ACT++: Building a Concurrent C++ with Actors

Dennis Kafura and Keung Hae Lee

TR 89-18

ACT++: Building a Concurrent C++ with Actors

Dennis Kafura Keung Hae Lee

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

kafura@vtopus.cs.vt.edu keung@vtopus.cs.vt.edu

Abstract

ACT++ (Actors in C++) is a concurrent object-oriented language being designed for distributed real-time applications. The language is a hybrid of the actor kernel language and the object-oriented language C++. The concurrency abstraction of ACT++ is derived from the actor model as defined by Agha. This paper discusses our experience in building a concurrent extension of C++ with the concurrency abstraction of the actor model. The current design of ACT++ and its implementation are described. Some problems found in the Agha's actor model are discussed in the context of distributed real-time applications. The use of ACT++ disclosed the difficulty of combining the actor model of concurrency with class inheritance in an object-oriented language.

1. Introduction

ACT++ is a concurrent object-oriented language being developed as a part of a research project on concurrent object-oriented real-time systems [Kafura 88, 89]. This project recognizes that concurrent object-oriented programming has features which closely correspond to several important aspects of distributed real-time systems. Among these corresponding elements are the following. The relationship between real-world objects and their software counterparts is made more explicit with an object-orientation. Communication via message passing, inherent in distributed real-time systems, is reflected in the message passing metaphor used in object-oriented languages. The interactions among concurrent, interacting real-world entities are more naturally and correctly captured by a language designed for expressing and controlling concurrency among autonomous objects. Evolving real-time software, responding to changes in requirements or other system

components, is more productively managed within the organizing structure of a class hierarchy.

The actor model of concurrent computation was first introduced by Hewitt [Hewitt 77] and extended by many others [Hewitt and Baker 77, Hewitt and Atkinson 79, Clinger 81, Hewitt and deJong 83]. More recently, [Agha 86] defined an actor model with a small number of powerful primitives. Since the introduction of the actor model by Hewitt, many languages have been proposed for programming concurrent computation using actors. Most early language designs were intended for AI programming, being derivatives of Lisp [Theriault 83, Attardi 87, Liberman 87]. Recently, several language designs in non AI domains have adopted the actor model as the basis of concurrency [Barry and Thomas 87, Athas and Boden 88, Yonezawa et al. 87, Delagi and Saraiya 88].

ACT++ represents our effort to test the utility of the actor style of programming in the domain of real-time systems applications, where programs are usually written in more conventional languages. One of our main goals in the design of ACT++ is to develop a language which supports the powerful actor concurrent computation model and provides software reusability through the class inheritance of an object-oriented language. Since the language is intended for exploring the actor style of programming and object-oriented programming with class inheritance, a requirement imposed on the current ACT++ design is that it should lend itself to an inexpensive implementation. The current ACT++ design extends C++ [Stroustrup 86] with a class hierarchy which provides the abstraction of the actor model of concurrency. This paper describes this design and implementation of ACT++.

In Section 2, we present the actor model developed by Agha. Some implications of the model properties are also discussed. Section 3 describes the design and implementation strategy of ACT++. Adapting Agha's actor model is motivated by our long-term goal of using ACT++ in distributed real-time programming. Example programs written in ACT++ are presented in Section 4. Section 5 details the implementation based on a C++ class hierarchy. Section 6 reviews other work that is related to this research. Future research issues are discussed in Section 7.

2. The Actor Computation Model

The actor model [Agha 86] is a concurrent computation model in which computation is achieved by actors communicating via message passing. The actor model consists of five basic elements: actors, mail queues, messages, behaviors, and acquaintances.

An *actor* is a self-contained active object. Interaction among actors can occur only through message passing. Each actor is associated with a unique *mail queue*, whose address serves as the identifier of the actor. The messages sent to an actor are buffered by its mail queue. Messages buffered in a mail queue are read one at a time in a strict FIFO order. If the message queue is empty when an actor is ready to receive the next message, the actor is blocked until a message arrives in the mail queue. An actor A can send a message to another actor B only if A knows the mail queue address of B. Mail queue addresses themselves may be passed as a part of a message. These last two properties allow the connectivity among actors to be dynamically reconfigured.

The *behavior* of an actor determines how the actor reacts to a request specified in the message being processed. A behavior is defined by a code body called the *behavior script*. The script contains method definitions, and *acquaintances*. Acquaintances are the names of other actors to which an actor can send messages. A behavior script corresponds to a class definition in other object-oriented languages while acquaintances correspond to instance variables. However, acquaintances of Agha's actor are read-only variables. There is a good reason for the *read-only* requirement. The actor model assumes *inherent concurrency*, which means that every statement of a behavior script is executed concurrently except when there is a sequencing constraint required by causal ordering. The inherent concurrency would not be possible without a guarantee that there is no data dependency among instructions. Hence, the actor model provides only side-effect free operations. For example, assignment operations are not allowed as in functional programming languages.

In processing a message, an actor can do three things: create new actors, send messages to actors, and specify a replacement behavior. Actors are dynamically created at run-time as needed during the course of computation. A new actor is created by the **new** operation. The mail queue address of the created actor is returned as a result. Other actors can send messages to the new actor using this mail queue address. Since the creator of a new actor can pass the mail queue address of the new actor in a message, the existence of a newly created actor can become known to other existing actors.

The **send** operation is the primitive used for asynchronous message passing. A message consists of the address of the target actor, the name of the method to be invoked, and parameters for the method invocation. An actor is not blocked by a message sending operation since the mail queue of the receiving actor buffers incoming messages.

The last primitive operation is the specification of a *replacement behavior*. The operation, called **become**, is the most novel concept in the actor model. The difficulty in understanding the actor model comes mainly from this unusual notion of a replacement behavior as explained below.

To avoid ambiguous terminology, let us distinguish the following entities: actor, behavior, and thread. The following definitions deviate from Agha's terminology used in the actor model. An actor, which is identified by a mail queue address, is a combination of a mail queue and all the activities and data structures associated with the mail queue. A *behavior script*, or *behavior* for short, determines what actions an actor may take when processing a message. A *thread* denotes a behavior script in execution. A thread has a list of acquaintances which constitute the local execution environment of the thread. The *behavior of a thread* refers to the behavior script executed by the thread. In Agha's model, a behavior receives exactly one request message.

The mechanism of replacement behavior allows an actor to change its behavior dynamically at run time. An actor can be bound to different behaviors at different points in time. These behaviors may not have the same methods or internal data structures. The only entity which does not change in the life of an actor is its mail queue address. For this reason, we define the *current behavior* to be the behavior of the most recently created thread of the actor. Similarly, the *state of an actor* is defined to be the state of the current behavior. If we need to refer to the behaviors of an actor which are not the current behavior, we call them *old behaviors*. Figure 1 shows a conceptual view of an actor.

We now describe the **become** operation: the operation used for specifying a replacement behavior. The become operation requires a behavior script name and a list of acquaintances. The operation creates a thread. The newly created thread is the current behavior of the actor. The replacement behavior determines how to handle the next unprocessed message. Since a thread can specify a replacement only once in its life, there is at most one behavior waiting for a message at any time.

The become operation plays two important roles in the actor model. The first is that the become operation is the mechanism by which an actor changes its state. Since actor operations have no side-effects, acquaintances cannot be modified by assignment. An actor changes its state by becoming another behavior with different acquaintances. It is possible to view each behavior as an object by itself which has a much smaller life-time compared with active objects in other concurrent object-oriented languages. This view naturally extends the replacement behavior to one with different methods and data structures, not to mention acquaintances whose values are different from those of old behaviors.

