ACT++ 2.0 : A Class Library for Concurrent Programming in C++ Using Actors[1]

By

Dennis Kafura, Manibrata Mukherji, and Greg Lavender

Virginia Tech

kafura@cs.vt.edu

ACT++ 2.0 is the most recent version of a class library for concurrent programming in C++. The underlying model of concurrent computation is the Actor model. Programs in ACT++ consist of a collection of active objects called actors. Actors execute concurrently and cooperate by sending request and reply messages. An agent, termed the behavior of an actor, is responsible for processing a single request message and for specifying a replacement behavior which processes the next available request message. One of the salient features of ACT++ is its ability to handle the Inheritance Anomaly - the interference between the inheritance mechanism of object-oriented languages and the specification of synchronization constraints in the methods of a class - using the notion of behavior sets. Another feature of ACT++ is its realization of I/O as an actor operation. A special type of actor, called an interface actor, provides a high level interface for a file. Interface actors are sent request messages whenever I/O is necessary and can also transparently perform asynchronous I/O. ACT++ has been implemented on the Sequent Symmetry multiprocessor using the PRESTO threads package.

1.0 Introduction

Structuring applications as a collection of objects that communicate and cooperate with each other to solve a single problem is argued to be a natural way of modeling the real world and of capturing the concurrency of operations that exist among the entities of the real world. The objects which model the real world entities can be thought of as autonomous agents which compute independently of each other and communicate with each other via messages. To exploit the concurrent behavior of the objects comprising an application it is imperative to use a programming language that enables one to specify and control concurrent objects.

While C++ [Stroustrup 86] is a widely used language, computation in a C++ program is carried out by a single thread of control and the sender of a message blocks until the requested operation is completed. Given the advances made in the realm of computers having multiple processors and the development of distributed programming paradigms, the latter limitation of C++ not only inhibits the user from exploiting the natural concurrency in the problem domain but also fails to exploit the physical parallelism of computations that can be achieved in order to enhance efficiency.

[1] This work was supported in part by the National Science Foundation under grant CCR-9104013.

Our goal in designing ACT++ was to provide the ability to specify and control concurrent objects in C++. Instead of inventing a new model of concurrent computation we decided to implement some features of the Actor model of concurrent computation [Agha 86, Agha and Hewitt 87]. [Kafura and Lee 90] describes the design issues and the implementation details of an earlier version of ACT++. This paper describes ACT++ 2.0, the latest version which enhances the earlier one in three respects. First, we have used the PRESTO [Bershad et al 88, Bershad 90] *threads* package as the underlying system to control concurrent objects. PRESTO was used because it is written in C++ and provides a high-level abstraction for concurrent programming using "light-weight" processes called *threads*. Second, we have implemented a concurrency control mechanism called *behavior sets* [Lavender and Kafura 90] which can be used to resolve the conflict between the inheritance mechanism of object-oriented languages and the specification of synchronization constraints. Third, we have introduced special features using which simultaneous input/output requests from multiple, simultaneously executing threads of control can be handled correctly and efficiently.

In the remainder of the paper, first, we introduce the salient features of the model of computation implemented in ACT++ 2.0. Then we introduce the different components that are available to a user of ACT++ and show the intended use of each component by means of examples. We conclude the paper by stating the current status of our work and directions of future research.

2.0 The ACT++ Model of Computation

In the following we summarize the salient features of the model of computation implemented in ACT++.

- An ACT++ program consists of a collection of actors which execute concurrently using independent threads of control.

- Actors communicate asynchronously via messages called *request* messages. Each actor has the ability to buffer messages in a queue called its *mail queue*. Actors can also participate in synchronous communications with objects that are not actors.

- Request messages contain requests for the execution of program segments in an actor called *behavior scripts*. Behavior scripts are implemented as the methods of a special type of object called the behavior object (also referred to as the behavior). Each behavior is responsible for processing exactly one request message.

