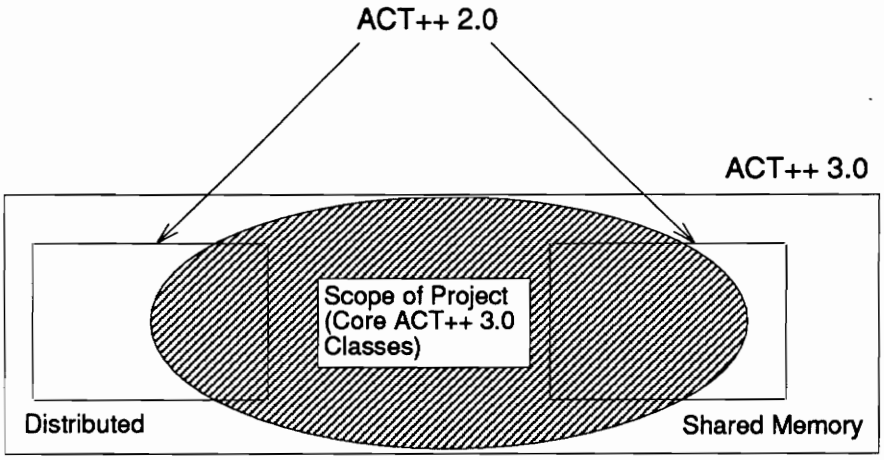


Figure 1.1: Project scope.

checking and is extendable to a distributed version.

The remainder of this report is organized as follows. Chapter 2 gives background for the current work. This includes an explanation of the actor model, motivations behind ACT++ 3.0, and finally the design of ACT++ 3.0, a separate issue from implementation. Chapters 3, 4, and 5 describe the contributions of this project. Chapter 3 presents a user's view of ACT++ 3.0, while Chapter 4 is an implementation view. Chapter 5 covers the stress testing of the system and the results of some performance tests. Chapter 6 covers future work and presents conclusions.

Chapter 2

Background

This chapter gives background for the design and implementation of ACT++ 3.0. The first section discusses the actor model and how ACT++ 3.0 relates to it. The second section covers the motivations behind ACT++ 3.0 and the third section gives an explanation of the design of ACT++ 3.0.

2.1 The Actor Model

The actor model is a framework for concurrent computation, where communication takes place via message passing. The actor model was first introduced by Hewitt et al., [HBS73] and later refined by Agha [AGH86]. The actor model is made up of three basic objects: actors, behaviors, and messages.

Actors: An actor can be defined as a "computational agent" [AGH86]. An actor contains a mail queue of *messages*. The mail queue has a unique mail address and also uniquely identifies the actor. The actor also has a *behavior* associated with it, which processes the messages in its mail queue. The mail queue itself is regarded as infinite and messages are processed in the order they arrive. Message delivery is considered reliable, though message delivery time is unbounded and the ordering of messages is not guaranteed. Figure 2.1 [AGH86] gives an example of an actor.

Actors are also able to send messages to other known actors and create new actors. Other known actors are referred to as *acquaintances*, and are known by their mail addresses. However, the acquaintance relationship is one-way; an actor j may not know any other actors i, even when all other actors i may know j. Upon creation of a new actor, the

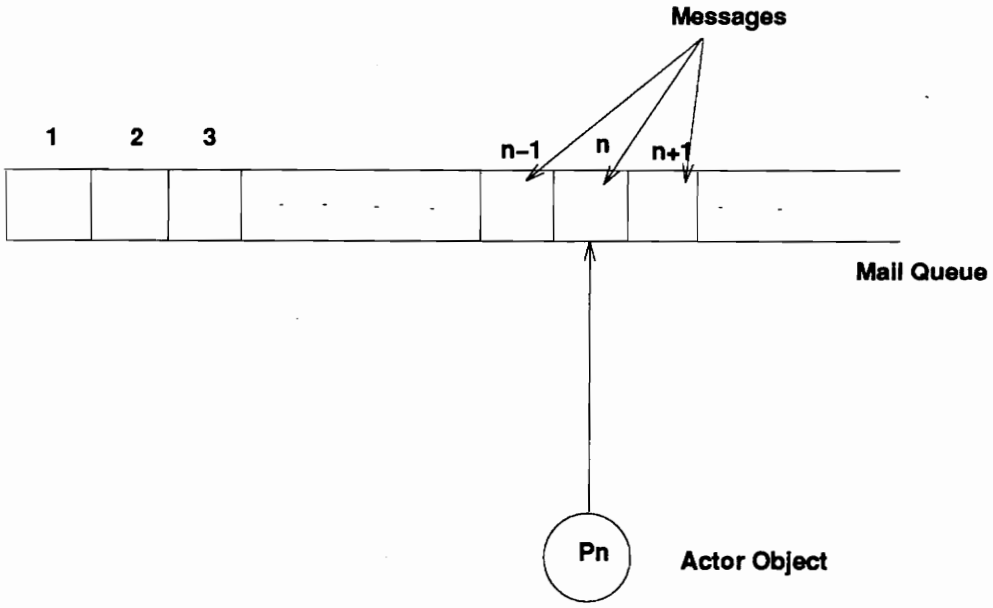


Figure 2.1: The actor object.

creating actor knows the created actor's mail address and can then distribute this name to other acquaintances through messages. In ACT++ 3.0, actors are implemented as a class.

Behaviors: Behaviors process messages in the actor mail queue. Each actor has one or more behaviors associated with it at any time, exactly one of which is the "current" behavior that processes the next message in the mail queue. Each instance of a behavior can only process one message. Also, a behavior can only perform certain operations, and in this way dictates the actions of the actor. At some point during the processing of a message, the behavior must specify its *replacement behavior*. This replacement behavior processes the next message in the actor's mail queue. The specifying of the replacement behavior is done via the *become* operation. It is possible for the behavior to *become* either itself, a new behavior, or nothing. In the case of becoming itself, the same behavior processes the next message, whereas in the second case, a different behavior processes the next message. In the third case, where a behavior becomes nothing, the actor terminates and stops performing

computations. By having to specify the replacement behavior after each message, it is possible to dynamically change the "state" of an actor, since the actions possible by an actor change with the behavior. When the mail queue is empty, the current behavior blocks execution, until a message is received. In ACT++ 3.0, a behavior is implemented as a class, and the class methods are the operations a behavior can perform.

Messages: Messages are the communication medium of the actor model. Messages contain two pieces of information: 1) the function to be performed by an actor's behavior, and 2) any arguments the behavior needs to execute that function. The number of arguments can vary, depending on the function to be executed, and is not restricted by the actor model. Messages are sent from one actor to another and message passing itself is asynchronous. A message sent to an actor is not acted upon until there is a behavior within that actor available to process it. The reader should note that a behavior in ACT++ 3.0, implemented as a class, can only perform operations for which it has corresponding methods. This implies that messages are meant to be sent to an actor only if they can be processed by the actor's current behavior. An "unused" message is put into the actor's mail queue and retrieved once a behavior is available. In ACT++ 3.0, a message is implemented as a class.

Putting it All Together: Figure 2.2 provides an example of the three components working together. Behavior *Pn* is processing message *n*, and then *becoming* behavior *Pn+1*, which in turn starts to process message *n+1*. Behavior *Pn* creates actor *Q*, with an initial behavior *Q1*, which then processes the first message in *Q's* mail queue. Behavior *Pn* also produces two messages which are sent to other known actors (acquaintances).

A behavior processes a message through the use of a thread. Since the functions executed by a behavior have a code size and life-time on the order of a procedure [KAL90], a lightweight process, such as a thread, is more efficient than a heavyweight process. By using threads, concurrent execution can be achieved within an actor. When a behavior does a *become* operation before it is done processing its current message and there exists another

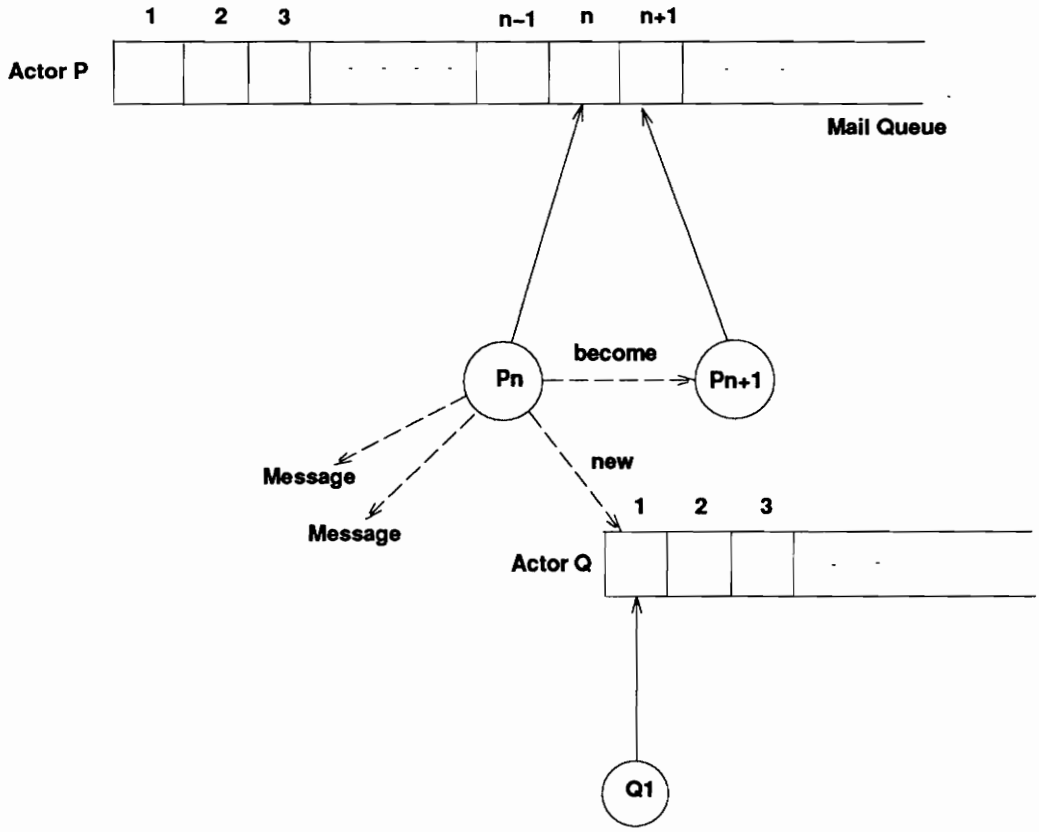


Figure 2.2: The three components of the actor model.

message to be processed, a second thread is created and started. This second thread now runs concurrently with the first thread. Concurrency at the system level can be seen in the case of multiple actors, where each actor has at least one thread of execution running at any given time.

Figure 2.3 shows an example of how a system of multiple actors could communicate. In the figure, an arrow from actor i to actor j means that i knows the mail address of j, where j is an acquaintance of i, but does not imply that j knows i's mail address. Recall that it is possible for actors to dynamically discover each other's mail address. This would be the result of an actor's mail address being passed in a message.

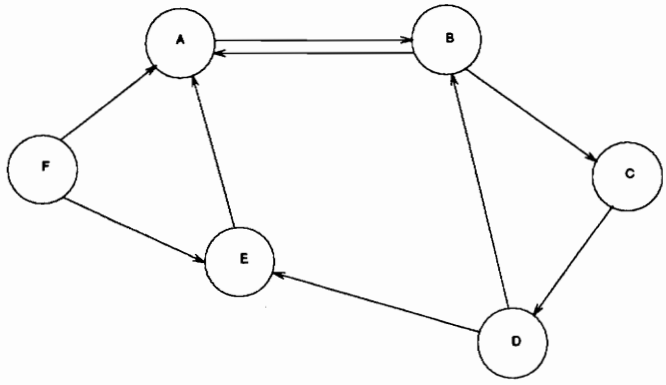


Figure 2.3: Topography of multiple actors.

Differences Between the Model and the Implementation: While ACT++ 3.0 conforms to the actor model as described above, certain aspects of the original model are not implemented, either because of the usage of C++, or because of the implementation itself. The following are the major deviations of ACT++ 3.0 from the original actor model [MUK92].

- The ACT++ 3.0 model has a coarser level of granularity in terms of concurrency. The original model contained expressions and subexpressions that could be run concurrently, but due to the use of C++, this is not possible in our implementation. A behavior is the finest grain of concurrency in ACT++ 3.0 (a behavior is sequentially executed). Concurrency, as mentioned before, is due to multiple threads in an actor, and multiple actors in a system.

- Actors in ACT++ 3.0 are not the only objects. Objects of any type are available for use in an ACT++ 3.0 program.

- The mail queue which uniquely identifies an actor is actually hidden. In ACT++ 3.0, we identify actors through an Acquaintace class, which is different than an object acquaintance described earlier. An acquaintance in ACT++ 3.0 merely has a pointer to an actor object. In fact, an actor is not directly accessible at all in this version.

- Reply messages are handled through an object called a Cbox. In the original model, an "insensitive actor" was used. Cboxes are defined to block a behavior when it tries to retrieve a reply message from an empty Cbox. Upon the reply message being placed in a Cbox, the behavior is unblocked, and allowed to retrieve the reply and continue.

- Messages in the shared memory version of ACT++ 3.0 do not arrive non-deterministically. Messages are received in the order they are sent.

2.2 Motivations for ACT++ 3.0

ACT++ 2.0 [MUK92] provided an object-oriented framework for concurrent programming. However, while realizing the general goal, the implementation fell short in some key areas described below.

- The use of PRESTO threads prevented version 2.0 from being portable among different machine architectures. Since threads are started off in functions, which may require different numbers of arguments, it is necessary to be able to collect the arguments into a list. Previously, this could be done by simply pulling the arguments from the stack, but in RISC machines arguments do not always appear on the stack, but may instead be located in registers. PRESTO threads solved this problem by tailoring their code to specific operating system versions on specific machine architectures, down to the assembly code level, thereby limiting the portability of version 2.0.

- PRESTO threads do not offer per-thread input/output (I/O) blocking. Instead, if one thread blocks on I/O, the entire *process* blocks. ACT++ 2.0 avoided process-level blocking by implementing an asynchronous I/O manager. The I/O manager only solves part of the problem, however, since process level blocking still occurs in the case of I/O with disk files. Furthermore, the I/O manager implementation is a significant part of version 2.0 and greatly increases its complexity.

- Since PRESTO threads are neither *native* to an operating system nor kernel supported, but instead a "package" on top of the operating system, they suffer in terms of efficiency and performance.

- The latest version of PRESTO threads uses a compiler which does not support C++ templates, restricting implementation flexibility of ACT++ 2.0, as well as programming flexibility for the user.

- PRESTO threads provide a method for an unknown number of arguments, each of an unknown type, to be passed to a procedure. This ability is used by ACT++ 2.0. Unfortunately, this method prevents static type checking, degrading the "usability" of ACT++ 2.0.

- Due to the lack of portability of PRESTO threads, ACT++ 2.0 is not suited for a user-transparent extension to a distributed environment.

By using Pthreads, we can overcome the first four limitations. Pthreads are an IEEE standard, implemented by major vendors. This makes it much more portable than PRESTO threads, both in terms of the number of platforms to run on as well as transparency to its actual implementation on different architectures. The Pthreads standard also requires per-thread I/O blocking, completely eliminating the need for an I/O manager in version 3.0. This significantly reduces the complexity of ACT++ 3.0 and further supports portability, as per-thread I/O blocking is provided by the Pthreads implementation. Since some vendors offer kernel-level thread support, and since Pthreads are implemented by the same vendors, it is possible that some implementations of Pthreads may be more efficient and have better performance. Lastly, the implementation of Pthreads is totally separate from that of ACT++ 3.0, whereas the implementation of ACT++ 2.0 merged code with that of PRESTO threads. By keeping the two parts separate, ACT++ 3.0 is not dependent on or restricted by the Pthreads implementation.