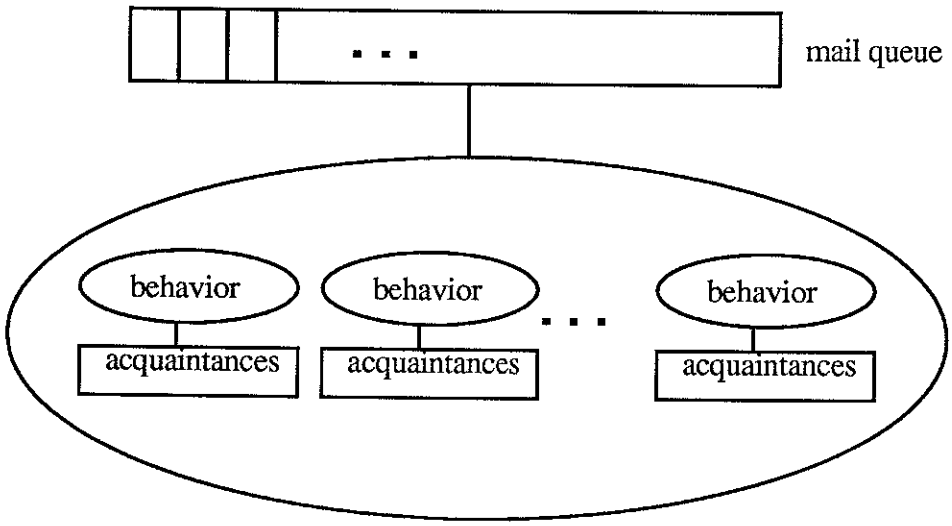

Figure 1 - Conceptual View of an Actor

The second important use of the become operation is as a synchronization mechanism. Besides the synchronization provided by message passing, the become operation is the only explicit synchronization primitive provided in the actor model. The consistency of the state of an actor is maintained by the use of the become operation. The combination of buffering provided by the mail queue and the become operation offers a synchronization mechanism which is similar to a *serializer* proposed in [Hewitt and Atkinson 79]. The become operation serializes the invocations of methods of an actor because the specification of a replacement behavior signals the mail queue to let the next request message enter the protected region of the actor. For example, consider an actor whose state is modified by the current behavior. The actor must prevent its state from being accessed by other invocation

threads until the modification is complete. The mail queue must block other requests until the modified state becomes available. Specifying a replacement behavior is the mechanism for informing the mail queue that the current behavior has finished computing the state of the actor, and so the next thread can thus be created.

A significant result of the become operation is the potential for concurrency inside an actor. Intra-object concurrency naturally arises when the current behavior continues after it specifies the replacement behavior. In this case, the two or more threads proceed concurrently.

3. Building Actors in C++

3.1 The Actor Model in ACT++

The effectiveness of the instruction level fine-grain concurrency of Agha's actor model is predicated on the availability of a large number of processors with extremely fast interprocessor communication. Instruction-level concurrency will not be effective on a system of a relatively small number of loosely coupled processors. Since the long-term application domain of ACT++ is distributed real-time systems, the fine-grain concurrency offered by the actor model is not desirable. Hence, we did not choose the instruction-level concurrency as a fundamental property of ACT++. An ACT++ behavior in execution is a sequential process called an *ultra-light process*. The term ultra-light derives from the fact that the code size and the life-time of a behavior is on the order of a procedure. The decision to remove the instruction-level concurrency from C++ has several significant impacts.

Removal of instruction-level concurrency assumption makes actors in our model more compatible with imperative language constructs. Method definitions are no longer subject to the requirement of side-effect free operations. Acquaintances, which are called instance variables in ACT++, can be written to. Hence, an actor can change state by directly modifying its instance variables with assignment operations. There is no loss of intra-actor concurrency at the level of behaviors. Concurrency in ACT++ comes from two sources: creation of a new actor and specification of a replacement behavior. The former results in inter-object concurrency, while the latter yields intra-object concurrency.

Another significant impact is that ACT++ encourages actors to be more coarse-grained objects. The become operation in Agha's actor model forces the data structures contained in

Actor

The language supports the primitive operations of the Agha's actor kernel language: **new**, **send**, and **become**. By declaring a class as a direct or indirect subclass of a built-in class ACTOR, one can define a behavior script, which is also called an *actor class* in ACT++. A new actor is created using a **New**[1] operation. The **New** operation creates an actor, which is a pair of a new mail queue and an instance of the specified actor class. The address of the mail queue is returned as the result.

A behavior script is written using both the actor primitives and the imperative language constructs of C++. The instructions in a behavior script are executed sequentially. An actor can specify itself as the replacement behavior by specifying ***self*** as the replacement behavior. In this case, the replacement behavior is created using a copy of the instance variables of the current behavior.

Message Passing

Invoking methods of passive objects has the semantics of a function call and will not be discussed further. Invoking a method of an actor is via asynchronous message passing. The asynchronous message passing in ACT++ is supported through predefined objects called *mail boxes*. Two types of messages are distinguished in ACT++, namely, *request messages* and *reply messages*. A request message is used for invoking a method while a reply message is used for the delivery of the result of a method invocation. Two types of mail boxes are available to support these two different kinds of messages: *Mbox* and *Cbox*. Mbox models the mail queue of the actor model while Cbox allows for programming with *futures* [Lieberman 87].

Request Messages and Mbox

A *request message* is used for sending a request to another actor. A request message consists of the name of a method to be executed by the receiving actor and arguments for invoking the method. A request message is sent by a **send** operation. The send operation needs the Mbox of the receiver and the message to be sent.

Request messages are buffered in the mail queue (Mbox) of the receiving actor. An actor can refer to its own Mbox using the pseudo variable self. Each behavior of an actor can process only one request message in the Mbox. Since a thread is created only if the mail queue has a message for the thread, the operation of receiving a request message from the

[1] To distinguish it from the new operation of C++, we write the first character in upper case.

actors to be very small because an actor specifying a replacement behavior must enumerate all the acquaintances to be included in the local execution environment of the replacement behavior. In the primitive actor model, programming can quickly become complex even with a small increase in the number of acquaintances. Hence, programmers are discouraged from gathering related data items in one actor. Agha's modeling of a stack, where each item of a stack is modeled as an actor, reflects this phenomenon.

The coarse actor granularity in ACT++ is also more compatible with class inheritance. Since a subclass typically needs its own instance variables in addition to inherited instance variables, the number of instance variables in a class increases as the class hierarchy grows deeper. If all instance variables must be explicitly specified in the become operation, it will be prohibitive to build a class hierarchy of more than one or two levels.

Objects

Two kinds of objects are distinguished in ACT++, namely *active objects* and *passive objects*. An *active object* proceeds concurrently with other objects. An active object, possessing its own thread of control, is an actor. All objects that are not actors are passive objects. A passive object does not have its own thread of control. The invocation of a method of a passive object is performed using the thread of the requesting object. The sender of a message to a passive object is blocked until the invocation is complete. Since a passive object has no power of controlling invocations of its methods, concurrent invocations of its methods may cause the state of the object to be inconsistent. Hence, a passive object must be known only to a single actor. The reference to a passive object cannot not be exported to other actors. Passive objects are used to build an actor, being private to the actor. All shared objects are defined to be actors. This restriction is enforced in the current implementation only by convention.