- The behavior object that will process an actor's next request message is called its *current behavior*. At any point in time only a single behavior object is designated as the current behavior of an actor. *The execution of a specific method in the behavior object of an actor is regarded as the processing of a message by that actor.*

- The processing of a request message by a behavior of an actor could lead to the occurrence of one or all of the following actions:
  - send asynchronous request messages to itself or other actors,
  - synchronously invoke methods in itself or in other non-actor objects,
  - create more actor or non-actor objects, and
  - specify the next behavior of the actor (called a *replacement behavior*) that will process the next request message.

- Actors may also exchange *reply* messages. As a result of a request message to it, an actor may send a reply message to the requestor with the result of the computation. A special type of object called a Cbox is used to buffer reply messages. Cboxes are defined with blocking semantics so that an actor blocks when it tries to extract a reply message from an empty Cbox.

- An actor in ACT++ is not restricted to process request messages in the strict order of their arrival. Instead, a replacement behavior can consult the values of its instance variables to decide which request message to process next.

3.0 Concurrent Object-Oriented Programming in ACT++

The universe of objects in an application program is partitioned into two sets - a set of *active objects* and a set of *passive objects*. The active objects, represented as actors in ACT++, can buffer messages and process messages asynchronously by creating an autonomous thread of control to execute the method requested in each message. Passive objects do not buffer messages and only engage in synchronous invocations using the thread of control of the requestor. The object definition facility and method invocation mechanisms provided by C++ specify the passive objects and the computations involving them. The class library provided by the ACT++ system is used to specify the active objects and the asynchronous message processing associated with them.

In the following sections we will introduce the various types of objects available to a user of ACT++. The specific types of objects that will be introduced are:

- Actors
- Messages
- Behaviors
- Behavior Sets
- Cboxes
- Objects used for asynchronous I/O
  - Interface actors
  - Rboxes and Wboxes

The purpose and the intended usage of each type of object is explained, the C++ class interface implementing each type is introduced, and some examples are given.

3.1 Actors

All actor objects in ACT++ are instantiations of the predefined `Actor` class. The methods of this class that are available to the user are the constructor for the class and the `print` method. The constructor has the following signature:

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

The Behavior pointer `init_beh` specifies the initial behavior object that is responsible for processing the first message to the actor. Whenever a message is accepted for processing by an actor, a *thread* (the italicized word *thread* is used to refer to a `Thread` object which is the unit of scheduling in PRESTO) is created to execute the method requested in the message. The integer variable `th_num` signifies the number of *threads* that can be simultaneously active in an actor. The default number of *threads* is one. The user can specify more that one *thread* to enable the possibility of concurrent message processing by allowing two or more behaviors of the actor to process request messages simultaneously. The character pointer `name` associates a name with an actor that can be printed when debugging an ACT++ program.

Every predefined object in ACT++ has a `print` method that can be used to print the values of the private data members of the class. This method is useful only during debugging an ACT++ program. The `print` method has the following signature:

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

This simple method will not be discussed further.

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

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

3.2 Messages

Messages play two inter-related roles in the Actor model. First, messages are the means of communication between actors. The computation performed by a system of actors is driven by the passing of messages. Second, the execution of an actor's current behavior is initiated by the availability of a message requesting the execution of some operation in that behavior's interface. Messages are buffered in a message queue in an actor and, in the original Actor model, are processed in a First-In First-Out (FIFO) order. The default FIFO message processing order has been extended in ACT++ to provide more control over the selection of messages. That is described in section 3.4.

All messages exchanged between actors in an ACT++ program are objects which are instantiations of the predefined class `Message`. There are two constructors in the `Message` class. Their signatures are:

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

A message object consists of the physical address of the method that is being requested for execution along with the arguments. The argument, of type `PFany` (a predefined type in PRESTO), is a pointer to a method obtained by specifying *&class_name::method_name*. The ellipsis after `PFany` signifies that an unknown number of arguments can follow. The same discussion applies to the second constructor except that its first argument is a pointer to an actor object. If constructed in the second way, the actor is used as the default destination of the message. Note that the actual sending does not take place when the message is constructed and at the time of sending, the message need not be sent to the default actor.