To solve the last two problems, a design different from that of ACT++ 2.0 is used for ACT++ 3.0. This new design solves both the static type checking problem and the trans-

parent extensibility issue. The manner in which the design solves these last two problems is discussed in the next section.

2.3 Design of ACT++ 3.0

The design of ACT++ 3.0 is different than that of ACT++ 2.0, which results in two advantages over the prior version: ACT++ 3.0 has the 1) ability to perform static type checking, and 2) when extended to a distributed version, ACT++ 3.0 helps provide a transparent interface to the user. This section explains how these two goals are met through the new design.

2.3.1 Static Type Checking

The Problem

When starting a thread, the customary procedure call includes two arguments: 1) the procedure in which the thread is to start executing, and 2) any arguments needed by that procedure. This presents a problem, since the number of arguments depends on what procedure the thread starts in. PRESTO threads' solution to this is to offer the PFany type, which allows the next parameter in a list to be unspecified, as shown below.

```
virtual int start(Objany obj, PFany pf, ...)
```

This is the PRESTO threads call to start a thread. The parameter *obj* refers to the object in which the thread is to start (PRESTO threads are based on C++), *pf* refers to the method in the object to execute, and "..." refers to the argument list. The notation "..." implies that an "unknown" number arguments follow.

ACT++ 2.0 does not exploit the PFany type for starting threads, but instead uses the PFany type in defining the message class. As discussed earlier, messages consist of the behavior method to be performed and the argument list for that method. The PFany type is used in the message constructor and has the following form and use [MUK92].

```
Message::Message(PFany, ...);
```

```
Message* new_mess = new Message((PFany)&beh_method, int arg1, char arg2);
```

This solution offers the advantage that the Message class constructor can now take any behavior method with any number of arguments, thus offering great flexibility, with little implementation overhead.

This solution works because the compiler does not check the arguments after the PFany type variable. The great disadvantage is that static type checking is not performed. For the user, this means that mistakes in a program may not be found until run-time. From a software engineering point of view, this is obviously undesirable, if not dangerous, since user errors will not be detected until run-time and will often result in abnormal termination of the computation.

The Solution

Pthreads does not offer anything analagous to the PFany type in PRESTO threads. This poses the problem of how to specify the argument list for a procedure when constructing a message. Obviously, the message class cannot specify a constructor for every possible case. Instead, ACT++ 3.0 provides an interface class for behaviors, which encapsulates the argument list for behavior methods and preserves static type checking of the arguments.

For each behavior the program uses, an interface class is developed. The interface class is generated by a translator that is part of ACT++3.0. This translator takes as input the prototypes for the behavior methods, described in the Interface Definition Language (IDL) [IDL88]. IDL is a standard which provides an interface between different programming languages. The generated interface is actually a template class, which takes the behavior from which it is derived as its template argument. Below is an example of the information used to produce an interface template, the interface template, and the corresponding user-defined behavior class.

The information from which the interface template is generated is the following.

```
interface Stack {
void push(in int i);
void pop(in Cbox<int>& c);
void Create(in int i);
};
```

Since each interface template generated follows the exact same structure, only the method names and argument lists are needed.

The behavior interface template generated by the translator from the information above is given below.

```
template<class T> class Stack {
  private:
    class StackMESSAGE {
       class push : public Message {
          int i;
       public:
          push (int i): i(i) {}
          ~push() {}
          void invoke(Behavior* beh) { ((T*) beh)->push(i); }
       };
       class pop : public Message {
          Cbox<int> c;
       public:
          pop(Cbox<int>& c): c(c) {}
          ~pop() {}
          void invoke(Behavior* beh) { ((T*) beh)->pop(c); }
       };
       class Create : public Constructor {
```

```
        int i;
      public:
        Create (int i): i(i),Constructor("Stack") {}
        Behavior* New() { return (Behavior *) new T(i); }
      };
    } ; // StackMESSAGE
  public:
    Message*     push(int i) { return new StackMESSAGE::push(i); }
    Message*     pop(Cbox<int>& c) { return new StackMESSAGE::pop(c); }
    Constructor* Create(int i) { return new StackMESSAGE::Create(i); }
};
```

The corresponding user-defined behavior class follows.

```
class Stackops : public  Behavior {
protected:
  int stack[100];
  int top;
  int size;
public:
  Stackops(int sz) : size(sz) {
    top = -1;
  }
  void push(int v) { /*push v onto stack*/
    stack[++top]=v;
    become(this);
  }
  void pop(Cbox<int>& c) {/*get top of stack and*/
    c = stack[top--];     /*put into the cbox c */
```

```
    become(this);
  }
};
```

The manner in which the interface template preserves type checking is as follows. When creating a message in ACT++ 3.0, a variable must be used which utilizes the behavior interface template.

```
Stack<Stackops>  aStack;
```

This variable is then used in the construction of a message as shown below.

```
Message* pushMsg1 = aStack.push(1);
```

The call to *aStack.push()* calls the public *push()* operation in the template class. The *push()* creates an object of type *Stack<Stackops>::StackMESSAGE::push* by calling its constructor with the argument *int i*. In the constructor, the input argument *i* is copied to a local private variable in the *Stack<Stackops>::StackMESSAGE::push* class. All input arguments are copied this way, leading to the *Stack<Stackops>::StackMESSAGE::push* object encapsulating all arguments needed for the behavior method. Since the push class is derived from the Message class, the *Stack<Stackops>::StackMESSAGE::push* object pointer returned can be cast to be a Message pointer. The Message class is used to provide each message with a set of inherited methods that are used by ACT++ 3.0 to process the message.

When a thread is to be started, the message's invoke method is then called as follows.

```
Behavior *b; /*behavior object that will process this message*/
             /*b is a pointer to Stackops*/

pushMsg->invoke(b);  /*pushMsg1 is message from above*/
```

This is the same as the *pushMsg1->Stack<Stackops>::StackMESSAGE::push::invoke()* method, which calls *(Stackops *)b->push(i)*. The value *i* here is the variable that is private

to the *Stack<Stackops>::StackMESSAGE::push* class. Note that all variables that were encapsulated in the message object have now been passed to the appropriate behavior method. Also note that the message object's invoke() method calls a specific behavior method, and thus encapsulates the call to the appropriate behavior method also.

Earlier we stated that it is not possible to specify a message constructor for every possibility, but by producing the interface template, we in fact do specify a constructor for the needed messages. In fact, the Message class itself does not have a defined constructor method. In this way, static type checking is preserved.

The Create method in the interface template is used to give an actor an initial behavior when creating that actor.

```
Acquaintance* stackActor = new Acquaintance (Stack.Create(100));
/*Will create acquaintance and actor it points to*/
```

An object of type *Stack<Stackops>::StackMESSAGE::Create* is returned by a call to *aStack.Create()*. Any input arguments are copied to local variables private to the object. Since the *Create* class is derived from the *Constructor* class, the returned pointer is cast as a pointer to a Constructor. The Constructor class provides a generic name through which an object returned from the Create method can be accessed. (For clarity, we define *ctor* as a pointer to the Constructor object.) The construction of an acquaintance forces the construction of an actor and the actor is passed the Constructor pointer *ctor*. In the actor class's constructor method, the actor's initial behavior is set as follows.

```
(Behavior *)actor_initial_behavior = (Constructor *)ctor->New();
```

The call to *ctor->New()* is actually a call to *Stack<Stackops>::StackMESSAGE::New()*, and returns a pointer to desired initial behavior.

The interface template also allows ACT++ 3.0 to circumvent a restriction imposed by Pthreads. The Pthreads call to create and start a thread takes only the procedure to be executed and only *one* argument for that procedure as input parameters. In the case

of ACT++ 3.0, the procedure to be executed will be a behavior class method, and may require more than one argument. Furthermore, C++ requires the method passed to the thread creation call be statically defined. This set of circumstances presents the following problems.

- Obviously, there may be more than one argument to pass to the behavior method.
- To start a thread, the thread must not only know the behavior method it must execute, but also in which specific behavior object to execute. The thread creation call does not allow for specifying the object in which the thread will execute.
- A statically defined method is independent of any information in an object of that class [STR91]. This negates the whole purpose of having a thread execute within a behavior object, as the behavior object may have a context within which the thread should execute. Furthermore, having the user define all behavior methods as static is not good software engineering practice, since it does not allow the user to program freely and be unaware of the implementation of ACT++ 3.0.

The use of the interface template solves these problems naturally. To solve the third problem, a static method was created in the Message class (*Message::run*). It alone is passed as the static method to the thread creation call. The one argument passed to the thread creation call is the message object produced by the interface template, i.e., pushMsg1 in the earlier example. This solves the first problem, since the message object already encapsulates all the arguments needed for the behavior method. The message object also knows which behavior method is to be executed, so it solves the problem of specifying the behavior method to execute.

Figure 2.4 shows how the second problem, determining the behavior object in which to execute, is solved at thread creation time. Step 1 shows *pushMsg1*, produced by an interface template, encapsulating the behavior method call and the behavior method arguments. Since the inherited Message class allows a behavior object to be bound to a message object,

step 2 shows *pushMsg1* being bound to behavior object *stack_beh*. This now lets *pushMsg1* access the behavior object also. In step 3, *Thread 2* is created to start in Message::run, with *pushMsg1* as its argument. In step four, Message::run() parses the input message object, and executes the thread in the correct behavior object and method.

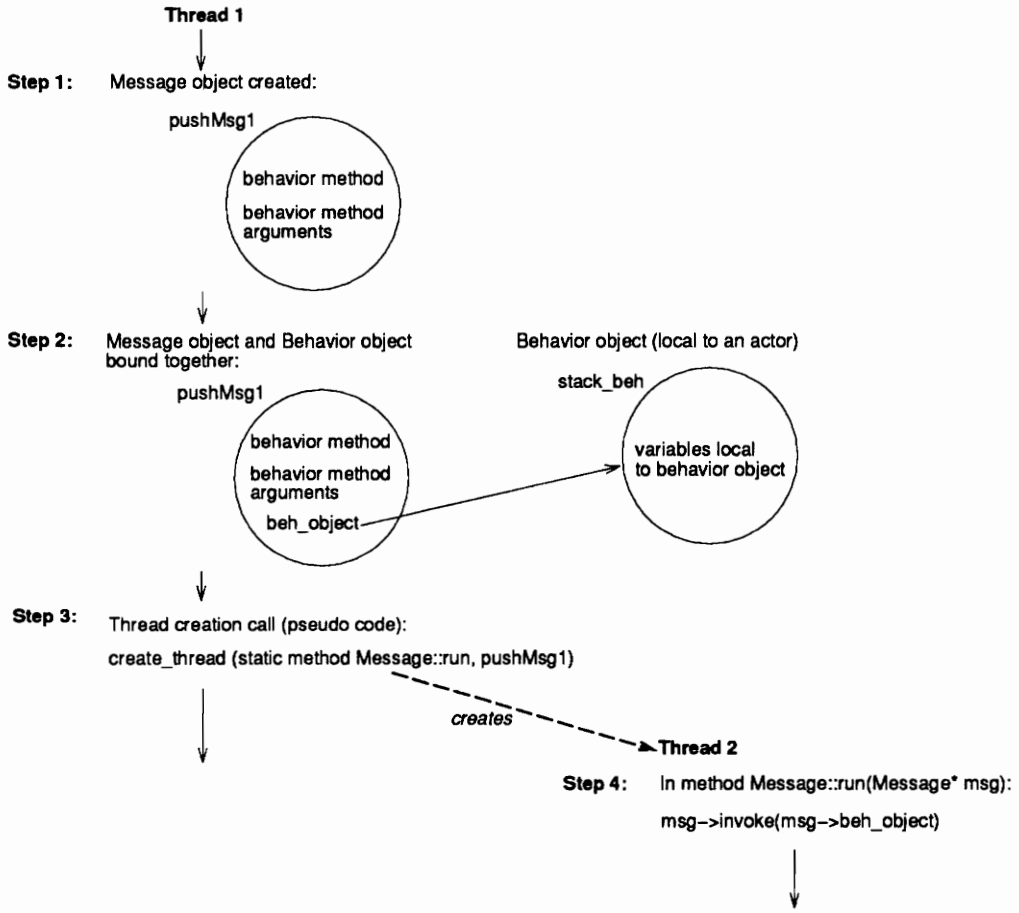


Figure 2.4: Thread creation and encapsulation with pthreads.

To demonstrate that Message::run() does in fact call the correct behavior method in the correct object, the msg−>invoke() call in step four can be traced back by substitution as follows.

```
msg->invoke(msg->beh_object)
```

```
.
pushMsg1->invoke(msg->beh_object)
.
pushMsg1->invoke(stack_beh)
.
pushMsg1->Stack<Stackops>::StackMESSAGE::push::invoke(stack_beh)
```

This in turn executes the following.

```
stack_beh->push(i);
```

This is the desired result. The argument *i* is a local variable in *pushMsg1*, defined when *pushMsg1* was created.

2.3.2 Transparency to a Distributed Version

One of the goals of ACT++ 3.0 is user-transparency when distributed. This means that when ACT++ 3.0 is running in a distributed environment, the user should be able to write his/her programs without, for example, knowing on which machine an actor is located in order to send that actor a message. This transparency is achieved through the use of acquaintances.

Acquaintances in the shared memory version of ACT++ 3.0 are merely pointers to actors. However, in the case of a distributed environment, acquaintances provide an important level of indirection. The acquaintance class keeps track of where the actor is and, when the client wants to send the actor a message, the acquaintance transparently routes the message to the remote actor. In ACT++ 3.0, all operations on actors are done via acquaintances. In this way, the user is oblivious to the physical location of the actor.

It should be noted that in the distributed version, there is transparency in *use*, but not in *creation*. The call to create an acquaintance and actor, in the distributed version is:

```
Acquaintance* stackActor = new Acquaintance ( ''node name'', Stack.Create(100));
```

```
/*Will create acquaintance and actor it points to on node specified*/
```

The node on which the actor is to be created is specified at creation, but as mentioned, the acquaintance will from then on transparently provide access to the actor during use.

Chapter 3

User's View of ACT++ 3.0

ACT++ 3.0 implements the major parts of the actor model through the use of classes. These classes provide a framework within which the user can develop actor-based programs. This chapter explains the concepts behind these classes and illustrates their use.

3.1 ACT++ 3.0 Class Concepts

The four classes available to the user are Acquaintances/Actors, Behaviors, Messages, and Cboxes. This section explains how these classes conceptually relate to the actor model and also give insight into how ACT++ 3.0 programs work.

3.1.1 The Acquaintance and Actor Classes

ACT++ 3.0 has been designed such that it is easily extensible to a distributed system. In a distributed system, the physical location of an actor, i.e., the machine where it executes, may not be the local machine. Recall that the *address* of an actor can be passed to other actors. Normally this would then entail that the user's program keep track of which machine the actor is located on. ACT++ 3.0 elimintates this by having an *Acquaintance* class which acts as a handle to the Actor class. Each acquaintance refers to only one actor, though an actor may have many acquaintances. All operations to be performed on an actor are done through an associated acquaintance.

Figure 3.1 shows the utility of the Acquaintance class in a distributed system in the case where the address of an actor is passed to other actors. After having distributed *acquaintance_object*, which is the handle for *actor_object*, the command to send a message

to *actor_object* is the same on each of the three machines. The senders of the messages do not need to know where *actor_object* is located, since *acquaintance_object* stores that information internally and, consequently, routes the information automatically.