Class

Each object is an instance of some class. A class defines the properties of its instance objects. A class can inherit from another existing class by declaring itself as a subclass of the existing class. A subclass can redefine, restrict, or extends the definition of its superclass. Active objects are instances of classes which are direct or indirect subclasses of a special class called *ACTOR*. Passive objects are instances of classes which are not subclasses of ACTOR.

actor may assume that every client actor provides the same method to receive a reply. Yet, the script of the sender of a request gets complex if it needs to receive more than one reply. For example, the *function-apply* actor that needs two inputs as defined in [Agha 86] had to create continuations in order to maintain an explicit state transition to differentiate the two inputs received using the same method. The Cbox mechanism simplifies the sender/receiver protocol by obviating the necessity to define a separate method for receiving a reply message.

The third reason is to allow for an efficient implementation of an actor which cannot proceed until a reply is received. The *call* and *reply* constructs in Agha's SAL language abstract away the details of the protocol for receiving a reply by requiring a specialized service from the compiler. The compiler is responsible for translating a *call* expression into a continuation which will wait for the reply. However, in the primitive actor model, the initiating actor becomes inactive by means of an *insensitive actor* until the reply is forwarded by the continuation. An actor, *A*, becomes insensitive by assuming a behavior which forwards all incoming messages to an auxiliary buffer actor. This forwarding continues until the message arrives which contains the result of the initiated operation. At this time, *A* sends the buffer actor a message directing the buffer to retransmit all queued messages back to *A*.

The Cbox mechanism is more flexible than insensitive actors because an actor may create many Cboxes before waiting on any of them. Alternatively, a Cbox may be used for receiving the same type of reply messages sent by multiple actors. Implementation of and-synchronization and or-synchronization are directly supported by the Cbox mechanism. An or-synchronization is implemented by reading only one reply message from a single Cbox to which multiple reply messages may be delivered. Attempts to read a message prior to the delivery of the message will cause the reader to block. The actor will be resumed if a reply from any sender is received. An and-synchronization of n events may be achieved by the action of reading reply messages from a Cbox n times consecutively. The actor reading the messages will be suspended and resumed repeatedly until all n messages are received by the Cbox[1].

[1] A further optimization can be achieved by extending the semantics of receiving from a Cbox. A possible extension to the current design is to provide a Cbox operation which takes the number of events and suspends the actor until all of the specified number of replies are received by the Cbox.

sending actor is implicit in a behavior script. Once the execution of a behavior begins, the message to be processed by the behavior is available and is read by a hidden **receive** operation on self. A behavior is not blocked by the **receive** operation performed on self since the thread of a behavior is not created until a message becomes available in its mail queue.

Reply Messages and Cbox

If the sender of a request message wants to receive the result of the method invocation, it may provide a Cbox name (see below) in the request message. The reply operation is used to transmit a reply message containing the result. The name of a Cbox specified in a request message is called the *reply destination* [Yonezawa et al. 87]. Since an actor knows the reply destination when it reads a request message, the reply destination needs not be explicitly provided by the programmer in the **reply** operation. If the sender, *A*, does not provide its own Cbox in a request message, a *reply forwarding* occurs. The reply is not delivered to the actor *A*. It is directly delivered to the actor who sent the current request message being processed by the actor *A*.

A Cbox in ACT++, named after the Cbox structure of Concurrent Smalltalk [Yokote and Tokoro 86], implements "waiting by necessity" [Caromel 88]. The Cbox is a special mailbox created to receive the reply resulting from an initiated operation. An actor can read from a Cbox using the **receive** operation. If a reply is available in the Cbox, it is immediately delivered to the actor. Otherwise, the **receive** operation blocks the caller until a reply arrives. An actor can check if the reply has arrived in a Cbox using the **in** operation on the Cbox.

Cbox was designed for three reasons. One reason is to avoid an actor which needs a reply being split into multiple fine-grained continuations. A Cbox provides a communication channel which allows the sender of a request message to receive a reply message directly from the server of the request without creating separate actors to implement a continuation waiting on a reply message.

The second motivation is to reduce the complexity of defining a behavior which needs to receive the result of the operation which the actor initiates. Using a single mail box for both request messages and reply messages requires a server actor to have some a priori knowledge of the client actor. The server must know what method of the client to invoke in order to pass the reply to the client. The a priori knowledge can be minimized if the server

It is also possible to create a customer actor to implement a continuation. One can create a customer actor by passing a Cbox as an argument of the initial behavior used by a New operation. The creator of a customer actor can have the reply for a request message sent to the customer by specifying the Cbox passed to the customer as the reply destination of the request message.

3.2 Development Strategy for ACT++

For an experimental language like ACT++, one is faced with the decision of whether to implement a translator for the new language. Since the primary use of ACT++ is exploratory, the detailed syntax for the language was considered of less importance than the general abstractions provided by the language. Therefore, ACT++ is built upon an existing object-oriented language syntax. The mechanisms for creating class hierarchies found in object-oriented languages make it possible to create abstractions that support actor concurrency without changing the definition of the base language. As explained below, C++ was chosen as the base language.

An implementation based on a language like C++ which supports class inheritance has several advantages. First, the compiler and other tools for C++ are directly available for programming in ACT++. We can avoid committing the effort to developing translators for premature language designs. The compiler development effort can be postponed until the language design has proven to be useful and stable. Second, since the base language supports class inheritance, the inheritance mechanism can be conveniently incorporated into ACT++. Although our experiments revealed that this expectation was only partly met by the current design [Kafura and Lee 89], class inheritance of the base language is provided to ACT++ programmers. Third, experimenting with the ACT++ language will identify deficiencies in the design. Evolutionary changes in the design are more easily accommodated with the modularity and reusability encouraged by class hierarchies. The reasons for choosing C++ as the base language are as follows: C++ is efficient, strongly-typed, and widely available. The strong support of overloading provided by C++ turned out to be a powerful tool for a syntactic extension of the language. C++ will also be the base language for the operating system kernel being designed for the execution of ACT++ programs.

3.3 Class Hierarchy

The major difficulty in combining C++ and actors lies in incorporating the asynchronous message passing mechanism into C++ which has no notion of either message passing or

concurrency but does insist on strong typing. The class hierarchy used in the current implementation is shown in Figure 2. Only the most important aspects of this hierarchy are given here. A more detailed discussion of these classes is provided in Section 5. An ACT++ programmer needs to know about only three classes: ACTOR, Mbox, and Cbox. Other classes found in the hierarchy are implementation details and are transparent to an ACT++ programmer. While ACT++ specific operations except New are directly supported as methods of the three classes, some operation names were beautified with a spoonful of syntactic sugar. The additional operations supported by ACT++ are summarized in Figure 3. We now describe each of these operations.