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

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

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

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

A variation of the second overloading is used by an actor to send a message to itself. Instead of using an actor pointer as the argument, the predefined macro `SELF` is used as the argument. `SELF` expands into a pointer to the actor itself. Note that a message object *must never* be sent to more than one actor.

There is **no type checking** of the arguments of the methods that are invoked through a message. This implies that the user bears full responsibility for ensuring type consistency between a message and the signature of the method that will be executed as a result of its delivery. The lack of type checking stems from the fact that the events of type checking the arguments and generating code for the actual execution of a method are inextricably interwoven in the C++ compiler. On seeing a method invocation at compile time, the C++ compiler does the type checking of the arguments, sets up the callee's activation record, and generates code to initiate the callee. In order to get control over the initiation of the callee, the C++ compiler must be bypassed and its actions simulated, as far as possible, at runtime. Since C++ is a statically typed language, no type information is available at runtime. As a result, in PRESTO, when a *thread* is started asynchronously inside the method of an object the arguments cannot be checked for type consistency. Since asynchronous message passing in ACT++ is built on the asynchronous *thread* execution feature of PRESTO, parameter passing restrictions present in PRESTO also apply to ACT++. The following is an excerpt from "The PRESTO User's Manual" [Bershad 90, page 8] indicating the restrictions involved.

> "Default and reference parameters will not work properly although virtual and inline functions work fine. Overloaded functions will also not work since the compiler cannot discern the types of the arguments to the function (or its return value) at compile time.
> Disallowing reference parameters implies that all parameters (including pointers) are passed by value. Consequently the programmer should be careful that pointers reference objects residing in the global address space (static or created by new) and do not point to objects on the stack of the thread that is starting the new thread. Failure to abide by this may result in the passed pointer referencing garbage (consider the stack's behavior on an asynchronous invocation)."

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

3.3 Behaviors

Messages sent to actors in ACT++ are actually requests to execute one of the methods in a behavior object of the actor. All behavior objects in ACT++ are instantiations of user defined subclasses of the predefined `Behavior` class. A major activity involved in writing an ACT++ program is defining all the behavior classes needed for the different actors in the system. Each of those classes has its own set of data members and a set of operations. Declaring those classes as subclasses of the `Behavior` class enables the inheritance of special operations. One of those special operations is used to specify replacement behaviors from the methods of the behavior classes.

3.3.1 Replacement Behaviors

Specification of replacement behaviors is the means by which actors retain their ability to process messages. That is done in ACT++ by the `become` operation defined in the `Behavior` class. The current behavior of an actor is responsible for specifying the replacement behavior that selects the next message to be processed. The replacement behavior object is usually an instantiation of the *same class as that of the current behavior*. Technically, assigning objects of different behavior classes as replacement behaviors is not impossible. But, conceptually, an actor serves a specific purpose which is usually realized by the operations defined in a single behavior class.

There are two variations of the `become` operation. The first one is used to designate a new object of the same class as that of the current behavior as the replacement behavior. Its signature is:

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

A new behavior object is created and a pointer to it is sent as an argument to the `become` operation.

The second variation marks the current behavior object as the replacement behavior instead of creating an entirely new object. This is specified as

```
become(THISBEH)
```

where THISBEH is a predefined macro which expands to a pointer to the current behavior object.

In ACT++, no low-level mechanism, such as locks, is available to explicitly define critical sections. Hence, a behavior object must always process a single message. Otherwise, interference between independent threads of control could corrupt the *state variables* - the instance variables of the current behavior of the actor. Therefore, first, one must always create a new behavior object as the replacement behavior which prevents the simultaneous execution of independent threads of control in the same object. Second, since the instance variables of the replacement behavior are initialized

using the values of those in the current behavior, the `become` operation must be specified only after the current behavior has finished updating the state variables. It is only when the `become` is specified as the last statement in a method of the current behavior can one use the `THISBEH` macro to specify the current behavior object as the replacement; this is more efficient in terms of space and time.