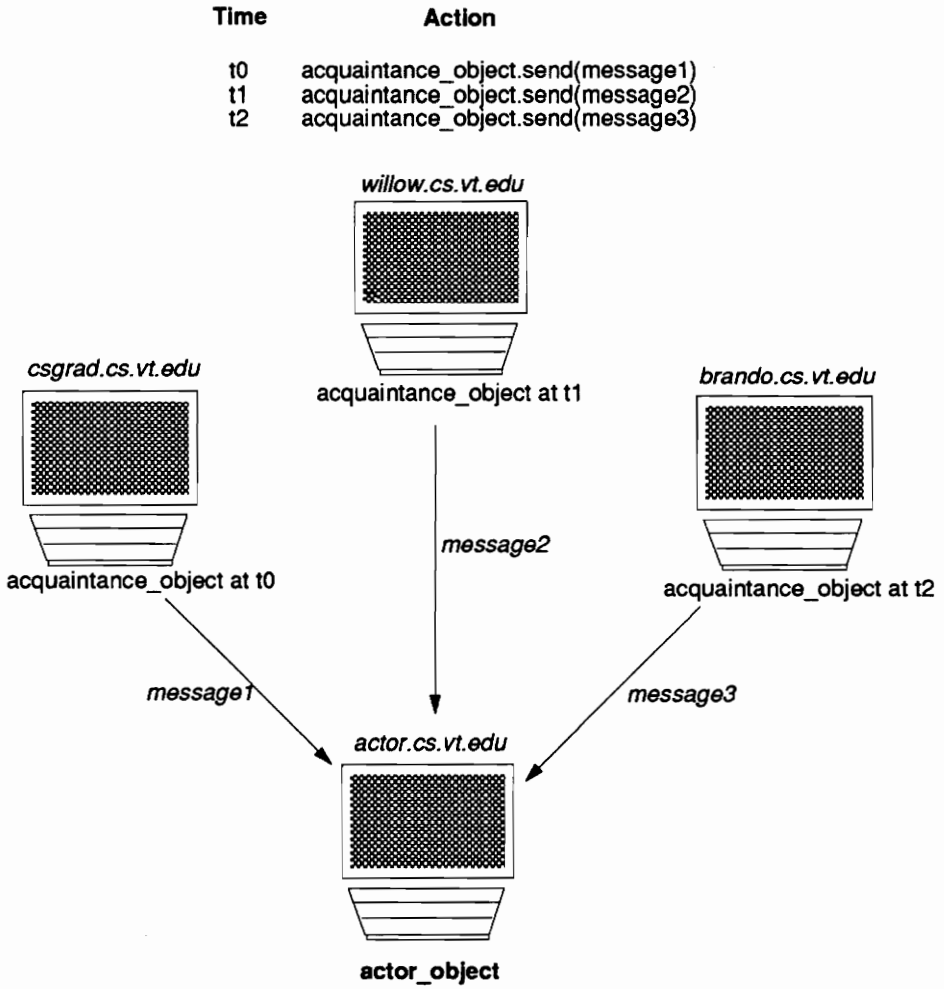


Figure 3.1: Acquaintance and actor relation.

Another aspect of the Acquaintance/Actor classes is the ability to specify the number of threads allowed to run concurrently within an actor. While the default is one thread, the user is allowed to specify more.

In the shared memory version of ACT++ 3.0, the acquaintance class merely adds a level

of indirection, but is kept to provide a foundation for the distribued version.

3.1.2 The Behavior Class

In ACT++ 3.0, a behavior is implemented by the user as a C++ class and is defined to be a derived class of the ACT++ 3.0 *Behavior* base class. The computations that a user-defined behavior can perform are defined by its public methods. Execution of a method is done via a lightweight process or a *thread.* Specification of the *replacement behavior* is done by the *become* operation.

In ACT++ 3.0, a behavior can specify its replacement behavior to be itself, another behavior, or no behavior. In the case of specifying itself as the replacement behavior, the same behavior object will process the actor's next message, whereas in the second case, a new behavior object will process the next message. In the third case, the actor the behavior is associated with terminates and stops processing messages. A behavior can specify its replacement behavior at any time during its execution.

By using threads and varying when a replacement behavior is specified and what replacement behavior is specified, six different cases of concurrency can occur [MUK92], shown in Figure 3.2.

When Replacement Behavior Specifed During Execution

What Behavior Specified		**Start**	**Middle**	**End**
	Itself	Behavior–level	Behavior–level	Serial
	New Behavior	Actor – level	Actor – level	Serial

Figure 3.2: Concurrency in an actor.

Actor-level denotes that there are multiple threads running concurrently within the actor, but in different behaviors. *Behavior-level* means that there are multiple threads

executing concurrently within the same behavior object (and consequently in the actor also). *Serial* denotes that there is only one thread executing within the behavior and actor at a time. When there is concurrency at the behavior-level, it is necessary that the user-defined behavior class provide the necessary mutual exclusion for any critical sections in its methods.

Behavior sets, used in ACT++ 2.0, have not yet been implemented in ACT++ 3.0.

3.1.3 The Message Class

Messages in ACT++ 3.0 are used to encapsulate the behavior method to execute and any arguments needed by the behavior method. A message object is created through the use of a behavior interface class (Section 2.3), and is passed to an actor by its acquaintance. A message object of this kind inherits methods from the Message class which are used by ACT++ 3.0 to process the message. Message objects in ACT++ 3.0 are not copied when sent to actors, but are instead passed by pointer. Since message objects are destructed after processing, the same message object cannot be re-used or sent to multiple actors. As noted earlier, a major change in messages from version 2.0 is that the arguments they encapsulate are statically type checked against the behavior method to be executed.

3.1.4 The Cbox Class

In the actor model [AGH86], *insensitive actors* were created as way for reply information to be returned to a requesting actor. However, the implementation of insensitive actors was found to be too costly, and so *cboxes* were created to handle return messages [KAF93]. Cboxes are implemented as a template class. The argument for the template is the data type for the requested return value. Once a cbox is specified, a behavior can attempt to read the return data from the cbox. If the data is not yet available, the behavior will *block*, until the data is written to the cbox (Figure 3.3). This not only provides a way for data to be returned, but also provides a way to synchronize the execution of multiple behaviors.

Cboxes in ACT++ 3.0 also have certain properties. Cboxes are created "empty".

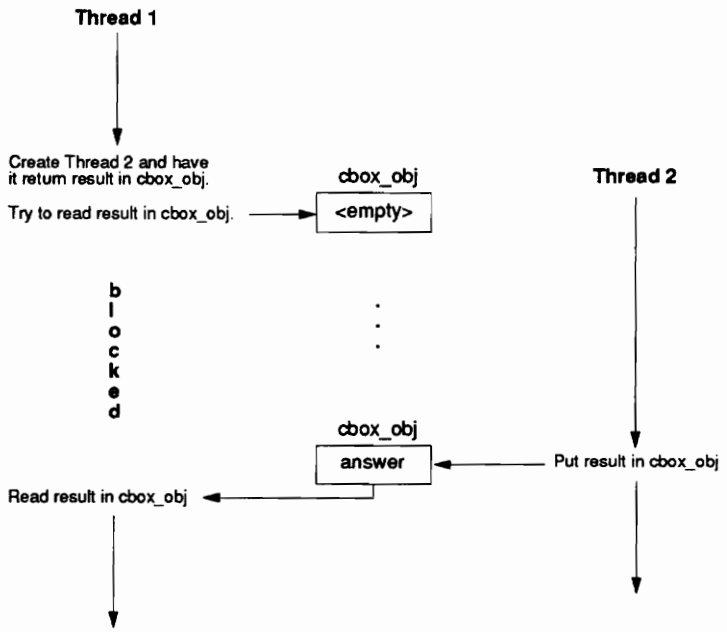


Figure 3.3: Using cboxes for return data.

Cboxes can be written to only once, but can be read from many times. This implies that multiple behaviors could block on a cbox until it was written to. Also, once a cbox is written to, any behavior possessing a reference to the cbox can read from it. Cboxes cannot be reset, meaning that once a cbox is written to, there is no way to empty the cbox. Lastly, the shared memory version reference counts cboxes. This means that ACT++ 3.0 keeps an active count of the number of threads that have a reference to a cbox and that the cbox is only destroyed once the reference count is zero. The distributed version does not yet reference count cboxes.

3.2 ACT++ 3.0 Class Descriptions

This section discusses the methods available to the four classes mentioned above and describes how they can be used in an ACT++ 3.0 program.

3.2.1 The Acquaintance and Actor Classes

As mentioned previously, acquaintances in ACT++ 3.0 provide handles to actors. Actions to be performed on actors are done through acquaintances, including creation. Two methods are available in the Acquaintance class.

The first method is the constructor for an acquaintance. The method interface is defined as:

```
Acquaintance::Acquaintance(Constructor* ctor,int = 1)
```

Creating an acquaintance automatically creates an actor for which the acquaintance is a handle. The Constructor class pointer, *ctor*, encapsulates the actor's initial behavior. The second argument specifies the maximum number of concurrently running threads that the actor can have. The call to construct an acquaintance is illustrated in the following example.

```
Stack<Stackops> aStack;  /*the interface template object*/
Acquaintance *new_acquaintance;

new_acquaintance = new Acquaintance(aStack.create(100),4);
```

This call constructs an acquaintace and an actor, where the actor's initial behavior is specified by the interface template's *Create* method (Section 2.3), and is allowed to have up to four concurrently running threads. Note that the interface template's Create method is passed as a pointer to a Constructor.

The second method interface for the Acquaintance class is the *send* method:

```
void Acquaintance::send(Message* msg);
```

The send method is used to deliver a message to the acquaintance's actor. In the distributed version, this includes routing the message to the correct node in the system. Once the message is delivered to the actor, the actor checks to see if there is an available behavior to process it. If a behavior is available, the behavior and message are "bound" together and a new thread is started to process the message. If a behavior is not available, the message is put into the actor's mail queue. In the case where the actor is dead, the message is deleted.

3.2.2 The Behavior Class

The Behavior class is used as the base class for all user-defined behaviors. The Behavior base class provides methods to specify replacement behaviors and to send messages to their own actor.

The first method, the *become* method, is used for specifying a replacement behavior. It is an overloaded method and offers two signatures:

```
void Behavior::become(void);
void Behavior::become(Behavior* b);
```

The first signature does not specify a replacement behavior and causes the actor to *terminate*. Upon termination, an actor stops receiving and processing messages. Any messages currently in the actor's mail queue or that arrive in the future are ignored.

The second signature is used to specify a replacement behavior. The C++ *this* pointer [STR91] can be used to allow the current behavior to name itself as the replacement behavior as follows.

```
become(this);
```

When the replacement behavior is not the current behavior, the current behavior is *deleted* at the end of its execution. This can raise a dangerous situation when there is

behavior-level concurrency. Since concurrently running threads are not guaranteed to finish in any particular order, the scenario in Figure 3.4 could arise.

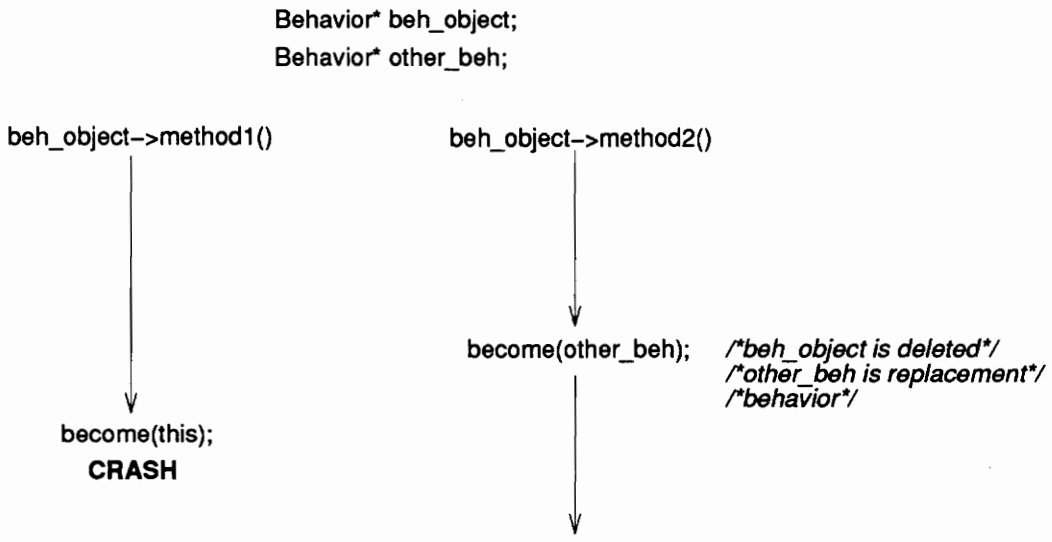


Figure 3.4: Synchronization with behaviors.

The *become(this)* operation in *beh_object->method1()* will cause the program to abnormally terminate, since *beh_object* has been deleted by the *become(other_obj)* operation in *beh_object->method2()*. Consequently, the *this* pointer now points to nothing. The user must provide the synchronization to avoid this situation.

The second method available from the Behavior class is the *send* method.

```
void Behavior::send(Message* msg);
```

This method is used for sending messages to a behavior's actor. The Behavior class in ACT++ 3.0 knows the actor with which it is associated, and so the send method implicitly knows to which actor the message should be delivered. In ACT++ 3.0, a behavior is always local to its actor, and therefore, there is no need to use the actor's acquaintance to send the message, even in the distributed version.

3.2.3 The Message Class

The Message class provides only a method for sending itself to an actor.

```
void Message::send(Acquaintance* acq);
```

```
Message* mesg;
mesg->send(Acquaintance* acq); /*send mesg to acq's actor*/
```

The send method has the message object send itself to an actor through the actor's acquaintance. This call operates in same manner as the Acquaintance::send() call explained earlier.

The construction of a message is not done explicitly through a Message class constructor. Instead, the behavior interface template (Section 2.3) is used to construct a message as given below.

```
Stack<Stackops> aStack;
```

```
Message* message = aStack.push(4);
```

All messages must be constructed in this manner since the interface template allows the C++ compiler to statically type check the arguments to be passed to the requested behavior method. In fact, the Message class itself does not have a defined constructor.

3.2.4 The Cbox Class

Cboxes in ACT++ 3.0 are defined as a template class. The template argument is not restricted. The template argument specifies the data type of the value to be written to and read from the cbox. The Cbox class is an interface to another class which actually implements the concept of a cbox. The cbox interface is needed to perform reference counting, so that cboxes can be deleted when no longer used. In the following discussion, a

"cbox" refers to the actual implementation of the cbox concept and a "Cbox object" refers to its interface. All actions to be performed on a cbox are done through the Cbox object. There are six Cbox object methods available. The first three methods are constructors and a destructor.

```
Cbox<T>::Cbox();
Cbox<T>::Cbox(Cbox<T>& c);
Cbox<T>::~Cbox();

Cbox<int> cbox_var;
Cbox<int> cbox_var(cbox2);
```

The first Cbox constructor creates an initially empty cbox of type T and the reference count of the cbox is set to one. The second constructor creates a Cbox object that accesses the same cbox as the passed-in Cbox object. The cbox reference count in this case is incremented by one. The destructor decrements the reference count of the cbox when the lifetime of the Cbox object ends. If the reference count is then zero, the cbox is also destructed.