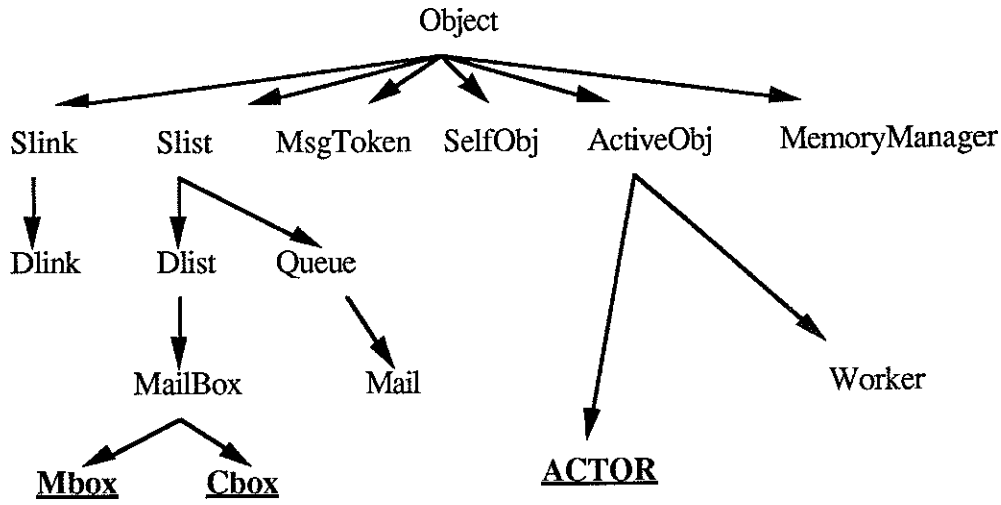

Figure 2 - ACT++ Class Hierarchy

The **New** operation for creating a new actor is a function which takes the name of an actor class as an argument. It would be appropriate if the New operation were to be provided as a class method of each actor class. However, since a class in C++ is not a run-time object unlike in Smalltalk-80, the New operation is implemented as a conventional function without being a member of any class. The **become** operation is a method of ACTOR. The operation is overloaded to allow an actor to specify self as the replacement behavior. The **send** operation is implemented as a method of the class Mbox. For cosmetic reasons, the syntax for sending a message is actually written as a series of stream-like output operations (<<). For example, the following statement sends to an Mbox called myMbox a message

consisting of aMethod, replyDestination, and two method invocation parameters, namely, firstInteger and secondInteger:

```
myMbox << aMethod << replyDestination << firstInteger << secondInteger;
```

An actor can read a request message from its Mbox using the operation >> on the pseudo variable self. Since the method name argument of a request message is used by the mail queue for selecting the method to invoke, the method name is not read by the >> operation. Similarly, the reply destination argument is not read by >>. The >> operation is used only to read the values for the parameters of the method. Each method of a behavior script begins with a >> statement. For example, continuing the example above, the definition of the method aMethod of an actor myMbox uses the following for reading a message from the Mbox:

```
self >> firstIntegerArg >> secondIntegerArg;
```

Operation	Class	Meaning
New	none	Create a new actor
become	ACTOR	Specify a replacement behavior
<<	Mbox,Cbox	Send a message to a mail box
>>	Mbox,Cbox	Receive from a Cbox or Self
reply	Cbox	Send a reply message
in	Cbox	Check if a Cbox is not empty

Figure 3 - Summary of ACT++ Operations

The class Cbox supports the **reply** method for transmitting a reply message. Since the reply destination is known when a request message is processed, the Cbox name is provided implicitly by the language. The following statement sends a reply message containing 23 to the reply destination:

serialized actor. The behavior script of a serialized actor must specify a replacement behavior at some point in its computation.

4.1 A Concurrent Factorial Actor

Figure 4 shows a concurrent factorial actor written in ACT++. The class name for the concurrent factorial actor is ConcFact. Since the class ConcFact is a subclass of ACTOR (line 3), the instances of ConcFact are active objects. The function main() is the driver of the program where the action of computing 20! starts. ACT++ regards the main() function as an actor without methods. In the main() function, the operation **New** creates a new actor whose initial behavior is defined by the class ConcFact (line 16). The **New** operation returns the address of an Mbox, which is bound to the Mbox variable factorial. The main() function sends a request message to factorial using << operators (line 17). The message contains the method name to be invoked, the reply destination myCbox, and a seed value n. It then receives the result from myCbox (line 18). The main() is blocked until the Cbox receives the result. The use of a Cbox allows the main() to receive the result directly from the actor factorial. Without a Cbox, it would be necessary to define an extra actor to receive and print the final result.

ConcFact uses another actor class called RangeProduct. The request message is read using the >> operation on self (line 24). ConcFact simply reformats the request message and forward the request to a RangeProduct actor (line 26). Note that ConcFact does not pass its own Cbox name in the request message sent to a RangeProduct actor. This is because ConcFact wants to have RangeProduct send the reply directly to main(). When RangeProduct performs a reply operation (line 42), it will send the reply to myCbox of main() since ConcFact does not change the reply destination.

The RangeProduct multiplies all numbers in the range specified by its two input arguments. The algorithm of RangeProduct is based on a divide-and-conquer technique. On receiving a request, a RangeProduct actor checks if the range contains only one number (line 32). If this condition is true, the lower limit is returned (line 33). Otherwise, the midpoint determining two sub-ranges is computed (line 35). To compute the products of these two sub-ranges concurrently, two new RangeProduct actors, rp1 and rp2 are created (lines 36-37). Each of rp1 and rp2 is sent a request message containing one sub-range (lines 39-40). This process continues until the sub-range being computed by a RangeProduct actor contains only one number.

```
1   #include "act.h"                          // include the ACT++ kernel classes
2
3   class ConcFact: ACTOR {
4   public:
5     void compute_factorial();
6   };

7   class RangeProduct: ACTOR {
8   public:
9     void compute_product();
10  };

11  main()
12  {
13    int k;
14    int n = 20;    /* compute 20!  */
15    Cbox myCbox;
16    Mbox factorial = New(ConcFact);        // bind it to an instance of ConcFact;
17    factorial << &ConcFact::compute_factorial << myCbox << n;
                                             // send a request message
18    myCbox >> k;                           // receive a reply from factorial
19    printf("%d\n", k);
20  }

21  void ConcFact::compute_factorial()          // a method of ConcFact
22  {
23    int m;
24    self >> m;                             // read a request message
25    Mbox rp1 = New(RangeProduct);
26    rp1 << &RangeProduct::compute_product << 1 << m;
27  }

28  void RangeProduct::compute_product()            // a method of RangeProduct
29  {
30    int low, mid, high, sub1, sub2;
31    self >> low >> high;
32    if (low>=high)
33        reply(low);
34    else {
35        mid = (low+high) / 2;
36        Mbox rp1 = New(RangeProduct);
37        Mbox rp2 = New(RangeProduct);
38        Cbox subCbox;
39        rp1 << &RangeProduct::compute_product << subCbox << low  << mid;
40        rp2 << &RangeProduct::compute_product << subCbox <<mid+1<<high;
41        subCbox >> sub1 >> sub2;
42        reply(sub1*sub2);
43    }
44  }
```

Figure 4 - Concurrent Factorial Actor in ACT++

```
reply(23);
```

Two other methods are provided by a Cbox: **in** and **receive**. The method **in()** is used to check whether or not the reply has arrived in a Cbox. The receive operation denoted by '>>' reads a reply from a Cbox. The use of these methods are illustrated in the following:

```
if (myCbox1.in()) then
        myCbox1 >> anInteger
else
        myCbox2 >> anInteger;
```

If myCbox1 has not yet received a reply, then myCbox2 is read, in which case the actor executing the statement may be blocked depending on the presence of a reply message in myCbox2.

An or-synchronization in which the actor waits for an event, either a or b, to occur is illustrated in the following:

```
eventCbox >> anyEvent;
do-after-synch-stuff;
```

An and-synchronization in which an actor waits for all three events to occur is illustrated in the following:

```
eventCbox >> event1 >> event2 >> event3;
do-after-synch-stuff;
```

4. Example Programs

Two example programs written in ACT++ are presented in Figure 4 and 5. The first program which computes 20! is an example of actors with no acquaintances. Such actors are called *unserialized* actors. Since no time-dependent errors can occur in invoking an unserialized actor, the language assumes that an unserialized actor has a hidden become operation at the beginning of its script so that the actor can immediately serve the next request in the mail queue. The second example, a cruise control problem, illustrates a

Two sub-range product will be eventually delivered to subCbox of the Range-Product actor which was created earliest. The actor multiplies the two sub-range products and sends the result to its reply destination, myCbox of the main() function. It is also possible to use a separate Cbox for each sub-range product. For this particular problem, distinguishing the two replies is not necessary. Hence, a single Cbox was used.