3.4 Selective Message Processing Using Behavior Sets

The notion of **behavior sets** [Lavender and Kafura 90] has been incorporated in ACT++ to provide the ability of selective message processing by the behaviors of an actor. Until now the same set of operations was available for execution in each behavior irrespective of its state. This property of behaviors is not conducive to modeling real-life objects that dynamically change their interface depending on their internal state. In order to model such dynamic objects using actors, one has to incorporate synchronizations involving the state variables in the methods of the behavior in order to realize selective message processing. Such synchronizations must not become an obstacle to the *inheritence mechanism* of object-oriented languages.

To be explicit, let us consider the bounded-buffer problem in which a buffer of a finite size is read by and written to by multiple independent actors. We consider the buffer to be an actor. Let `Bounded_Buffer` be the class that defines the behavior of the buffer actor. The class will have two methods in it; the `in` method adds one item to the buffer and the `out` method extracts one item from the buffer. When the buffer is empty the `out` method is inapplicable and its execution due to a request message can be delayed by an explicit test in the method itself. While this is a plausible solution it introduces a problem called the **Inheritence Anomaly** [Matsuoka et al 90, America 87, Briot and Yonezawa 90, Kafura and Lee 89]. Due to the anomaly, when the `Bounded_Buffer` class is subclassed, problems arise due to the state dependent synchronization code in the body of `out` - one has to provide a complete redefinition of the `out` method in the subclass. This renders the `out` method in the superclass useless for any derived classes.

The solution proposed to this problem in [Lavender and Kafura 90] is to separate the state dependent synchronization mechanism from the methods of a behavior class and consider it to be a separate entity subject to explicit manipulation. The specific notions introduced to implement this selective method invocation scheme are *behavior sets*, a *next-behavior-set function*, and *object-state functions*.

A behavior set is an object used by a behavior to process messages selectively depending on the behavior's state. A behavior set contains the method identifiers of a subset of the methods of the class of which the behavior object is an instance. The behavior set represents the interface of the behavior object during the selection of a request message. *A behavior will execute a message only if the method in the request message is an element of the behavior set*. Otherwise the behavior will wait for the arrival of an acceptable message. The behavior set is established by the next-behavior-set function at the time a replacement behavior is specified (and also when the *initial* behavior is created). The next-behavior-

set function in turn depends on the object-state functions which consult the data members of the behavior to determine the current state of the actor. The relationships between these elements is shown in *Figure 1*. The conceptual "state" of an actor is represented by the state variables of the current behavior which are consulted by the object-state functions. The next-behavior-set function invokes the object-state functions in order to compute the behavior set of the replacement behavior.

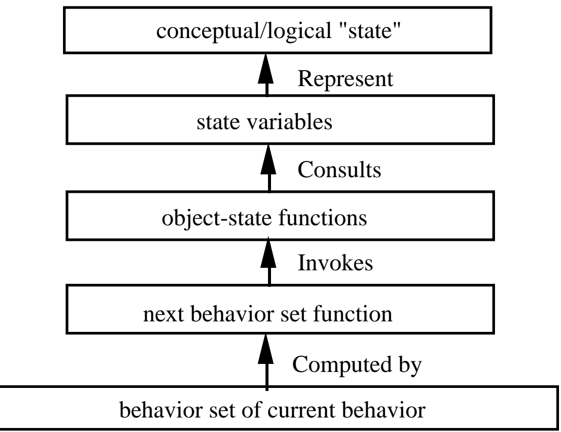

Figure 1. Data and function dependencies involved in the determination of a behavior set.