The Cbox class also offers a method for an existing Cbox object to reference the same cbox referenced by another Cbox object by overloading the "=" operator.

```
void Cbox<T>::operator = (Cbox<T> v)
/*So can do myCbox_object = anotherCbox_object*/

Cbox<int> cbox_obj1;
Cbox<int> cbox_obj2;

at time t1:  cbox_obj1 = cbox_obj2;
at time t2:  cbox_obj1 = cbox_obj2;
```

Initially, *cbox1_obj1* and *cbox2_obj2* refer to two different cboxes. After the assignment operation at time t1, *cbox_obj1* refers to the same cbox as *cbox_obj2*. At time t1, ACT++ 3.0 increments *cbox_obj2's* cbox reference counter, decrements *cbox_obj1's* cbox reference counter, and destructs the cbox if the count becomes zero. However, at time t2, ACT++ 3.0 will not decrement or increment any counters since both Cbox objects refer to the same cbox.

The Cbox class is written such that Cbox objects which are passed by pointer, reference, or value reference the same cbox.

The last two methods in the Cbox class are used for writing to and reading from the cbox.

```
Cbox<int> cbox;

void Cbox<T>::operator = (T v);  /*write to cbox*/
cbox = 5;

void Cbox<T>::operator T();      /*read from cbox*/
int num = cbox;
```

The first method stores the value in the cbox, and unblocks any threads trying to read from the cbox. The second method allows the value in the cbox to be read. It will block the thread if the cbox is empty. ACT++ 3.0 does not enforce the write-once policy. However, threads will no longer block on a cbox that has been written to.

3.3 ACT++ 3.0 Guidelines and Utilities

Beyond understanding and using the classes, ACT++ 3.0 has some guidelines and utilities due to the use of Pthreads and the design of ACT++ 3.0.

3.3.1 Interface Template Generation

The first guideline concerns the creation of the behavior interface templates (Section 2.3). As was mentioned, an IDL [IDL88] translator has been built into ACT++ 3.0 which takes an IDL specification file and generates the appropriate behavior interface template. An IDL specification file must be built for each user-defined behavior class. Consider a user-defined behavior class with the following public method interfaces.

```
Stackops::Stackops(int size);
Stackops::push(int number);
Stackops::pop(Cbox<int>& c);
```

The format for the corresponding IDL file is defined as follows.

```
interface Stack {
        void push(in int size);
        void pop(in Cbox<int>& c);
        void Create(in int number);
};
```

The user-defined behavior's constructor method always corresponds to the Create call in the IDL file. All other behavior methods have a call defined in the IDL file the same as the relationship shown for the push and pop methods. The name of the generated interface template class is determined by the name of the interface in the IDL file:

```
interface Stack { } -> template<class>T class Stack { }
```

The behavior interface template file that is produced depends on the name of the IDL file:

```
IDL file ''stack.idl'' -> interface template file ''stack.h''.
```

3.3.2 The main() Procedure

When writing the main() procedure, the user must rename it *usermain()* with the following arguments.

```
usermain(int argc, char* argv[]);
```

The usermain() procedure should be written exactly the same as the main procedure, treating *argc* and *argv* like normal command line arguments. The only difference is the name of the procedure. The main() procedure is actually implemented in ACT++ 3.0, and calls usermain(). The main() procedure is internal to ACT++ 3.0 because when using Pthreads, if the the program's *main()* procedure is not dependent on the threads it creates, it can finish and exit, causing the process to exit, before any of the threads have had a chance to run. Unfortunately, the user is not informed if all threads were able to execute. The main() procedure is used in ACT++ 3.0 to let the user know whether all threads have run when the program exits. The main() procedure displays to the screen a message saying either the program "Finished execution normally," meaning all threads have run, or how many threads were not able to completely execute.

3.3.3 The Threadwait() Utility

The use of the main() procedure still poses a problem since the user may want to prevent the program from exiting before all threads have run. The *Threadwait()* utility can be used in the usermain() procedure to halt the usermain() process until all threads have run. The Threadwait() utility can be used multiple times within the usermain() procedure.

```
usermain(int argc, char* argv[])
{
    create thread 1; /*this thread creates many other threads*/
        .
    Threadwait();
```

```
        .
    create thread 2; /*this thread creates many other threads*/
        .
    Threadwait();
}
```

The above example, written in pseudo-code, demonstrates how Threadwait() can be used. At the first Threadwait(), the usermain() procedure stops and waits for *thread 1* and any other threads that may be created due to it to finish executing. Then, usermain() continues until it reaches the second Threadwait(), at which point it waits until *thread 2* and any threads due to it have finished execution. At that point, usermain() returns to main() internal to ACT++ 3.0, which notifies the user if any threads are still running and then exits the program. Used in this manner, the Threadwait() utility can be used as a crude means of synchronization.

Chapter 4

Implementation of ACT++ 3.0

This chapter discusses implementation issues in ACT++ 3.0. To provide a context for certain decisions that were made, the first two sections of this chapter describe the general method of thread creation and execution and the class relationships. The last two sections discuss each class individually.

4.1 Thread Creation and Execution

Recall that a thread is created to execute a behavior method, and that a thread is only created when there is both a behavior and a message available within an actor. There are three separate circumstances under which a new thread is created.

- First, if a thread/behavior sends a message to an actor, it checks to see if there is a behavior available in that actor to process that message and, if there is, a thread is created.

- Second, when the current thread/behavior does a *become* operation, a behavior becomes available. A check is made to see if the actor, with which this thread/behavior is associated, has a message to be processed. If there is a message, a thread is created.

- Third, when the final thread has finished execution, it checks to see if there is both a message and a behavior available in the actor and, if there is, a thread is created. The current thread then finishes its own execution and terminates. Because the actor may restrict the number of threads it can have currently running, an actor at its limit will not create any new threads, even if behaviors and messages are available. Since

the current thread is terminating, the number of threads in the actor will be reduced by one, and a new thread can take its place.

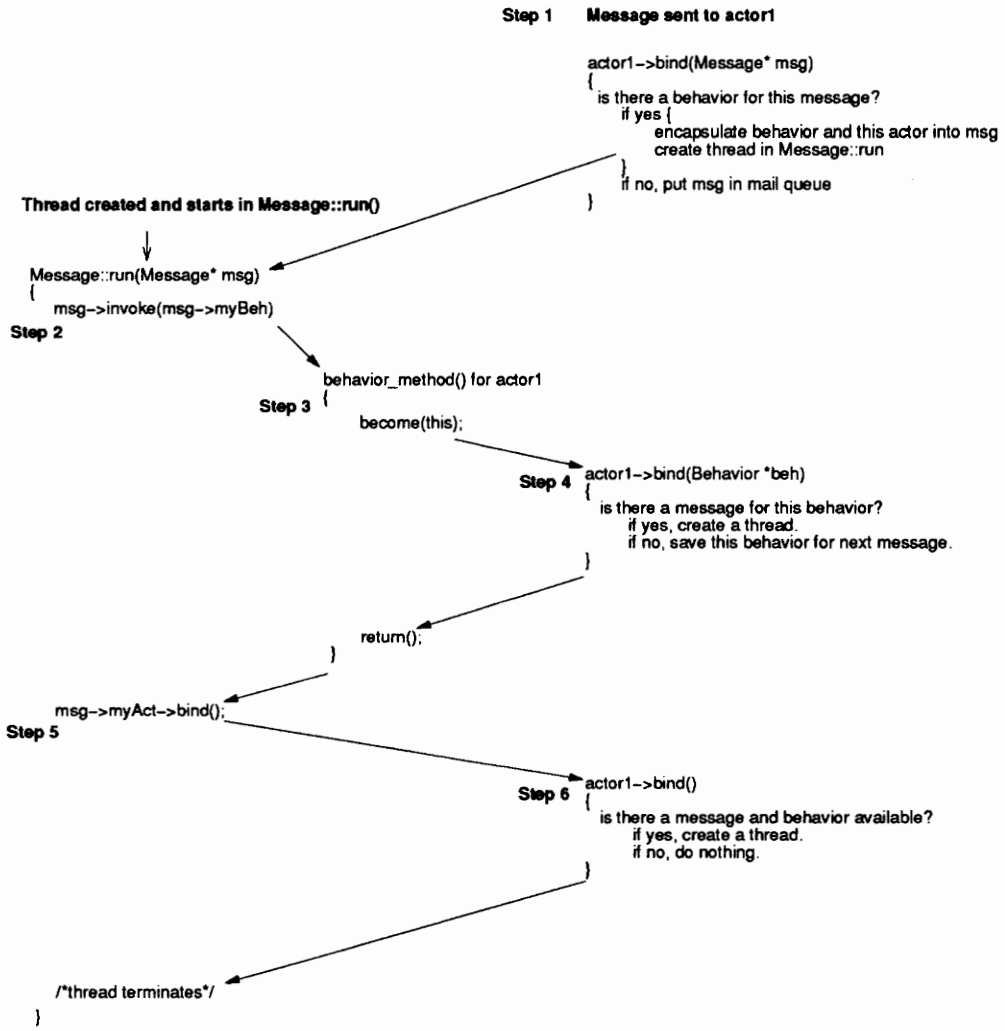


Figure 4.1: Life of a thread.

Figure 4.1 shows the life of a thread in ACT++ 3.0, and how its execution weaves its way through the Actor, Message, and Behavior classes. In step 1, the thread of interest is created when a message is sent to *actor1*. As was noted in Section 2.3, a thread is always started in the static Message::run() method. The passed message object *msg* encapsulates

not only the requested behavior method and arguments, but now also encapsulates the behavior object in which to execute, and the actor in which this thread will execute. In step 2, the invoke() method is called, which causes the the thread to execute the requested behavior method. The behavior method is executed in the correct behavior object because the behavior object was encapsulated in *msg* in step 1. In step 3, the behavior method at some point in its execution specifies its replacement behavior. The thread, now in step 4, goes to *actor1* and checks to see if there is a message available for the new behavior, and if there is, creates a new thread. The thread returns to the behavior method, and in step 5, finishes execution of the behavior method, and returns to Message::run(). At this point, the thread is ready to terminate, but first proceeds to step 6. In step 6, the thread checks to see if there is both a behavior and message available in *actor1*, and if there is, creates a new thread. Step 6 is done for the case where an actor has reached its limit of concurrent threads. Since the current thread is going to terminate, there is now room for another thread, and so this last check for a behavior and a message is done. The Message::run method knows which actor to check because *actor1* was encapsulated in *msg* in step 1. After step 6, the thread finally terminates. It should be noted that a thread executes a behavior method, but has a lifetime beyond the scope of the behavior method.

4.2 Class Relationships

The class relationships in ACT++ 3.0 can be viewed as a dependency relation. For example, the Actor class depends on the Lock class because the Lock class is independent of the other classes. A hierarchy based on this dependency relationship is shown in Figure 4.2. The arrows show which classes depend on others. Dependence is shown only if the dependent class uses the support class and if the support class is independent of the dependent class. Classes in italics are accessible by the user. The classes are further broken down into two groups. The first group, *conceptual classes*, directly relate to some component or concept of the actor model. The second group, *utility classes*, are created to help in the implementation

of the conceptual classes and do not relate to any concepts of the actor model.

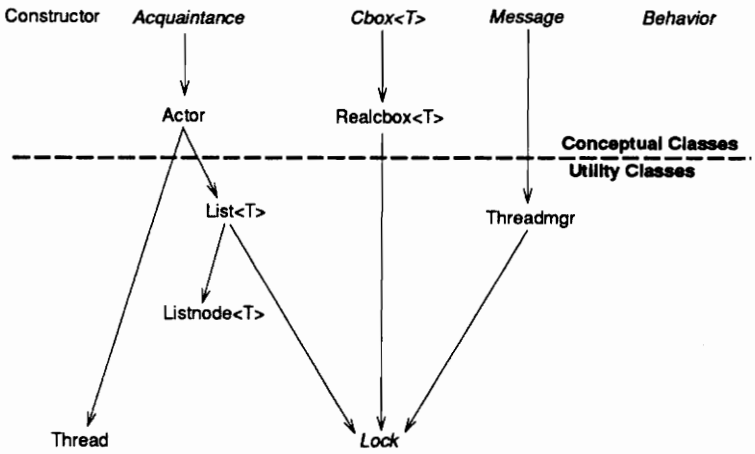


Figure 4.2: Class relationships in ACT++ 3.0.

4.3 Conceptual Class Implementations

This section describes implementations of the conceptual classes. This section does not describe each class in a line-by-line fashion, but instead details the purpose of the class and its methods, goals in building the class, and explains parts of the implementation that are not obvious.

4.3.1 The Constructor Class

The Constructor class in the shared memory version is used merely as a "wrapper" for the initial behavior when creating an actor using the behavior interface template (Section 2.3). The Constructor class is defined as follows.

```
class Constructor{
  char* myName;                  /*constructor's name*/
 public:
  char* name();                  /*return name of this constructor*/
```

```
  Constructor(char* n);          /*give constructor name n*/
  virtual Behavior* New() = 0;
};
```

The constructor method takes a pointer to a string and sets *myName* equal to it (the string is not copied). The name() method returns *myName*. In the distributed version, these methods will be used more fully. The purely virtual method New() is defined in each behavior class interface template's Create class.

Recall that when creating an actor through its acquaintance, the initial behavior is specified through the use of the interface template's Create() method, which returns a pointer to a Constructor. The pointer is passed to the Actor class's constructor method. The Actor class constructor calls the Constructor pointer's New() method, which returns the desired initial behavior.

4.3.2 The Acquaintance Class

The Acquaintance class is used as the interface to the Actor class. The Acquaintance class described here is for the shared memory version. In the distributed version, the Acquaintance class will also handle a remote actor.

The Acquaintance class is defined as follows.

```
class Acquaintance {
  Actor* act;         /*actor this acquaintance points to*/
 public:
  void send(Message* msg);                  /*send message to the actor*/
  Acquaintance(Constructor* ctor,int = 1); /*create actor using ctor  */
};
```

The *act* private variable is a pointer to the actor the acquaintance references. The send() method delivers the message to actor pointed to by *act* by calling

```
act->bind(msg);
```

The Actor::bind() call will create a new thread if a behavior is available in the actor to process the passed in message.

The Acquaintance class constructor takes two arguments. The first, a pointer to a Constructor object, encapsulates the initial behavior for the actor. The second argument is used to specify the maximum number of threads the actor can have concurrently running. It is set by default to be 1. The constructor method constructs a new actor, pointed to by *act*, by calling the Actor class' constructor method. Both arguments are passed to the Actor class constructor.

4.3.3 The Message Class

All messages in ACT++ 3.0 inherit from this class. Messages are constructed through the behavior interface templates (Section 2.3) which return an object of a class derived from the Message class. Each derived type encapsulates the behavior method and arguments for a specific behavior's method. The Message class provides the message object with methods used by ACT++ 3.0 to process the message.

The Message class is definded as follows.

```
class Message {
  Behavior*     myBeh;   /*used so message knows what beh it's bound to*/
  Actor*        myAct;   /*used so message knows what actor it's bound to*/
 public:
  virtual ~Message();
  virtual void invoke(Behavior*) = 0;/*calls the behavior method*/
  void send(Acquaintance* acq); /*send message to actor via acquaintance*/
  static void run(Message *msg);/*thread start routine*/
  void bind(Actor* act);        /*set this actor as mine*/
  void bind(Behavior *beh);     /*set this beh as mine*/
```

```
};
```

Recall from Section 4.1 that a thread starts its execution in the Message::run() method. The argument passed to the run() method is a message object produced by the behavior interface template (Section 2.3). The run() method is what a created thread actually executes. The behavior method is executed as a part of the run() method (Figure 4.1). The run() method is implemented as follows.

```
void Message::run(Message* msg){
  msg->invoke(msg->myBeh);
  msg->myAct->bind();      /*see if msg & beh exist for new thread*/
  if ((msg->myBeh->self_repl()) == 0){
    delete msg->myBeh;     /*don't need this beh - not reused*/
  }
  delete msg;
  threadmgr--;
}
```

The passed message object of a Message class derived type has a defined invoke() method which corresponds to the Message class' purely virtual invoke() method. The behavior object specified in *myBeh* is the behavior object that will process the message and was set in the Actor class when it was determined that both a behavior and a message were available for processing (*myBeh* is the available behavior). The defined invoke method then calls the behavior object's appropriate computation method.