4.2 A Cruise Control Problem

The following is an adapted definition of a cruise control problem described in [Gehani 83].

> A cruise control maintains an automobile at a constant speed selected by the driver. The cruise control mechanism is set in action by driving the automobile at the desired cruising speed and then pushing the cruise control button. The cruise control mechanism is disengaged by depressing either the brake or the accelerator. When neither of these are depressed the cruise control mechanism must check the current speed and adjust the throttle to the requested cruising speed. It takes 0.1 second for the throttle to adjust its opening.

The modeling of the cruise control problem includes actors representing accelerator, cruise button, brake, speedometer, throttle, and timer. The cruise control actor also needs to be defined. We show only the definition of the cruise control actor. The cruise control actor may be in one of two states at any time: the cruise control is on or off. These states are defined by actor classes named CruiseOn and CruiseOff, respectively. From the description of the problem, we know that the cruise control actor may receive a request message from the accelerator, cruise button, and brake. We also find that the cruise controller must know the throttle actor, speedometer, and timer. We assume that there exists some low level I/O mechanism which allows the actors to interact with the physical devices. The cruise button actor sends an on() message while the accelerator and the brake send an off() message to the cruise control actor. We believe that the intended meaning of the code is obvious and so do not provide a separate description.

```
#include "act.h"
class CruiseOff;
class CruiseOn;

class CruiseOn : ACTOR {
    Mbox mySpeedometer, myThrottle, myTimer;      // instance variables
    int cruisingSpeed;
public:
CruiseOn (Mbox aSpeedometer, Mbox aThrottle, Mbox aTimer, int aSpeed)
    { // constructor
        mySpeedometer = aSpeedometer;
        myThrottle = aThrottle;
        myTimer = aTimer;
        cruisingSpeed = aSpeed;
    }
void off()
    { // turn off cruise control
        become(new CruiseOff(mySpeedometer, myThrottle, myTimer));
    }
void on()
    { // control already on; no action required; ignore this request }
void maintain()
    { // try to maintain the specified cruising speed
        int currentSpeed, aSignal;
        Cbox speedCbox, timerCbox;
        mySpeedometer << &Speedometer::speed << speedCbox;
        speedCbox >> currentSpeed;
        myThrottle << &Throttle::adjust << cruisingSpeed - currentSpeed;
        myTimer << &Timer::wakeup << 0.1;
        timerCbox >> aSignal;
        become(Self);
        self << &CruiseOn::maintain;
    }
};

class CruiseOff : ACTOR {
        Mbox mySpeedometer, myThrottle, myTimer;
public:
CruiseOff (Mbox aSpeedometer, Mbox aThrottle, Mbox aTimer)
    {
        mySpeedometer = aSpeedometer; myThrottle = aThrottle; myTimer = aTimer;
    }
void on()
 { // turn on the cruise control
        int desiredSpeed;
        self >> desiredSpeed;
        become(new CruiseOn(mySpeedometer, myThrottle, myTimer, desiredSpeed));
        self << &CruiseOn::maintain;
}
void off()
    { // control already off; no action required; ignore this message }
void maintain()
    { // control already off; no action required; ignore this message }
};
```

Figure 5 - A Cruise Controller

Several issues, however, are raised by this example. Messages for on() should not make any difference if cruise control is already on. For example, the driver may press the cruise control button while the cruise control is engaged. The driver should see no visible effect. Similarly, off() and maintain() messages should be ignored if cruise control is off. This problem was handled in the above program by dummy methods, which simply ignore the requests being processed. Two dummy methods were used: on() in CruiseOn and off() in CruiseOff(). However, this solution does not seem satisfactory since it is not feasible to have a dummy method for every unanticipated method. More importantly, some request messages cannot simply be ignored even if they are not served by the current behavior [Kafura and Lee 89]. They may need to be saved for later processing. This example illustrates a more general issue of handling request messages which are not recognized by the current behavior of an actor.

A different kind of difficulty arises when the accelerator or the brake pedal is depressed while the cruise control is on. Messages currently buffered in the mail queue must be discarded since the cruise control mechanism is disengaged. This requires the off() message sent by the accelerator to be served with a higher priority than the other buffered messages. The lack of an urgency notion in the actor model makes it difficult to model even a primitive real-time problem like the cruise control.

Although the classes CruiseOff and CruiseOn were defined as separate behaviors, it is awkward to define them as full fledged classes because they are just parts of a larger behavior, say, CruiseControl. We believe that it would be more natural to combine the two behaviors into a single class. This splitting of a coherent object definition was forced by the semantics of Agha's become operation. Splitting the behavior of an actor in this way is also undesirable from the perspective of class inheritance. One must define subclasses of both CruiseOff and CruiseOn in defining an actor using the cruise control actor. A more fundamental problem resulting from the conflict between the become operation and class inheritance is discussed in [Kafura and Lee 89].

5. Implementation of ACT++ Language Kernel

The classes shown earlier in Figure 2 were implemented on a 3B2 machine which runs Unix System V. The actors are created as light-weight processes which share the same Unix address space. In this section, we present a detailed description of the implementation. We first provide an overview of the class hierarchy of the ACT++

language kernel. The implementation of the kernel language primitives are discussed in detail. The implementation of the pseudo variable self is also described. Finally, we describe the run-time model of the language by following the thread of an actor which sends a message to another actor.

5.1 An Overview of Classes

The class hierarchy used for implementing the current ACT++ kernel was shown in Figure 2. The class ACTOR extends its superclass ActiveObj with the addition of a become() method. The class ActiveObj defines the primitive behavior of all active objects. An instance of ActiveObj contains information needed by the scheduling mechanism. Instance variables for the size and the base address of the run-time stack, frame pointer, and argument pointer for the current thread are present in the ActiveObj class. Methods supported by ActiveObj include operations used to start, suspend and resume a thread. These operations are used by the scheduling mechanism which is embodied in a class called Worker. Although the current implementation supports only a single Worker, the future design will support multiple Workers, each of which runs on a separate node in the network. The class Worker defines variables for scheduling policy information and provides methods needed to perform context switching of threads.

A MemoryManager object is responsible for dynamic storage allocation needed for running actors. Since the current implementation makes use of the C++ object creation mechanism for creating new actors, the function of the MemoryManager object is limited to allocation and deallocation of stack spaces. Future versions of the MemoryManager will incorporate automatic garbage-collection of actors. Garbage-collection of actors is known to be different from other traditional real-time garbage collectors, which are concerned about reclaiming passive data objects. The problem of collecting actors is described in [Nelson 89].

The class Mail is used to create a message from a list of data items. An instance of Mail is a list of MsgToken objects. Each MsgToken object corresponds to a data item supplied in a message sending operation. The overloading of << methods allows a message to be created using a stream-like interface. The reverse operation >> is provided to allow data items to be extracted in a way similar to a stream input operation. The class Mail is not directly visible to a programmer. A programmer uses the << method of an Mbox when sending a message.