The thread returns to the run() method after the invoke() finishes. It then executes:

```
msg->myAct->bind();
```

myAct refers to the actor in which the message is processed. It was set in the Actor class at the same time as *myBeh*. The call to the actor's bind() method checks to see if there is an

available behavior and message in that actor. If there is, a new thread is created. The next statement deletes a behavior if it is not re-used. This occurs in the case where this thread does a become operation and specifies a behavior other than *myBeh* as the replacement behavior. The message object is then deleted since it has already been processed.

The *threadmgr−−* call decrements the count of running threads in the system since this thread will immediately terminate.

The virtual destructor has no operations. It is placed here so that the destructor specifed for each subclass in the interface template will be called. The reason for the destructor in the interface class is due to the reference counting of Cboxes, explained further in Section 4.3.5.

The two bind functions are used to set *myAct* and *myBeh* and are both called in the Actor class, immediately before a thread is to created.

4.3.4 The Actor Class

The Actor class is the implementation of an actor. This class is mainly responsible for "managing" the processing of messages through its behaviors. The Actor class is not accessible by the user and is accessed through the Acquaintance class. The Actor class is defined as follows.

```
class Actor{
  int             max_threads;  /* max number of threads allowable       */
  int             curr_threads; /* number of threads currently running   */
  Behavior        *nextBeh;     /*The next behavior to assign to a thread*/
  Lock            mutex;        /*used for synch in actor               */
  List<Message>   mailq;        /*where messages for actor are kept     */
  Condition       condition;    /*flag to indicate if actor is dead     */

 public:
```

```
  enum Condition {dead, alive};
  Actor(Constructor*, int = 1);
  void terminate();                    /*make this a dead actor*/
  virtual void bind(Message*);        /*put msg in mailq and attempt to*/
                                       /*and see if can start new thread*/
  virtual void bind(Behavior*);       /*make this the next actor behavior*/
                                       /*and see if can start new thread*/
  virtual void bind();                /*see if msg + beh exist to start*/
                                       /*new thread*/
};
```

The enumerated type Condition refers to the state of the actor. A "dead" actor does not receive or process messages, whereas an "alive" actor receives and processes messages.

The *max_threads* member is the maximum number of concurrently executing threads that the actor can have. The instance variable *curr_threads* is the current number of concurrently running threads in the actor. The behavior object that processes the next message in the mail queue is pointed to by *next_Beh*. When a behavior is not available to process the next message, *next_Beh* is set to NULL_BEHAVIOR which is defined in the Behavior class. To prevent race conditions, *mutex* is used to provide mutual exclusion for critical sections in the class methods. The actor's mail queue is defined by *mailq*, a List class template instantiated with the Message class type. The mail queue is infinite and operates in a FIFO manner. The variable *condition* indicates if the actor is alive or dead.

The Actor constructor method is called in the Acquaintance constructor method and is passed a pointer to a Constructor object and an integer specifying the maximum number of concurrently running threads allowable (the default is 1). The Constructor pointer's New() method is called, returning the actor's initial behavior; *next_Beh* is assigned to that initial behavior. The initial behavior is informed of its actor by the Behavior::set_actor() method. The last steps in the constructor are to set the thread count variables and to set the actor to be "alive."

The terminate() method is called in Behavior::become(void) and terminates the actor (*condition* = dead). A terminated actor cannot be revived.

The three bind() methods are the heart of the Actor class. Each bind() method attempts to combine a message in the mail queue, an available behavior, and a possibly new thread. The first bind() method is called when an actor is sent a message. The second bind() method is called when a replacement behavior is specified in a Behavior::become() method and the third bind() is performed in Message::run() right before the thread terminates.

For a thread to be created, the following conditions must be met.

- The actor must be "alive" to process messages. If *bind(Message* msg)* was called, the passed in message object is deleted.
- Both a behavior and a message must be available. The first two bind()'s are passed either a message or a behavior, respectively, meaning that they only need to check for an available behavior or message, respectively. The third bind() must check for both. In the first two binds, if the needed behavior or message is not available, the passed in argument is saved in the appropriate variable (*next_Beh* or *mailq*) for later use.
- The maximum number of concurrent threads cannot be exceeded. If the actor is at its limit, the first two bind()'s save their passed in arguments in the appropriate variables. In the case of the third bind(), the thread count is not checked, since this bind() is called only when a thread is about to terminate. This call decrements the thread count for the terminating thread and guarantees that at least one thread can now be created.

If all three conditions are met, a thread is created. All three bind()'s increment the thread count and let the message have access to the actor and its behavior by calling the Message::bind() methods (Section 4.3.3). The *next_Beh* variable is set to NULL_BEHAVIOR since that behavior is now going to process a message and is, therefore, no longer available. Finally, the thread creation call is executed. As was stated in Section 2.3, all thread creation calls in ACT++ 3.0 have the same arguments.

4.3.5 The Cbox<T> and realCbox<T> Classes

The Cbox class is a handle for the realCbox class, which is the "real" implementation of the cbox concept. The main reason for the Cbox class is to provide a means for reference counting realCbox objects. The Cbox and realCbox classes are templates and can take any type as their argument. The realCbox class is defined as follows.

```
template<class T> class realCbox{
  T value;
  Lock guard;
  int refcount;
 public:
  realCbox() : guard(Locked);
  ~realCbox();

  operator T();           /*v = realCbox<T>*/
  void operator = (T v); /*realCbox<T> = v*/

  void addref();
  int remref();
};
```

The private variable *value* holds the value that is stored in the cbox, *guard* is used to block threads that try to read an empty cbox, and *refcount* is the count of Cbox objects that reference this realCbox object.

The constructor for a realCbox initializes *guard* to be locked and sets *refcount* to 1 since a realCbox cannot be created without creating an associated Cbox object. The two overloaded operator functions are used to place data in the cbox and to read data from the cbox. These two functions block threads and unblock threads by using the *guard* lock.

```
  operator T(){
    T val;
    guard.lock();
    val = value;
    guard.unlock();
    return(val);
  }

  void operator = (T v) {
    value = v;
    guard.unlock();
  }
```

When a thread tries to read the value in the cbox, the "operator T()" method tries to lock *guard*. Since *guard* is locked at initialization, trying to lock the locked *guard* causes the thread to block. The thread is unblocked when another thread writes to the cbox. The "operator = (T v)" method first sets *value* equal to the input argument and then unlocks *guard*. This unblocks any threads trying to read from the cbox. One thread acquires the lock and is allowed to read the value from the cbox. After the value is read, the thread unlocks *guard* and lets the next thread execute. Note that this now allows any thread to read the cbox since *guard* is always unlocked after a read is completed.

The addref() and remref() methods are used by the Cbox class to increment and decrement the realCbox's reference count.

```
template<class T> class Cbox{
  realCbox<T> *cboxptr;
 public:
  Cbox();
  Cbox(Cbox<T>& c);
```

```
  ~Cbox();
  void operator = (Cbox<T> v); /*myCbox = otherCbox*/
  void operator = (T v);       /*Cbox = value*/
  operator T();                /*value = Cbox*/
};
```

The instance variable *cboxptr* is the pointer to a realCbox object. At construction, a realCbox is also constructed, taking the same template argument. The second constructor does not create a new realCbox, but instead the created Cbox object points to the same realCbox as the input Cbox object. The reference count of the realCbox is incremented. The "operator = (Cbox<T> v)" method works in a similar fashion.

```
  void operator = (Cbox<T> v){   /*so can do myCbox = aCbox*/
    if (cboxptr == v.cboxptr)
      return;
    else{
      if ((cboxptr->remref()) == 0)
        delete cboxptr;
      cboxptr = v.cboxptr;
      cboxptr->addref();
    }
  }
```

The "operator = (Cbox<T> v)" method lets an existing Cbox object, with an existing realCbox, reference another Cbox object's realCbox instead. As can be seen from the code, if both Cbox objects point to the same realCbox, nothing is done. However, if they do not point to the same realCbox, then the discarded realCbox's reference count is decremented. If the reference count becomes zero, that realCbox is destructed. The other realCbox's reference count is incremented. The second constructor and the above overloaded operator provide a way for a realCbox object to be easily passed from actor to actor.

The ease in passing a realCbox object creates a problem with message objects. Whenever a message object is created by an interface template subclass, it keeps a copy of the arguments passed to it. If a Cbox object is passed, then the message also copies it, increasing the realCbox' reference count. Normally, one could assume that when the message is deleted, the Cbox object is also deleted and the realCbox reference count is decremented. However, this does not happen. Deleting a message calls the Message class' destructor, which neither destructs the Cbox object nor decrements the realCbox's reference count. This leads to realCbox's never being deleted. To solve this problem, the Message class' destructor is defined as virtual, and the interface template subclass, which inherits from the Message class, is generated with a defined destructor. Then, when the message is deleted, the subclass' destructor is called instead of the Message class' destructor, causing the Cbox object to destruct and the realCbox's reference count to decrement.

The Cbox class destructor method decrements the count of the referenced realCbox. If the reference count becomes zero, the realCbox is,also destructed.

The last two methods provide the interface to read and write to the realCbox. Both of these methods call the corresponding method in the realCbox class.

4.3.6 The Behavior Class

The Behavior class is the base class for all user-defined behaviors, and is defined as follows.

```
#define NULL_BEHAVIOR (Behavior*)0          /* null behavior */


class Behavior{
  BehReUse  beh_reused;       /*flag saying if behavior reused*/
 protected:
  Actor     *my_actor;         /*actor to which beh is bound*/
 public:
```

```
  enum BehReUse {Reused,Not_Reused};
  Behavior();
  void become(Behavior* b);     /*next behavior will be b*/
  void become(void);            /*terminates the actor*/
  int self_repl();              /*tells if behavior is being reused*/
  void set_actor(Actor *a);     /*assign actor to behavior*/
  void send(Message *msg);      /*send a message to my actor*/
};
```

The enumerated type BehReUse is only used in this class. The NULL_BEHAVIOR constant is used in the Actor class to reset a local behavior object member.

The private member *beh_reused* is used to mark the behavior object as Reused or Not_Reused. If a behavior specifies itself as the replacement behavior (*become(this)*), then it is marked Reused, otherwise it is marked Not_Reused. A behavior is marked as such so the Message::run() method can determine if the behavior should be deleted. A behavior is deleted in Message::run() *after* the execution of the behavior method (Figure 4.1). Deleting the behavior object earlier may cause an executing behavior method to abnormally terminate, since it may try to access information specific to the just-deleted behavior object. The Message::run() method accesses whether or not a behavior is re-used through the self_repl() method.

The protected member *my_actor* points to the actor object with which the behavior is associated. It is necessary for a behavior to know its actor so that when it does a become operation, it can acess the actor to see if there is a message available for the replacement behavior to process.

The constructor for the Behavior class sets *my_actor* to (Actor *)0 and sets *beh_reused* to Not_Reused. The variable *my_actor* is set to an actor when the behavior is 1) the actor's initial behavior, or 2) the behavior is a replacement behavior.

The first become() method marks the current behavior Reused or Not_Reused as explained earlier and informs the replacement behavior, if it is a new behavior, of its actor.

The method then checks to see if the actor has a message to process.

The second become() shuts down the *my_actor* object by calling its terminate() method.

The set_actor() method is used to set *my_actor* to point to an actor object. It is used in the first become operation to inform a new behavior of its actor and is used in the Actor class constructor to tell the initial behavior its actor.

The send() method is used by a behavior to deliver messages to its actor. In this way, a behavior can send itself a message. Since a behavior is always local to its actor, the send message can be implemented as follows.

```
my_actor->bind(msg);
```

By making *my_actor* protected, the user-defined behaviors can inherit access to it and use the send() method and not have to send messages through an acquaintance.

4.4 The Utility Class Implementations

This section discusses the implementation of the utility classes. Utility classes can be defined as classes that do not relate directly to the actor model. As in the previous section, the discussion covers each class and its methods, goals in building the class, and explains parts of the implementation that are not obvious.

4.4.1 The List<T> and Listnode<T> Classes

The List class is a template class that provides a FIFO queue of elements whose type is given by its template argument. The List class is implemented primarily for use by the Actor class' mail queue. However, it is implemented as a template to enable code re-use, in case another queue, such as a behavior set queue, is needed in the future.

The List class is implemented as a linked list. Each node in the list is of type Listnode<T>, defined as follows [STR91].

```
template<class T>
struct Listnode{
  T  *node;
  Listnode *next;
  inline Listnode(T *a);
  ~Listnode();
};
```

The Listnode class takes the same template argument as the List class. The Listnode constructor produces a single node, where the data in the node is the passed in argument. The destructor does not delete the data element in the node.

The List class is defined as follows.

```
template<class T> class List{
  Listnode<T>*   list_head;
  Listnode<T>*   list_tail;
  int            list_length;
  Lock           list_lock;  /*so these methods are safe*/

 public:
  inline List(T* ptr = 0);
  inline ~List();

  inline void append(T* ptr);
  inline void prepend(T* ptr);
  inline T* get();
  inline int length();
};
#endif
```

The constructor initializes the list. It takes a null pointer by default, in which case a list with zero nodes is constructed. However, the user can specify an initial data element, in which case the list is constructed with one node.

The destructor deletes all nodes in the list. However, any values T in those nodes are not destroyed.

The append() method constructs a node using its argument value and adds that node to the end of the list. Prepend() works the same way, but puts the node in the front of the list. The get() method returns the data element T from the node at the head of the list and length() returns the number of nodes in the list.

Note that the List class and Listnode class are designed to work only with pointers to the template argument T. This implies that these two classes do not copy the data objects passed to them.

4.4.2 The Threadmgr Class

The Threadmgr class is used to keep a count of the number of running threads in the system. A running thread is defined as a thread that has been created, but has not yet finished execution, and so includes any threads that are blocked. Because of its purpose, only one object of this class is declared in a program. The object is declared to be global to the program and is accessed only in the main() procedure, the Threadwait() utility, in the Message::run() method, and in the Thread::fork() method. This class is only used in the shared memory version and is defined as follows.

```
class Threadmgr{
  int num_threads;     /*current number of threads*/
  Lock mgr_lock;       /*for mutual exclusion*/
  Cbox<int> *mgr_cbox;/*for threadwait*/
 public:
  Threadmgr();
```

```
  cleanup();
  void operator ++ ();
  void operator -- ();

  void threadwait();
};
```

The instance variable *num_threads* keeps a current count of threads in the system. It is initialized to zero in the constructor. The thread count is incremented through the "operator ++()" method and decremented in the "operator --()" method. The thread count is incremented only in the Thread class, immediately before a thread is created, and is decremented only in Message::run, immediately before a thread terminates. The cbox *mgr_cbox* is used to halt the usermain() procedure when the Threadwait() utility is called. It is originally set as a null pointer. The lock *mgr_lock* is used to provide mutual exclusion in critical sections of the class methods.

The cleanup() method, called by the main() procedure immediately before program termination, prints out a message telling if either the program finished normally or if it finished before one or more threads were able to complete their execution.

The threadwait() method is called by the Threadwait() utility. Recall that the Threadwait() utility halts the usermain() procedure until all threads in the system have completed execution. When the threadwait() method is called, a cbox local to the method is constructed. If, at that time, there are threads still running, *mgr_cbox* is set to point to the constructed cbox. The method then tries to read from the cbox and blocks since the cbox is empty. If zero threads are running, the threadwait does nothing and returns since there are no threads to wait for.

In the "operator --()" method, where *num_threads* is decremented, if the thread count reaches zero and if *mgr_cbox* does point to a cbox, then a dummy value is placed in *mgr_cbox*'s cbox. This cbox is the same cbox that is local to threadwait(). Since the common cbox

has been written to, the threadwait() method unblocks. The threadwait() method then resets *mgr_cbox* to be a null pointer so it can be reused and returns, allowing the usermain() procedure to continue.

By using a cbox that is local to threadwait(), the Threadwait() utility can be used multiple times, since the cbox is destructed each time threadwait() returns. By having a class defined cbox pointer, the cbox local to threadwait() can be accessed by the public method "operator $--$()", which can then place a value in it when the thread count reaches zero.

4.4.3 The Thread Class

The Thread class provides an interface to the Pthreads library. Only a small subset of the Pthreads functions are reflected in the Thread class interface. For example, ACT+ 3.0 does not deal with thread scheduling and thread prioritization, but instead uses the default round-robin scheduler for Pthreads, where all threads have the same priority.

Currently, the Thread Class is implemented on the Ultrix implementation of Pthreads [ULT93] and on Solaris threads [SOL93], which do not strictly conform to the Pthreads standard. They implement the needed subset of the Pthreads library for which Solaris threads do conform to the Pthreads standard.

The Thread class is defined as:

```
class Thread {
 private:
  pthread_t        tid;    /*thread identifier*/
 public:
  Thread();
  ~Thread();

  Thread fork(any_t obj,any_t arg1);
```

```
};
```

The fork() method is used to create and start threads and is called only in the Actor class. The first parameter to fork() is the procedure to be executed and the second parameter is the one argument that can be passed to the procedure. These two parameters are passed as type *any_t*, which is a typedef for a pointer to a void, so the program can compile cleanly. In the fork() method, the global thread count is incremented when the thread is created. If a thread cannot be created, fork() causes the program to exit.

The fork() method creates and starts a thread by calling the Pthreads thread creation routine. The semantics of that routine do not distinguish between creating a thread and starting a thread. So, after a successful call to the fork() method, a thread is known to have been created and put in the scheduler's ready queue, but it cannot be known when that thread will run.

Threads are created as *detached*, meaning that as soon as a thread terminates, its memory can be immediately re-used by the system. This is done to use memory efficiently.

Threads are created with the default stacksize. This is not efficient, since the stacksize is quite large. However, Solaris threads do not work in daemonized processes unless they have the default stacksize. To maintain consistency, threads using the Ultrix implementation of Pthreads are also created with default stack size. We believe the problem with Solaris threads are due to either some system requirements internal to the machine architecture or to the Pthreads implementation.

The constructor is only used by Solaris to set flags on variables passed to the Solaris thread creation routine in order to specify that a thread is detached and has default stacksize.

The destructor is not used in either implementation.

Thread class objects are only used in the Actor class. After the thread is created, the thread object is discarded. This implies that ACT++ 3.0 does not retain any "handles" to a created thread.

4.4.4 The Lock Class

The Lock class is used to provide a binary semaphore for mutual exclusion. Besides the Thread class, this is the only class which actually uses the Pthreads library. As with the Thread class, the Lock class is implemented on the Ultrix implementation of Pthreads and on Solaris threads. Solaris threads implement the needed subset of the Pthreads library, and for this subset works the same as the Pthreads standard.

Pthreads offer mutexes for mutual exclusion. However, mutexes in Pthreads have "owners," which means that the only thread which can unlock a mutex is the thread which locked it. Recalling the implementation of the realCbox class, this is not desirable. In the realCbox class, a thread is blocked when it tries to lock an already locked guard. That thread is unblocked when some other thread unlocks the guard. To use Cboxes as intended, it is necessary to provide for the case where the blocked thread is the creator of the cbox and by initialization, locks its guard, and some other thread unlocks that guard. Using Pthread mutexes would prohibit this, since only the thread which locked the guard would be able to unlock it. To allow for an "ownerless" mutex, the Lock class was created.

The Lock class is defined as follows.

```
class Lock{
 private:
  pthread_mutex_t spin_mtx;  /*to synch access to count*/
  pthread_cond_t  spin_cond; /*the cond var on which we will block*/
  LockState       state;     /*state of the lock*/

 public:
  enum LockState  {Locked,Unlocked};

  Lock(LockState init_state = Unlocked) {
```

```
    pthread_mutex_init(&spin_mtx,pthread_mutexattr_default);
    pthread_cond_init(&spin_cond,pthread_condattr_default);
    state = init_state;
  }
  ~Lock(){
    pthread_mutex_destroy(&spin_mtx);
    pthread_cond_destroy(&spin_cond);
   }

  void lock(){
    pthread_mutex_lock(&spin_mtx);
    while (state == Locked){
      pthread_cond_wait(&spin_cond,&spin_mtx);
    }
    state = Locked;
    pthread_mutex_unlock(&spin_mtx);
  }

  void unlock(){
    pthread_mutex_lock(&spin_mtx);
    state = Unlocked;
    pthread_mutex_unlock(&spin_mtx);
    pthread_cond_broadcast(&spin_cond);
  }
};
```

A Lock object is considered locked or unlocked depending on the value of *state*. The Pthreads mutex *spin_mtx* is used to provide mutual exclusion to *state*. The Pthreads con-

dition variable *spin_cond* is used to block threads when *state* is Locked. By basing the Lock class on a simple variable, any thread can lock or unlock a Lock object.

When a thread locks the Lock object, *state* is set equal to Locked. Any subsequent threads attempting a lock() go into the while loop and block on *spin_cond*. When the Lock object is unlocked, *state* is set to Unlocked. The semantics of the condition broadcast on *spin_cond*, dictate that all blocked threads are unblocked, but only one is allowed to acquire the mutex lock *spin_mtx* and continue; the rest block on *spin_mtx*. The unblocked thread checks the while loop's condition, sees that it is false, changes *state* to Locked, and unlocks *spin_mtx*. All threads blocked on *spin_mtx* now wake up and check the while loop's condition again. Since *state* is Locked, these threads block again on *spin_cond*. Through *spin_mtx*, threads attempting to lock the lock object are prevented from interfering in the above process by blocking them on entry to lock(). When *state* is safely changed, unlocking *spin_mtx* lets these threads continue to the while loop and block on *spin_cond*.

4.4.5 Main() and Threadwait()

As mentioned in the previous section, the user must call their "main()" usermain() since ACT++ 3.0 already has a main() procedure defined. This allows the Threadmgr object to work hidden in the background. Procedure main() looks like the following.

```
#include "cbox.h"
#include "Threadmgr.h"
extern void usermain(int argc,char* argv[]);
Threadmgr threadmgr;

main(int argc,char* argv[])
{
  usermain(argc,argv);
  threadmgr.cleanup();
```

```
}
```

In this way, the user does not have to remember to declare a threadmgr object or call threadmgr.cleanup().

The same is true of Threadwait(). Threadwait() is used to block the usermain() from executing until all threads have been allowed to finish. Threadwait() is implemented as follows.

```
#include "Threadmgr.h"
extern Threadmgr threadmgr;

void Threadwait(){
  threadmgr.threadwait();
}
```

As can be seen, it is only a call to threadmgr.threadwait(). However, by using Threadwait(), ACT++ 3.0 hides the Threadmgr class. The user neither provides the "extern" declaration for the Threadmgr class nor directly calls its threadwait() routine.

Chapter 5

Stress Testing and Performance Results

5.1 Stress Testing

ACT++ 3.0 has been stress-tested to verify its implementation. The main goal is to test the memory management of the system. The functionality of the system as a whole is also implicitly tested since the system must first operate correctly for the stress tests themselves to succeed. However, special attention is paid to the functionality of the Actor, Behavior, Message, Cbox, and realCbox classes during these tests.

These tests check for memory leaks by constantly creating and deleting objects within an infinite loop. The infinite loop is achieved by the behavior method always sending the same message to its actor, namely to execute that behavior method again. The tests print out a message after every 1,000 or 10,000 threads have been executed and the user checks for memory leaks by executing a "ps -ux" system call every couple of messages. The "RSS" and "SZ" columns shows how much real memory and virtual memory, respectively, a program is using. If the test program does not have any memory leaks, then these values will not change from call to call. If the test does have leaks, then the values will increase. By checking every few thousand messages, the user can see if the program is wasting as little as one-byte of memory per thread since the "RSS" column shows memory usage in 1024-byte blocks.

As a side note, it seems that the system needs some time to "stabilize." When doing the initial "ps -ux" commands early in the program's execution, the "SZ" and "RSS" increase slightly. However, after the test prints out a few messages, they do stabilize at a constant number. Since the program continues, but the increase does not continue, we believe that

the increase is not due to ACT++ 3.0, but perhaps due to the operating system or the "ps" system call itself.

These tests implicitly verify parts of the system because of they way the execute. For example, these tests show that the Behavior::become() operation works correctly and that all the Actor::bind() methods also work correctly. If these did not work correctly, then the test programs would fail in their execution.

5.1.1 Threads and Messages Test

This goal of this test is to see if there are any memory leaks pertaining to the creation and destruction of threads and messages. The test uses a singly-threaded actor whose behavior has a method newthread(). The usermain() creates the actor with the initial behavior and sends it a message requesting the behavior method newthread(). The newthread() increases a counter, sends a message to the actor requesting the newthread() method again, and finally specifies itself as the replacement behavior. The counter keeps a tally of the number of times the behavior method has been executed and, therefore, how many threads have run. A message is printed out to the user every 10,000 threads. The test has no specified stopping point since the behavior always sends a message to the actor.

This test checks for memory leaks in threads since a large number of threads are created and terminated during its execution. It also tests for memory leaks in message creation and deletion since one message is created and deleted per thread.

5.1.2 Behaviors Test

This test checks for memory leaks in behavior creation and deletion. This test is similar to the one above, but a new replacement behavior is created for each new message. In this way, the behavior that sends the message is deleted and a new one is created to replace it. The test prints out a message for every 1,000 behaviors that are created and deleted.

Both this test and the previous test implicitly verify Behavior::become(), Actor::bind(Message* msg), and Actor::bind(Behavior* beh).

5.1.3 Multiple Actors and Threads Test

This test verifies the ability of ACT++ 3.0 to handle multiple actors, each with multiple threads. There are twenty actors in the system, each with a maximum of ten concurrent threads. Twenty messages are sent initially to each actor, so that the ten threads have a chance to run concurrently. The executed behavior method is the same as for the previous two tests, but the become() operation is done first so that the maximum number of threads are allowed to run concurrently. A message is printed out by each actor after every 1,000 threads it runs. The test is considered successful if it is able to run continuously and all twenty actors are able to print out their messages periodically.

This test can also be used to test for memory leaks, since the system scale is much larger than for the previous tests. However, this would only find the same memory leaks that would be found by the first two tests. This test also verifies that Actor::bind() functions correctly. When the first thread in an actor executes, the behavior method it is executing does a become() operation immediately, eventually causing nine more threads to be spawned. Subsequent become() calls do not start new threads because the actor thread limit is reached. When the first thread terminates, room becomes available for another thread and since the eleventh message is still in the mail queue, the call to Actor::bind() will be successful.

5.1.4 Cbox Memory and Functionality Test

Two tests are performed on cboxes. The first test is used to verify that reference counting functions correctly. The second test verifies that cboxes can be passed through messages and that threads block on empty cboxes and wake up when the cbox is filled with a value.

The first cbox test is exactly the same as the Threads and Messages test, except that newthread() is now passed a cbox. The newthread() method does nothing with the passed cbox, but it does create another cbox for the next execution of newthread(). In this situation, there will be up to three objects that reference the cbox, the message object, through

which the cbox is passed, the newthread() method that creates the cbox, and the subsequent newthread() method that receives the cbox. The message's reference to the cbox is eliminated when the message is deleted. Both executions of the newthread() method lose their reference to the cbox when they finish executing since the encapsulating cbox object is local to both of them. Therefore, since there are no references left, the cbox should be destroyed.

A message is printed out every 10,000 threads. The success of this test can be measured by the "ps -ux" command.

The second cbox test is simpler. Five actors are created and sent a message to wait on a cbox. All five actors wait on the same cbox. A sixth actor is created and sent a message to write information into the same cbox. Each of the first five actors prints out a message before they read the cbox and after they have read the cbox. The sixth actor prints out a message before and after the cbox is written.

Since the order in which the actors run is non-deterministic, the test is considered successful if the reading actors, which execute before the writing actor, only print out their message for an attempt to read the cbox, indicating that they block on the empty cbox. The writing actor should be able to print out both of its messages, after which all the blocked actors should be able to read the cbox and print out the appropriate message, indicating they were awakened.

This tests not only the blocking and unblocking semantics of a cbox, but also proves that the same cbox was referenced by six different actors, implying that cboxes can be sent through messages.

5.1.5 Actors Test

There is no memory test for actors. Actors are not reference counted by their acquaintances like cboxes because actors are active; they have threads of execution running in them. So even if an actor has no acquaintances, it may still be computing. Deleting the actor in this case is inadvisable. Determining when an actor should be deleted is one of the

problems not currently addressed by ACT++ 3.0, but is instead left for future work.

5.2 Performance Tests

Threads offer the advantage over processes of being lightweight and being able to run concurrently. However, with concurrency comes overhead, which may nullify the advantages of using threads. Three performance tests were done using ACT++ 3.0 to determine its overhead. The first test measured thread creation time and the second test measured context-switching time. The third test compared processes and threads in solving an application. Specifically, a timed comparison of the quicksort algorithm wass done using recursion at the process level versus using threads and ACT++ 3.0.

All tests were run on csgrad.cs.vt.edu (DECStation 5000/240 running Ultrix 4.3.1) and on actor.cs.vt.edu (SPARCStation 10 running Solaris 2.3.) The tests were run when the machines were at their "usual" load. For csgrad.cs.vt.edu, this implies a moderately-loaded machine, and for actor.cs.vt.edu, a lightly-loaded machine. Because the testing conditions were not rigorous, these tests are not meant to be definitive, but instead suggestive of the performance characteristics of threads and ACT++ 3.0.

5.2.1 Thread Creation Time

Since every thread must go through a creation process, thread creation time can play a large role in system overhead. Thread creation time in ACT++ 3.0 was measured using the program in Appendix A. An actor is created and given an initial behavior with three methods. The first method, perf_start(), sends 1,000 messages to the actor. The first 999 messages request the second method, perf_become(), and the 1,000th message requests the last method, perf_end(). Method perf_start() starts a timer and executes *become(this)*. This causes perf_become(), whose only operation is to do a *become(this)*, to be executed 999 times. The 1,000th message is processed by perf_end(), which stops the timer, divides the value by 1,000, and returns it to the usermain() procedure. This value is the time to

create one thread and is considered to be fairly accurate since the perf_become() method performs no internal computation, but instead immediately starts a new thread by doing the become() operation.

As a comparison, the raw thread creation time was also calculated using only Pthreads and none of the ACT++ 3.0 code, to show what overhead is due to ACT++ 3.0. The program for raw thread creation time is also in Appendix A. It works in the same manner as the ACT++ 3.0 thread creation time, by having a single thread spawning off a new thread before terminating. The new thread then creates another thread and terminates. This process continues until 1,000 threads have been created, after which the time is taken and the per thread creation time calculated.