The Slink, Dlink, Slist, Dlist, and Queue implements generic data objects for singly and doubly linked nodes, singly and doubly linked lists, and queues. MailBox is implemented as a queue of Mail objects. Two mail boxes, Mbox and Cbox are subclasses of MailBox. An Mbox supports the method <<, which takes a method name and returns a reference to a Mail object which is newly added to the Mbox. Subsequent arguments, if any, of a message sending operation are attached to the returned Mail object using the << operation of the Mail object. The use of << operations in a message sending operation is overloaded for each type of argument. A Cbox supports the overloaded method << which allows a reply to be sent to the Cbox object. A Cbox object also provides the overloaded method >> for reading a reply from the Cbox. The method in() of a Cbox returns TRUE or FALSE depending on the availability of a reply message in the Cbox.

5.2 The New, become, send, receive, and reply Operations

The **New** operation creates a new Mbox, and binds together the Mbox and a behavior supplied in the call. The binding is represented by setting a variable in each object to identify the other object. A pointer to the Mbox is returned.

The **become** operation changes the current behavior of an actor associated with the Mbox by resetting a variable in Mbox to point to the replacement behavior specified as the argument of the call. If a message is available in the mail queue, the replacement behavior is given the message. The become operation also modifies the execution state of the actor. Figure 6 shows the state transitions of a behavior.

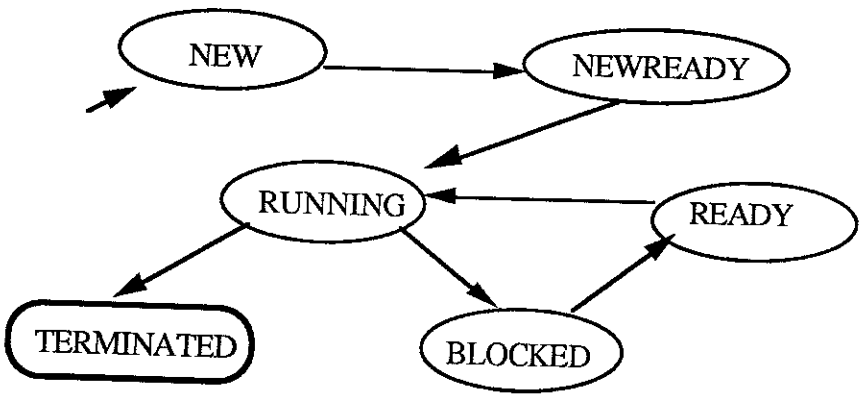

Figure 6 - State transitions of a behavior

A behavior has the NEW state when it is initially created. The behavior changes to the NEWREADY state if the mail queue has a request message. The behavior is given the message and placed in the ready queue of the Worker. If no message is found in the mail queue when a behavior is newly created, the behavior remains in the NEW state, which causes the behavior to wait for the arrival of a new message. When a message arrives at an Mbox, two possibilities exist. If the Mbox has a behavior waiting for a message, the behavior is assigned the message. Then, the behavior is appended to the ready queue, its state being changed to NEWREADY. Otherwise, the message is placed in the mail queue of the Mbox.

A behavior in the NEWREADY state is changed to RUNNING when it is selected for execution by the scheduler. A RUNNING behavior is put in the BLOCKED state if the behavior attempts to **receive** from an empty Cbox. When a reply message arrives in the Cbox later, the behavior is moved to the READY state. The difference between NEWREADY and READY is that a behavior in the former is yet to be allocated a run-time stack while a behavior in the latter has already allocated a stack. The state TERMINATED indicates a behavior execution has been completed. The resources which were used by a behavior in TERMINATED are collected for recycling by the system.

The operation **reply** is implemented as a macro which is expanded to sending a message to a reply destination. The reply destination of a reply message is found in the structure self, which exists in each actor. A variable in self stores the reply destination specified in the request message currently being processed. We now describe the implementation of self.

5.3 self

The pseudo variable self is an instance variable defined in ACTOR. The variable is a reference to an instance of SelfObj which is essentially a pair of Mbox and Mail variables. The << operation applied to self invokes the << operation of the Mbox while the >> operation is directed to the Mail object. When an actor is created, the Mbox variable of self in the actor is initialized to the Mbox of the actor. The Mail variable of self is set to an instance of Mail when a message is bound to a behavior of the actor. The definition of the class SelfObj is shown in Figure 7.

```
class SelfObj : Object {
    Mbox& myMbox;                    // private instance variables
    Mail& myMail;
public:
Mail& operator<<(Method p)           // overload << to receive a
  {                                  // method name
    return (myMbox << p);
  }
  ...                                // overload << to receive args
                                     //  of a method invocation
Mail& operator>>(int x)              // overload >> to receive an
  {                                  //  integer
    return (myMail >> x);
  }
  ...                                // overload >> to receive other
                                     //  types of arguments
};
```

Figure 7 - Implementation of SelfObj

5.4 A Trace of an Execution Thread

In this section, we present a trace of the execution of the factorial actor discussed in Section 4.1 in order to give a more detailed description of the implementation. Figure 8 shows the data structures related to an actor at run-time. An actor is represented by the actor's current behavior. The instance data structure of an actor class is the focal point of a running behavior. The instance of an actor class contains the self variable and a pointer to the run-time stack. The self object identifies a mail queue which represents the actor to which the behavior belongs and a request message being processed by the behavior. The mail queue is also linked to the current behavior by a pointer in its structure. This binding is changed when the current behavior specifies a replacement behavior. Associated with a mail queue are a list of request messages. Cboxes are allocated in the heap memory. The Cbox variables in the stack link a behavior to its Cboxes. Each Cbox is associated with a list of reply messages.

The program shown in Figure 4 is used in the following discussion. The New operation (line 16) creates an Mbox and an instance of ConcFact. These two objects are bound together and the pointer to the Mbox is returned. The pointer is assigned to the variable factorial. The state of the new behavior is initialized to NEW. The next line sends a request

message to factorial. In the current implementation, a method pointer is used as the method name to be invoked. The first << operation results in the creation of a Mail object, to which subsequent << operations are applied. The message consisting of a method pointer, a reply destination, and an integer is sent to the factorial actor. Since the current behavior of factorial is waiting for a message, the request message is bound to the behavior. The behavior is then placed in the ready queue of the Worker.