This process is repeated ten times each time each program is run and the collected values are averaged to give a final value. Each program was run 10 times. The low, average, and high values are shown below in Table 5.1 and Table 5.2.

Table 5.1: Thread Creation Times on actor (microseconds)

Machine	Low	Average	High
Pthreads only	111.253	134.325	161.472
ACT++ 3.0	187.754	197.222	215.467

Table 5.2: Thread Creation Times on csgrad (microseconds)

Machine	Low	Average	High
Pthreads only	75.782	86.923	106.639
ACT++ 3.0	208.599	215.193	256.634

As can be seen from both tables, ACT++ 3.0 does add some overhead to the thread creation time. This is to be expected, since ACT++ 3.0 has extra processing in order to implement the actor model. The variation in the times per trial can be attributed

to machine load since the performance test is always the same. However, the times are usually centered around the average, with only a few results near the low and high times. The difference in times between the two machines may be due to their implementation of Pthreads. While the Pthread standard specifies interfaces and functionality, it does not specify implementation details. The two machines may implement threads differently, and thus have different performance characteristics.

5.2.2 Context-Switching Time

Another major factor in system overhead is context-switching. The program in Appendix B was used to measure the per thread context switching time in ACT++ 3.0. Two singly-threaded actors are created, each with access to two locks. The actors' behaviors access the locks in the following manner.

```
int context_switch = 0;
Lock lock1,lock2;
lock1 is initially unlocked
lock2 is initially locked

void switch_become(){  /*ACTOR 1's behavior*/
  while(1){
    lock1.lock();
    lock2.unlock();
    context_switch++;
    if (context_switch == 2001)
      break;
    }
  become (this);
}
```

```
void switch_follow() { /*ACTOR 2's behavior*/
  while (1){
    lock2.lock();
    lock1.unlock();
    context_switch++;
    if (context_switch == 2000)
      break;
  }
  become(this);
}
```

By tracing from the initial lock conditions, it can be seen that the execution of the behaviors is chained together. A timer is started by Actor 1 before the two behaviors enter the chained loop. The behaviors perform 2,000 context switches, after which Actor 1's behavior breaks from the loop and processes its next message. The resulting behavior method stops the timer and reports the time to usermain().

This provides a good estimate of the context switching time since the two behaviors have little processing involved. By doing the test this way we can also keep track of the number of times a context switch occurs. A better test however, would eliminate the need to keep a counter, since it adds extra, unwanted processing to the behavior.

The above process is repeated ten times each time the program is run and the collected values are averaged to give a final value. The program was run 10 times. The low, average, and high values are shown in Table 5.3. A comparison was not done against a Pthreads implementation, since the actor model has no concept of context-switching, and there no part of ACT++ 3.0 that deals with context-switching. The times shown in the table therefore also reflect the time to context switch using Pthreads since ACT++ 3.0 has no extra processing for context switches.

Table 5.3: Context Switching Times (microseconds)

Machine	Low	Average	High
actor (SPARCStation 10)	43.0375	43.8580	47.1244
csgrad (Ultrix 4.3.1)	21.4718	30.2196	50.9507

Interestingly, on actor the times were all very close to each other, whereas on csgrad they had a larger distribution. However, in general, the times on both machines were centered around the average. The high time for csgrad seemed to be an anomaly since it was significantly greater than the other times. This may be due to some characteristic of the machine load on that particular trial.

5.2.3 The Quicksort Algorithm

The quicksort algorithm was chosen as a way to test an ACT++ 3.0 implementation against a regular process since the quicksort algorithm can be easily mapped to threads. The program for the algorithm is in Appendix C. Instead of a procedure, the ACT++ 3.0 version has a quicksort behavior method. Where the procedure does recursion, the behavior method creates and sends a message to its actor, requesting the quicksort behavior method for the partitioned data. This presents the problem of knowing when the algorithm has finished since the quicksort does not work its way back up the recursion tree like a mergesort. Two implementations were used to solve this problem.

The first implementation uses cboxes, where each behavior method creates and puts a cbox in each message. It sends the messages to the actor and then tries to read the cbox, causing it to block. Each level of the recursion tree does this until there is no recursion left. A behavior method, upon finishing its execution, puts a value into the cbox that was passed into it. Therefore, the behavior methods that are the leaves of the recursion tree will wake their parents who sent the cbox, who will in turn wake their parents, and so on, until the original behavior method wakes up. At this point, the sort is known to be done.

The second solution is to use the Threadwait() utility. Threads are allowed to terminate after sending their quicksort messages since those threads have nothing left to do. Therefore, the last threads to execute are the recursion tree leaf threads. When all the leaf threads finish, the quicksort is done and the Threadwait() utility unblocks.

Table 5.4: Quicksort on actor (microseconds)

Number of elements	Process	ACT++ 3.0 with Cboxes	ACT++ 3.0 with Threadwait()
500	5,844	1,197,624	262,046
600	7,059	1,411,697	313,340
700	8,585	1,490,659	357,875
800	9,519	1,814,093	403,744
900	10,640	1,037,191	437,955
1000	12,276	2,286,234	496,626

Table 5.5: Quicksort on csgrad (microseconds)

Number of elements	Process	ACT++ 3.0 with Cboxes	ACT++ 3.0 with Threadwait()
500	11,718	910,154	285,138
600	15,624	839,844	332,010
700	15,624	1,027,334	406,224
800	23,432	1,171,854	496,126
900	23,436	1,226,540	558,548
1000	23,500	1,585,882	683,614

All three implementations were given the same list to sort, with results shown in Table 5.4 and Table 5.5. The "Number of elements" column refers to how many elements were sorted, and the "Process" column refers to the time to do a recursive, process-level quicksort. The results in Table 5.4 and Table 5.5 reflect only one trial. However, these numbers are only meant to suggest how the three quicksort methods perform in comparison

to one another, and not meant to show definitive results.

On actor, the cbox-based implementation was 191 times slower than the process sort. The Threadwait()-based quicksort was 42.5 times slower. On csgrad, The cbox-based implementation was 61 times slower than the process quicksort and the Threadwait()-based implementation was 24 times slower. Running this trial multiple times shows the same relative difference in times among the three quicksort implementations.

Table 5.6: Quicksort Thread Count on actor

Number of elements	ACT++ 3.0 with Cboxes	ACT++ 3.0 with Threadwait()
500	1000/208	1000/2
600	1200/285	1200/2
700	1400/238	1400/2
800	1600/377	1600/2
900	1798/392	1798/2
1000	2000/508	2000/2

Table 5.7: Quicksort Thread Count on csgrad

Number of elements	ACT++ 3.0 with Cboxes	ACT++ 3.0 with Threadwait()
500	999/190	999/2
600	1200/278	1200/2
700	1399/247	1399/2
800	1600/372	1600/2
900	1800/369	1800/2
1000	2000/405	2000/2

Table 5.6 and Table 5.7 show the number of threads created to do the quicksort and the maximum number of threads that were concurrently running for each implementation. This includes any threads that blocked. The format for the data is "number of threads created/maximum number of threads executing concurrently." From the data, it can be

seen that threads do not improve the quicksort time, but instead make it worse. This can be due to the following reasons.

- Some applications do not translate well to concurrent computation. As can be seen from the Threadwait() implementation, only two threads ever ran concurrently in the quicksort. In the Cbox implementation, a majority of the threads were blocked on cboxes. The nature of the quicksort algorithm does not lend itself to concurrent computation.

- Thread creation is expensive. The quicksort caused many threads to be created, and as seen in Table 5.1, thread creation time is quite expensive. Furthermore, as the algorithm gets further down into the recursion tree, threads do less work, making their thread creation time significant in comparison to their execution time.

- Context switching is also costly. The difference in times between the Cbox implementation and the Threadwait() implementation can be attributed to the fact that the Cbox implementation keeps hundreds of threads in the system. Each of these threads are initially blocked on a cbox. When that thread wakes up and runs, it checks the cbox corresponding to the second recursive call it performed. If that child thread has not yet finished, the current thread will block again. Furthermore, once the thread wakes up from this block, the thread only puts a value in the cbox that wakes up its parent. The thread then finishes executing and terminates. In this situation, each context switch results in little work done. Multiplied over hundreds of threads, this non-productive context switching will become expensive.

Despite these problems, threads and ACT++ 3.0 can be useful. The key is to apply threads and ACT++ 3.0 to the correct problem domain. Characteristics of this domain might be support of massive concurrency and operations with significant computation time when compared with system overhead.

Chapter 6

Conclusion

The goal of this project was to implement the actor model, using POSIX threads, and C++. This project serves as a major part of the shared memory version of ACT++ 3.0, and as a core part of the distributed version. This version offers a few major differences from the previous version.

- It is more portable due to the use of Pthreads. There are many platforms which support Pthreads, allowing ACT++ 3.0 to be used on many different machines. Also, the implementation of Pthreads on these platforms is completely hidden to ACT++ 3.0, providing only a standardized interface to a defined set of commands. Changes in the implementation of Pthreads will not affect the implementation of ACT++ 3.0.
- There is no need for an asynchronous I/O manager, since the Pthreads standard require Pthreads implementations to offer per-thread asynchronous I/O blocking.
- The new design of ACT++ 3.0 allows for static type checking of message arguments against the intended behavior methods.

The shared memory version of ACT++ 3.0 has been completed, and is now being beta-tested by outside users. The distributed version is being built upon the core classes described in this report.

In the course of completing this project, some observations were made that affect the implementation and use of ACT++ 3.0. First, the implementation of ACT++ 3.0, while based on a standardized threads package, does not offer one-hundred percent portability. Implementations of Pthreads on different platforms work in subtely different ways, which is

only noticeable when using Pthreads in somewhat unusual situations. Furthermore, not all platforms are completely "thread safe," since certain system calls may not be implemented for multithreading. ACT++ 3.0, to be portable, is then required to implement machine-specific code, to overcome these obstacles in the Pthreads implementations. While this code is only a minute part of the implementation, it does lessen the degree of separation between ACT++ 3.0 and Pthreads. These Pthreads implementation issues may be due to the "newness" of threads and the "oldness" of the systems on which they are implemented. In the future, as threads are incorporated into the operating systems, most of these problems should disappear.

Also, while ACT++ 3.0 does offer a framework for concurrent computing, it is not meant to be used for all applications. First, not all applications are suited for concurrent computation, as was shown with the quicksort program in Section 5.2.3. Second, threads in ACT++ 3.0 carry some overhead with them due to creation time and context switching. Applications that use ACT++ 3.0 should use threads efficiently, to the point where thread creation time and context-switching time are a small part of the overall execution time.

ACT++ 3.0 would also perform better in a multiprocessor environment where multiple threads could actually run concurrently, instead of being time-sliced within a process on a single-processor machine. This would also reduce the number of context-switches due to time-slicing, since there are now more processors to work on.

Lastly, ACT++ 3.0 does allow the writing of concurrent programs to be relatively painless. Once one program is written, it seems that all other programs follow the same general model. Many of the test programs written for ACT++ 3.0 only involved changing names of classes and changing only one or two behaviors slightly. "Actor-ising" C code was also easily done, as procedures were converted into classes.

Due to the long time used for thread creation in ACT++ 3.0, a topic for future work would be the implementation of a thread pool. A thread pool would consist of already created threads which are not executing a behavior method. When an actor binds a message and behavior together, instead of creating a new thread, the actor could check the thread

pool and "give" the message and behavior to one of the threads there. That thread would then start running again, using the behavior to process the message. This would allow the re-use of threads and eliminate much of the time spent creating threads.

Another topic for future work would be the garbage collection of actors, both in a shared memory and distributed environment. Currently, there is no way to know when to delete an actor. They cannot be reference counted like cboxes because they are active objects. Even if there are no acquaintances associated with an actor, it might still be performing computations. While difficult, garbage collecting actors would be beneficial, especially in large systems where actors are constantly being created.

The implementation of behavior sets for ACT++ 3.0 is also a topic for future work. Implemented in ACT++ 2.0, behavior sets allows for selective message processing in an actor.

A final topic for future work would be to implement a method for the main() procedure to wait for all threads to execute, before it exits the program. The current solution, to have a usermain() and a Threadwait() utility, does work, but it does not sufficiently hide the implementation of ACT++ 3.0 from the user, violating "good" software engineering principles.

REFERENCES

[AGH86] Gul A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, MA, 1986.

[BER88] Bershad, B.N., Lazowska, E.D., and Levy, H.M., "PRESTO: A System for Object-Oriented Parallel Programming," *Software Practice and Experience*, vol. 18, iss. 8, pages 713 - 732, August 1988.

[BER90] Brian N. Bershad, "The PRESTO User's Manual," Report, Department of Computer Science, University of Washington, Seattle, Washington, 1990.

[HBS73] Hewitt, C., Bishop, P., and Steiger, R., "A Univeral Modular Actor Formalism for Artificial Intelligence," *Proceedings of the 3rd IJCAI*, pages 235-245, 1973.

[IEE92] Technical Committee on Operating Systems of the IEEE Computer Society, "Threads Extension for Portable Operating Systems," Draft P1003.4a/D6, February 26, 1992.

[KAF93] Kafura, D., Mukherji, M., Lavender, G., "ACT++ 2.0: A Class Library for Concurrent Programming in C++ using Actors," *Journal of Object-Oriented Programming*, vol. 6, iss. 6, pages 47 - 55, 62, October 1993.

[KAL90] Dennis Kafura and Keung Hae Lee, "ACT++: Building a Concurrent C++ with Actors," *Journal of Object-Oriented Programming*, vol. 3, iss. 1, pages 25-37, May/June 1990.

[MUK92] Manibrata Mukherji, "The Implementation of ACT++ On A Shared Memory Multiprocessor," M.S. Project Report, Department of Computer Science, Virginia Polytechnic Institute and State University, February 1992.

[IDL88] Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Document Number 91.12.1, Revision 1.1, Draft 10, December 1991.

[STR91] Bjarne Stroustrup, *The C++ Programming Language*, second edition, Addison-Wesley, Menlo Park, CA, 1991.

[ULT93] Digital Equipment Corporation, POSIX 1003.4a interface for Ultrix 4.3.1, Maynard, MA, 1993.

[SOL93] Sun Microsystems, Inc., Solaris threads for Solaris 2.3, Mountain View, CA, 1993.

Appendix A

Thread Creation Time Program

This Appendix contains the ACT++ 3.0 and pthread-thread creation time programs. The ACT++ 3.0 program is first, followed by the pthreads-only implementation.

usermain.C:

```
extern "C" {
#include <stdio.h>
}
#include "act++.h"
#include "perftestops.h"
#include "perftest.h"

usermain(int argc, char* argv[])
{
  int t;
  double average = 0.0;
  double temp;

  Perftest<Perftestops>  aPerftest;
  Acquaintance* PerftestActor = new Acquaintance ( aPerftest.Create());
  Message* msg[10];

  for (t = 0; t< 10; t++){
    Cbox<double> main_cbox;
```

```
    msg[t] = aPerftest.perf_start(main_cbox);
    PerftestActor->send(msg[t]);
    average = main_cbox + average;
  }

  average = average/10.0;
  printf("\nTOTAL AVERAGE for thread creation = %f microseconds\n",average);
}
```

This is "perftestops.h" and contains the behavior methods:

```
#ifndef  _perftestops_h
#define  _perftestops_h

extern "C" {
#include <sys/time.h>
}
#include "act++.h"
#include "perftest.h"

struct timeval tp,tp2;
struct timezone tzp,tzp2;

class Perftestops : public  Behavior {
public:
  Perftestops() {};

  void perf_become(){
    become (this);
```

```
}

void perf_end(Cbox<double>& cbox){
  gettimeofday(&tp2,&tzp2);
  long sec = tp2.tv_sec - tp.tv_sec;
  long usec = tp2.tv_usec - tp.tv_usec;
  if (usec < 0){
    sec--;
    usec = 1000000 + usec;
  }
  double ave = (sec * 1000000 + usec)/1000.00;
  printf("Average time per thread = %f microseconds\n",ave);
  cbox = ave;

  become(this);
}

void perf_start(Cbox<double>& cbox) {
  int t;
  Message *msg[1000];
  Perftest<Perftestops> perfstack;

  for (t = 0; t<999; t++){
    msg[t] = perfstack.perf_become();
    this->send(msg[t]);
  }
  msg[999] = perfstack.perf_end(cbox);
  this->send(msg[999]);
```

```
    gettimeofday(&tp,&tzp);
    become(this);
  }
};
//perftestops
#endif
```

This is the Raw Pthreads-only Thread creation time program:

```
extern "C" {
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>
}
#include "../../include/Lock.h"

int x;
struct timeval tp,tp2;
struct timezone tzp,tzp2;
Lock lock;

class dummy{
public:
  static void timeproc(){
    pthread_t tid;
    if (x < 1000){
      pthread_create(&tid,pthread_attr_default,
     (pthread_startroutine_t)dummy::timeproc,NULL);
      pthread_detach(&tid);
```

```
      x++;
    }
    else{
      gettimeofday(&tp2,&tzp2);
      lock.unlock();
    }
  }
};

main()
{
  pthread_t tid;
  double ave;
  double total;
  int loop;

  total = 0;
  for (loop = 0; loop < 10; loop++){
    x = 0;
    pthread_create(&tid,pthread_attr_default,
   (pthread_startroutine_t)dummy::timeproc,NULL);
    pthread_detach(&tid);
    lock.unlock();
    lock.lock();
    gettimeofday(&tp,&tzp);
    lock.lock();
    long sec = tp2.tv_sec - tp.tv_sec;
    long usec = tp2.tv_usec - tp.tv_usec;
```

```
    if (usec < 0){
      sec--;
      usec = 1000000 + usec;
    }
    ave = (sec * 1000000 + usec)/1000.00;
    printf("Average time per thread = %f microseconds\n",ave);
    total = total + ave;
  }
  total = total/10.00;
  printf("Total average = %f microseconds\n",total);
}
```

Appendix B

Context-Switching Time Program

usermain.C:

```
extern "C" {
#include <stdio.h>
}
#include "act++.h"
#include "switchops.h"
#include "switch.h"

usermain(int argc, char* argv[])
{
  int t;
  double average = 0.0;
  double temp;

  Switch<Switchops>  aSwitch;
  Acquaintance* SwitchActor = new Acquaintance ( aSwitch.Create());
  Message* msg[10];

  for (t = 0; t< 10; t++){
    Cbox<double> main_cbox;
    msg[t] = aSwitch.switch_start(main_cbox);
    SwitchActor->send(msg[t]);
```

```
    average = main_cbox + average;
  }

  average = average/10.0;
  printf("\nTOTAL AVERAGE for context switching = %f microseconds\n",average);
}
```

This is the file "switchops.h" and contains the behavior methods:

```
#ifndef  _switchops_h
#define  _switchops_h

extern "C" {
#include <sys/time.h>
}
#include "act++.h"
#include "switch.h"

struct timeval tp,tp2;
struct timezone tzp,tzp2;

Lock lock1,lock2;
int context_switch;

class Switchops : public  Behavior {
public:
  Switchops() {};

  void switch_end(Cbox<double>& cbox){
```

```
  gettimeofday(&tp2,&tzp2);
  long sec = tp2.tv_sec - tp.tv_sec;
  long usec = tp2.tv_usec - tp.tv_usec;
  if (usec < 0){
    sec--;
    usec = 1000000 + usec;
  }
  double ave = (sec * 1000000 + usec)/2001.00;
  printf("Average time per switch = %f microseconds\n",ave);
  lock2.unlock();
  lock1.unlock();
  cbox = ave;

  become(this);
}

void switch_become(){
  while(1){
    lock1.lock();
    lock2.unlock();
    context_switch++;
    if (context_switch == 2001)
    break;
  }
  become (this);
}

void switch_follow() {
```

```
    while (1){
      lock2.lock();
      lock1.unlock();
      context_switch++;
      if (context_switch == 2000)
break;
    }
    become(this);
  }

  void switch_start(Cbox<double>& cbox) {
    int t;
    Message *msg[1000];
    Message *msg2[1000];
    Switch<Switchops> switchstack;

    for (t = 0; t < 1; t++){
      msg[t] = switchstack.switch_become();
      this->send(msg[t]);
    }
    msg[2] = switchstack.switch_end(cbox);
    this->send(msg[2]);

    lock2.lock();

    Acquaintance* otherActor = new Acquaintance (switchstack.Create());
    for (t = 0; t < 1; t++){
      msg2[t] = switchstack.switch_follow();
```

```
      otherActor->send(msg2[t]);
    }
    context_switch = 0;
    gettimeofday(&tp,&tzp);
    become(this);
  }
};
//switchops
#endif
```

Appendix C

Quicksort Program

There were three types of quicksorts performed. The file proc_qsort.C contains the process quicksort code. The file qsortops.h contains the quicksort with cboxes, and qsortops2 contains the quicksort with Threadwait(). SortObj.h contains the class used in accessing the array of elements.

usermain.C:

```
/************************************************/
/*This program does a quicksort on lists of 500 -*/
/*1000 items (in increments of 100).  The times  */
/*for the sorts will be listed.  The program is  */
/*run two ways - thru the process, and with multiple*/
/*threads.  In the case of multiple threads, the */
/*number of threads created is determined by the */
/*the number of recursive calls to quicksort.    */
/************************************************/
extern "C" {
#include <stdio.h>
#include <sys/time.h>
}
#include "act++.h"
#include "SortObj.h"
#include "qsort.h"
#include "qsortops.h"
```

```
#include "qsort2.h"
#include "qsortops2.h"

extern void quicksort(int left, int right, SortObj& obj);

extern struct timezone tzp;
extern struct timeval tp;

void set_rnd_gen()
{
  struct timeval tv;
  struct timezone tz;

  gettimeofday (&tv,&tz);
  srandom(tv.tv_sec);
}

void make_list(int size, SortObj* listone, SortObj* listtwo, SortObj* listthree)
{
  int i,j;
  long u;
  set_rnd_gen();
  for (i = 1; i <= size; i++){
    u = random();
    j = u % 10000;
    listone->place_val(i,j);
    listtwo->place_val(i,j);
    listthree->place_val(i,j);
```

```
  }
}

usermain(int argc, char* argv[])
{
  int  loop,num;
  struct timezone tzp2;
  struct timeval tp2;

  num = 1000;
  printf("Number of threads max for multiple threads case = %d\n",num);

  printf("Elements        1 Thread          Many Threads          Many Threads\n");
  printf("Elements                           with Cbox             w/o Cbox\n");
  for (loop = 500; loop <= 1000; loop = loop + 100){
    SortObj list1(loop),list2(loop),list3(loop);
    Qsort<Qsortops> interface;
    Qsort2<Qsortops2> interface2;
    Cbox<int> done_cbox;

    make_list(loop,&list1,&list2,&list3);

/*do the process quicksort*/
    gettimeofday(&tp,&tzp);
    quicksort(1,loop,list1);
    gettimeofday(&tp2,&tzp2);
    long sec = tp2.tv_sec - tp.tv_sec;
    long usec = tp2.tv_usec - tp.tv_usec;
```

```
    if (usec < 0){
      sec--;
      usec = 1000000 + usec;
    }
    usec = usec + 1000000 * sec;
    list1.listcheck();
/*done with process quicksort*/

/*do the multithreaded quicksort with cboxes*/
    Acquaintance* acquaintance = new Acquaintance(interface.Create(),num);
    Message* msg = interface.qsort_start(1,loop,list2,done_cbox);
    acquaintance->send(msg);
    int s = done_cbox;
    Threadwait(); /*so can get thread counts*/
    gettimeofday(&tp2,&tzp2);
    long sec2 = tp2.tv_sec - tp.tv_sec;
    long usec2 = tp2.tv_usec - tp.tv_usec;
    if (usec2 < 0){
      sec2--;
      usec2 = 1000000 + usec2;
    }
    usec2 = usec2 + 1000000 * sec2;
    list2.listcheck();
/*done with multithreaded quicksort*/

/*do the multithreaded quicksort without cboxes*/
    Acquaintance* acquaintance2 = new Acquaintance(interface2.Create(),num);
    Message* msg2 = interface2.qsort_start(1,loop,list3);
```

```
    acquaintance2->send(msg2);
    Threadwait();
    gettimeofday(&tp2,&tzp2);
    long qsec2 = tp2.tv_sec - tp.tv_sec;
    long qusec2 = tp2.tv_usec - tp.tv_usec;
    if (qusec2 < 0){
      qsec2--;
      qusec2 = 1000000 + qusec2;
    }
    qusec2 = qusec2 + 1000000 * qsec2;
    list3.listcheck();
/*done with multithreaded quicksort*/

    sec = usec/1000000;
    sec2 = usec2/1000000;
    qsec2 = qusec2/1000000;

    usec = usec - sec * 1000000;
    usec2 = usec2 - sec2 * 1000000;
    qusec2 = qusec2 - qsec2 * 1000000;

    printf("%d          %d sec %d usec    %d sec %d usec    %d sec %d usec\n",
   loop,sec,usec,sec2,usec2,qsec2,qusec2);
  }
}
```

The process quicksort file "qsort.C":

```
/*this file is for the process run quicksort*/
```

```
/*algorithm is the same, but no actor stuff */
/*is here*/

extern "C" {
#include <stdio.h>
}
#include "SortObj.h"

int partition(int left,int right,int pivot,int pivotindex,SortObj& object){
  if (right == (left + 1)){
    if (object.elmnt(right) < object.elmnt(left))
      object.swap(left,right);
    return(right);
  }
  else{
    while (left < right){
      while ((object.elmnt(left) <= pivot) && (left < pivotindex)){
left++;
      }
      if (right > left){
while ((object.elmnt(right) >= pivot) && (right > pivotindex)){
  right--;
}
      }
      if (left < right){
object.swap(left,right);
if (left == pivotindex){
  pivotindex = right;
```

```
  left++;
}
else if (right == pivotindex){
  pivotindex = left;
  right--;
}
      }
    }
    return(right);
  }
}

void quicksort(int left,int right,SortObj& object)
{
  int pivot,pivotindex,k;

  if (right > left){
    pivotindex = left + (right - left)/2;  /*get middle element*/
    pivot = object.elmnt(pivotindex);
    k = partition(left,right,pivot,pivotindex,object);
    if (k != left){
      quicksort(left,k-1,object);
      quicksort(k,right,object);
    }
    else if (k == left){
      quicksort(k+1,right,object);
    }
  }
```

```
}
```

The Cbox quicksort behavior methods in "qsortops.h":

```
/*this file is for the multi-threaded cbox quicksort*/

#ifndef qsortops_h
#define qsortops_h
extern "C" {
#include <stdio.h>
}
#include "act++.h"
#include "qsort.h"

struct timeval tp;
struct timezone tzp;

class Qsortops : public Behavior {
 private:
  int partition(int left,int right,int pivot,int pivotindex,SortObj& object){
    if (right == (left + 1)){
      if (object.elmnt(right) < object.elmnt(left))
object.swap(left,right);
      return(right);
    }
    else{
      while (left < right){
while ((object.elmnt(left) <= pivot) && (left < pivotindex)){
  left++;
```

```
}
if (right > left){
  while ((object.elmnt(right) >= pivot) && (right > pivotindex)){
    right--;
  }
}
if (left < right){
  object.swap(left,right);
  if (left == pivotindex){
    pivotindex = right;
    left++;
  }
  else if (right == pivotindex){
    pivotindex = left;
    right--;
  }
}
      }
      return(right);
    }
  }

 public:
  Qsortops() { }

  void qsort_start(int left,int right,SortObj& object,Cbox<int>& cbox)
    {
      Qsort<Qsortops> sort_interface;
```

```
      Message *msg = sort_interface.quicksort(left,right,object,cbox);
      this->send(msg);
      gettimeofday(&tp,&tzp);
      become(this);
    }

  void quicksort(int left,int right,SortObj& object,Cbox<int>& cbox)
    {
      int pivot,pivotindex,k;
      Qsort<Qsortops> sort_interface;
      Message *msg1,*msg2;
      Cbox<int> done_cbox,done_cbox2;

      become(this);
      if (right > left){
pivotindex = left + (right - left)/2;  /*get middle element*/
pivot = object.elmnt(pivotindex);
k = partition(left,right,pivot,pivotindex,object);
if (k != left){
    msg1 = sort_interface.quicksort(left,k-1,object,done_cbox);
    msg2 = sort_interface.quicksort(k,right,object,done_cbox2);
    this->send(msg1);
    this->send(msg2);
    int t = done_cbox; /*so we know when the threads below us finished*/
    int t2 = done_cbox2;
  }
else if (k == left){
```

```
    msg1 = sort_interface.quicksort(k+1,right,object,done_cbox);
    this->send(msg1);
    int t = done_cbox; /*so we know when the threads below us finished*/
  }
      }
      cbox = 1;
    }
};
#endif
```

The quicksort behavior methods using Threadwait() in "qsortops2.h":

```
/*this file is for the multi-threaded quicksort*/
/*using Threadwait()*/

#ifndef qsortops2_h
#define qsortops2_h
extern "C" {
#include <stdio.h>
}
#include "act++.h"
#include "qsort2.h"

extern struct timeval tp;
extern struct timezone tzp;

class Qsortops2 : public Behavior {
 private:
  int partition(int left,int right,int pivot,int pivotindex,SortObj& object){
```

```
    if (right == (left + 1)){
      if (object.elmnt(right) < object.elmnt(left))
object.swap(left,right);
      return(right);
    }
    else{
      while (left < right){
while ((object.elmnt(left) <= pivot) && (left < pivotindex)){
  left++;
}
if (right > left){
  while ((object.elmnt(right) >= pivot) && (right > pivotindex)){
    right--;
  }
}
if (left < right){
  object.swap(left,right);
  if (left == pivotindex){
    pivotindex = right;
    left++;
  }
  else if (right == pivotindex){
    pivotindex = left;
    right--;
  }
}
      }
      return(right);
```

```
    }
  }

 public:
  Qsortops2() { }

  void qsort_start(int left,int right,SortObj& object)
    {
      Qsort2<Qsortops2> sort_interface;

      Message *msg = sort_interface.quicksort(left,right,object);
      this->send(msg);
      gettimeofday(&tp,&tzp);
      become(this);
    }

  void quicksort(int left,int right,SortObj& object)
    {
      int pivot,pivotindex,k;
      Qsort2<Qsortops2> sort_interface;
      Message *msg1,*msg2;

      become(this);
      if (right > left){
pivotindex = left + (right - left)/2;  /*get middle element*/
pivot = object.elmnt(pivotindex);
k = partition(left,right,pivot,pivotindex,object);
if (k != left){
```

```
    msg1 = sort_interface.quicksort(left,k-1,object);
    msg2 = sort_interface.quicksort(k,right,object);
    this->send(msg1);
    this->send(msg2);
  }
else if (k == left){
    msg1 = sort_interface.quicksort(k+1,right,object);
    this->send(msg1);
  }
      }
    }
};
#endif
```