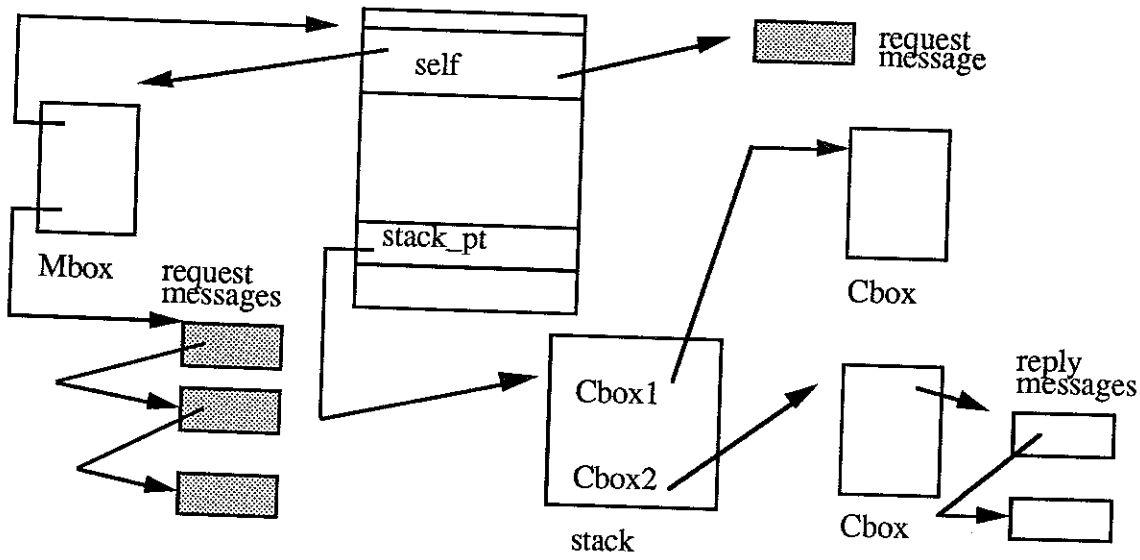

Figure 8 - Run-time data structures for an actor

To allow the main() to be regarded as an actor, an actor representing the thread of main() is created when the first ready behavior is put in the ready queue. From then on, main() is treated as an actor. In the program, main() executes the >> operation on myCbox (line 18). Since the Cbox has not yet received a reply, the main() actor is blocked. Register values are saved in the run-time stack while the frame pointer and the argument pointer are saved in its instance variables defined by ActiveObj so that the actor can be restored later. The state of the main() actor is changed to BLOCKED. An instance variable of myCbox is updated to the address of the main() actor. The Cbox myCbox is then marked as one on which an actor is waiting. The scheduler (Worker) now takes the control. The Worker is implemented as a light-weight process which has its own stack. The Worker state is restored using the frame pointer and the argument pointer saved in its object structure.

The Worker schedules the next ready behavior found in its ready queue. A behavior found in the ready queue may be in either NEWREADY or READY state. Both states imply that a behavior is ready to run. More specifically NEWREADY indicates that the behavior is yet to be allocated a run-time stack, while READY indicates the behavior is resumed with its old stack. In our example, the next ready behavior is in the NEWREADY state. The Worker first allocates a stack space with the assistance of the MemoryManager. The Worker then switches to the behavior by first saving its own state in its stack, and jumping to the entry point of the behavior. The control of the behavior will automatically return to the Worker when the behavior execution finishes. The entry point is located in the method pointer field of the request message being processed by the behavior and is retrieved via a method defined in ACTOR . Hence, line 18 results in blocking of the main() actor and the execution of the compute() method of a ConcFact actor.

The execution of the ConcFact actor causes many more actors to run concurrently. Eventually, the RangeProduct actor created by ConcFact will receive replies from two actors computing sub-ranges (line 41) as discussed in Section 4.1. The RangeProduct actor sends the final result to myCbox of main() (line 42). The arrival of the reply to myCbox moves the main() actor from BLOCKED to READY, with the value in the message being assigned to the variable k. When the main() is resumed by the scheduler, main() continues its execution at line 19. At this time, the value of k is available. Hence, main() prints the correct result of 20! and finishes. Since the return point for main() is not touched, the control returns in a normal way, i.e., to the operating system kernel.

6. Related Work

A concurrent extension of C++ based on Ada tasking is discussed in [Buhr et al. 88]. Since our interest is in actor-based concurrency, it is not included in the following discussion. A brief comparison of ACT++ with other similar languages is presented below and summarized in Figure 9. This comparison is not intended to be a general survey of concurrent object-oriented languages nor is it intended as a general classification of the features of such languages. Rather, its purpose is restricted to placing ACT++ in the spectrum of object-oriented languages that have adopted the actor model as the basis for concurrency. The principal points of comparison are the underlying base (non-concurrent) language, the application domain that conditions the language's design, the architecture of the system used or envisioned to host application programs, and the sharing mechanism, if any, employed by the language.

	Base Language	Sharing Mechanism	Object Interface	Application Domain	Architecture
ACT++	C++	Inheritance	Replaceable	Real-Time	Distributed
Actra	Smalltalk	Inheritance	Fixed	Real-Time	Multiprocessor
Actalk	Smalltalk	Inheritance	Replaceable	Language desgin, classification	Uniprocessor
ABCL/1	new	none	Fixed	Distributed Programming	Distributed
Cantor	new	none	Fixed	High Speed Computation	Multiprocessor
Lamina	Lisp	none	Fixed	Real-Time AI	Multiprocessor
Act-1	Lisp	Delegation	Fixed	Artificial Intelligence	Distributed

Figure 9 - Summary of Actor Languages

The Actra language [Barry and Thomas 87, LaLonde et al. 87] is used by Dave Thomas and others at Carleton University. This language, and the research project from which it came, pioneered the application of object-oriented techniques in embedded, real-time systems. The technology has matured sufficiently to be used in commercial developments. Actra is implemented as an extension of Smalltalk and runs under the Harmony real-time operating system [Gentleman 85]. A class is added to the Smalltalk hierarchy that implements the basic actor abstraction. Actor objects are created as instances of a class which lie in the inheritance subtree rooted by this class. The actor abstractions in ACT++ and Actra are quite different. First, Actra lacks a become operation - the actor's interface is fixed. Second, Actra's message passing semantics are those of the Harmony system (unbuffered, synchronous) while those of Agha's actor model, and ACT++, are buffered and asynchronous. Actra uses only those aspects of the the actor model which support "programming by personfication" - computing elements are described by roles such as worker, administrator, client. ACT++ uses the fuller technical aspects of actors as presented in [Agha 86]. One final difference between Actra and ACT++ is the intended target architecture. Actra uses a shared memory multiprocessor while ACT++ is intended for use in a non-shared memory distributed environment.

A related but independent research Actalk [Briot 89] is a small language kernel nicely built into Smalltalk-80. Actalk is intended for classifying and simulating actor languages using a single framework. The approach used for creating "activeness" in Actalk is close to that of ACT++. A class hierarchy consisting of two classes, Actor and ActorBehavior, is used to extend passive Smalltalk objects to active objects communicating via asynchronous message passing. Simulation of ABCL/1 and Agha's actor language kernel is demonstrated by extending the class hierarchy. The success of Actalk is partly due to a skillful exploitation of the Smalltalk-80 languages features such as late binding, message passing syntax, and concurrency. These three features were more difficult to implement starting with C++ as the base language.

The actor model is applied to research in distributed computing by the ABCL/1 language [Yonezawa et al. 87, Tomlinson and Scheevel 88]. ABCL/1 has been used in a wide variety of distributed problems and a distributed implementation is available. Like other actor-based languages, ABCL/1 modifies the actor semantics to conform with the requirements of the application domain. The significant modifications are:

- messages between two actors are ordered
- "express" messages allow preemption
- both remote procedure call and future style message passing are provided

ABCL/1 does not employ the notion of a replacement behavior and there is no support for shared abstractions.

Continuing research into message-passing multiprocessor hardware systems at Caltech motivated the design of Cantor [Athas and Boden 88]. The fine granularity afforded by a highly-to-massively parallel architecture and the message oriented method of processor interconnection are well matched with the assumptions of the actor model. In contrast to ACT++, Cantor does not have a become operation and does not provide for shared abstractions. Experience with Cantor has highlighted the difficulty of the garbage collection problem.

The Lamina language [Delagi and Saraiya 88] supports research into the application of artificial intelligence techniques to real-time problems. The Lamina object model is similar to actors in two respects: message delivery is nondeterministic; and the object's computation is triggered by the receipt of a message. Futures and streams are used to

structure message passing among objects. Lamina, however, departs radically from the actor model by incorporating a form of shared memory. Lamina also incorporates both an explicit continuation (passing a mail queue address) and an implicit continuation (hidden closures encapsulated in the originating object). One example of this research is ELINT - a system to recognize and discriminate among passively acquired electronic signal sources. It was found that explicit continuations predominated the ELINT application. Further research is needed to provide additional evidence on this comparison. ACT++ provides the same form of explicit continuations as Lamina but uses the combination of Cboxes and replacement behavior to achieve the same effect as Lamina's implicit continuations. Lamina has no counterpart of the actor become operation. No sharing mechanism is present in Lamina.

Act-1 [Lieberman 87] and its successors [Theriault 83, Agha and Hewitt 87] are Lisp-based languages developed at MIT. In these languages the primitive mechanisms of Lisp are represented as actors. Sharing of abstraction is supported through a delegation mechanism. The underlying use of actors and delegation provides a means, not present in Lisp, of creating entities which can evolve in an upwardly compatible fashion. Highly parallel and distributed artificial intelligence applications are also more easily supported by the use of the actor model. Concurrency in Act-1 is generated by the use of futures and restricted by the use of serializers and guardians. The use of serializers and guardians achieves the effect of the mail queue mechanism of Agha's actor model. However, Act-1 does not provide the become operation. The reusability issue is not addressed by Act-1.

7. Future Research

In ACT++, the become operation is mainly used for synchronizing on a shared actor object. However, in Agha's actor model, chaos in program structure may result since an actor is free to become a behavior with little conceptual relation to the current behavior. We view such an unbounded become operation as a harmful construct like the indiscriminate use of goto statements. While the current implementation of ACT++ places no restriction on specifying a replacement behavior, we are investigating ways of controlling such unrestricted become operations.

The use of a method pointer as a method id requires all actors to execute in a single address space. The use of a pointer must be replaced by a logical name before the language is extended to a distributed environment. Many issues crucial for real-time systems

programming have been left out. Future designs will consider the issues of distribution and real-time programming support.

Our decision to remain within the framework of a C++ inevitably caused certain inelegances in ACT++ syntax. Although the syntax of the current design is not as elegant as one designed without such a constraint, we believe the current design was useful enough to serve our goal of experimenting with the concurrency abstraction of actors. The design of ACT++ will continue to evolve as we gain more experience in programming real-time problems in the actor paradigm.

8. Conclusion

In this paper, we described how the concurrency abstraction of the actor model could be incorporated into C++. The design and implementation of ACT++ based on a class hierarchy were discussed. The classes in the hierarchy were described in some detail. Two example programs written in ACT++ were presented.

An argument was offered that the primitive actor model needs to be adapted for efficient implementation on conventional architectures. Specifically, we argued that the actor model should allow more coarse-grained actors to be created. Another difficulty is that the messages in a mail queue need to be discriminated. The notion of urgency which is fundamental in real-time programming needs to be addressed. The problem of a coherent object being broken into small pieces was illustrated by an example. We also argued that request messages and reply messages should be handled differently. The Cbox mechanism was proposed for dealing with reply messages. The flexibility of the Cbox mechanism was also demonstrated.

We consider the fundamental characteristics of the actor model to be active objects, asynchronous message passing, dynamic creation of new actors, reconfiguration of actor topology, and the become operation. The formal actor model defined by Agha should be recast in the light of the intended applications. We believe that a language may well be said to be based on the actor model if the language supports the above five characteristics.

Acknowledgement

The authors thank Greg Lavender for his critical reading of a draft of this paper.

9. References

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

[Agha and Hewitt 87] Gul Agha and Carl Hewitt, Concurrent Programming Using Actors, In Object-Oriented Concurrent Programming, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 37-53.

[Athas and Boden 88] W. C. Athas and N. J. Boden, Cantor: An Actor Programming System for Scientific Computing, Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, vol. 24, No 4, April 1989.

[Attardi 87] Giuseppe Attardi, Concurrent Strategy Execution in Omega, In Object-Oriented Concurrent Programming, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 259-276.

[Barry et al. 87] Brian M. Barry et al., Using Objects to Design and Build Radar ESM Systems, OOPSLA '87 Conference Proceedings, SIGPLAN, ACM, 1987.

[Briot 89] Jean-Pierre Briot, Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment, To appear in the Proceedings of European Conference on Object-Oriented Programming (ECOOP' 89), July 1989.

[Buhr et al. 88] P. A. Buhr, G. J. Ditchfield, and C. R. Zarnke, Concurrency in C++, Research Report CS-88-30, University of Waterloo, Waterloo, Ontario, July 1988.

[Caromel 88] Denis Caromel, A Generalized Model for Concurrent and Distributed Object-Oriented Programming, Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, vol. 24, No 4, April 1989.

[Clinger 81] W. D. Clinger, Foundations of Actor Semantics, Technical Report 633, MIT Artificial Intelligence Laboratory, 1981.

[Delagi and Saraiya 88] B. A. Delagi, N. P. Saraiya, ELINT in LAMINA: Application of a Concurrent Object Oriented Language, Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, vol. 24, No 4, April 1989.

[Gehani 83] Narain Gehani, *Ada Concurrent Programming*, Prentice-Hall, 1983.

[Gentlemen 85] W. M. Gentlemen, Using the Harmony Operating System, National Research Council of Canada Report No. 24685 National Research Council of Canada, Ottawa, Canada, May 1985.

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

[Hewitt 77] Carl Hewitt, Viewing Control Structures as Patterns of Passing Messages, Journal of Artificial Intelligence, 323-364, June, 1977.

[Hewitt and Baker 77] Carl Hewitt and H. Baker, Laws for Communicating Parallel Processes, IFIP Conference Proceedings, August 1977.

[Hewitt and Atkinson 79] C.E. Hewitt and R. Atkinson, Specification and Proof Techniques for Serializers, IEEE Transactions on Software Engineering, Vol. SE-5, No 1, January 1979.

[Hewitt and deJong 83] Carl Hewitt and Peter de Jong, Analyzing the Roles of Descriptions and Actions in Open Systems, Proceedings of the National Conference on Artificial Intelligence, AAAI, August 1983.

[Nelson 89] Jeffrey Nelson, Automatic, Incremental, On-the-fly Garbage Collection of Actors, MS Thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, February 1989.

[Kafura 88] Dennis Kafura, Concurrent Object-Oriented Real-Time Systems Research, Technical Report, TR 88-47, Department of Computer Science, Virginia Polytechnic Institute and State University, 1988.

[Kafura 89] Dennis Kafura, Concurrent Object-Oriented Real-Time Systems Research, SIGPLAN Notices, ACM, February 1989.

[Kafura and Lee 89] Dennis Kafura and Keung Hae Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, To appear in the Proceedings of European Conference on Object-Oriented Programming (ECOOP' 89), 1989. Will also appear in Computer Journal, August 1989.

[Lalonde et al. 86] W. R. Lalonde et al., An Exemplar Based Smalltalk, OOPSLA '86 Conference Proceedings, ACM, 1986.

[Lieberman 87] Henry Lieberman, Concurrent Object-Oriented Programming in Act 1, In Object-Oriented Concurrent Programming, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 9-36.

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

[Theriault 83] D. G. Theriault, Issues in the Design and Implementation of Act2, Technical Report 728, MIT Artificial Intelligence Laboratory, 1983.

[Tomlinson and Scheevel 88] Chris Tomlinson, M. Scheevel, Concurrent Object-Oriented Programming, Microelectronics and Computer Technology Corporation, MCC Technical Report No. ACA-ST-078-88, March 1988.

[Yokote and Tokoro 86] Yasuhiko Yokote and Mario Tokoro, Concurrent Programming in Concurrent Smalltalk, In Object-Oriented Concurrent Programming, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987.

[Yonezawa et al. 87] Akinori Yonezawa, Etsuya Shibayama, et al., Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, In Object-Oriented Concurrent Programming, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 55-89.