To illustrate the usefulness of the behavior set idea let us consider the `Bounded_Buffer` behavior once again. As shown in *Figure 2,* there are three possible states of the bounded buffer - *empty*, *partial* and *full*. Corresponding to these three states we will define three *object-state functions* - `empty()`, `partial()`, and `full()`. These three functions will have access to the data member of the `Bounded_Buffer` class, `count`, which is used to keep a count of the number of elements in the buffer. There will be three behavior sets: `empty_set` which has the identifier for the `in` method, `partial_set` which has identifiers for both the `in` and `out` methods, and `full_set` which has the identifier for the `out` method only. We also define a next-behavior-set function which uses the object-state functions to determine which one of these behavior sets to establish as the behavior set of the replacement behavior.

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

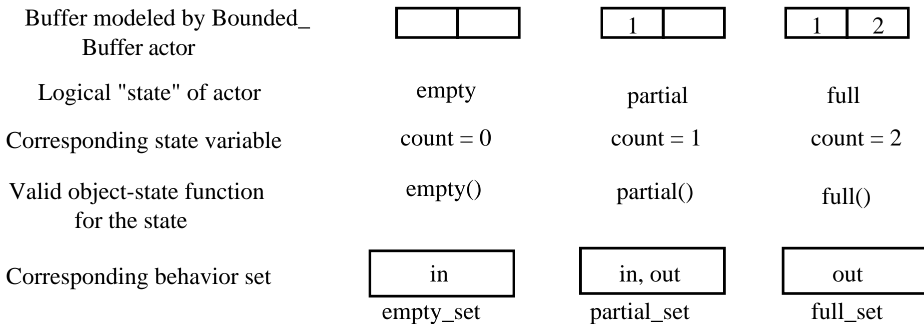

Figure 2: For a bounded-buffer of size two the different configurations of the buffer, the state of the actor, the object-state function valid in each state, and the corresponding behavior sets are shown.

3.4.1 Specifying Behavior Sets in ACT++

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

`(PFany)&Class_name::Method_Name`.

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

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

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

The first constructor is a default constructor that is used to construct an empty behavior set object. The second constructor is used to construct a behavior set whose elements are addresses of the methods supplied as arguments to the constructor. This constructor can take an arbitrary number of arguments (indicated by the ellipsis) all of which must be of type `PFany`.

The three major operations defined on behavior sets are defined as friend functions of the `Behavior_Set` class. There are two overloadings of the '=' operator and one overloading of the '+' operator. The signatures of these overloadings are:

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

The first overloading of '=' is used when the declaration of a null behavior set and its initialization are separated as in the following code:

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

The overloading of '+' is used to form the union of two behavior sets non-destructively. The '+' operator is always used with the second overloading of the '=' operator. The new behavior set queue returned by '+' matches the second argument of '='. The '=' operator deletes the existing behavior set queue of the behavior set that appears as its first argument and reassigns it to point to the new behavior set queue.

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

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

In the constructor of the `Behavior` class `curr_bset` is set to zero and, as shown above, the `NextBehavior_Set` method is null. The `curr_bset` member is used to store the current behavior set of the behavior. The `NextBehavior_Set` function is used to determine the behavior set of the initial behavior object and that for all subsequent replacement behaviors. It is the responsibility of the user to provide a definition of the `NextBehavior_Set` method in the behavior subclasses and ensure that the `curr_bset` member is maintained properly. Note that the determination of the next behavior set is based on the values of the state variables of the behavior. Accessing the state variables of the object is best done by using object-state functions.

A complete class definition of the `Bounded_Buffer` behavior that uses behavior sets and exemplifies all the aspects discussed above can be found in [Lavender and Kafura 90].

3.5 Cboxes

Another feature necessary to program with actors in ACT++ is a **Cbox**. If an actor expects some information to be returned as a result of its request message(s) then there must be a way to specify how the returned information is delivered to the requestor. In Agha's model the concept of **insensitive actors** is introduced for this purpose. Insensitive actors lead to a very costly implementation because the message queue of the actor is used for both receiving request messages and also for receiving a reply message. To overcome the overhead of handling reply messages, ACT++ separates the

handling of request and reply messages. The repository of reply messages is called a Cbox and, unlike the unique message queue of an actor, there can be many Cboxes declared in a behavior.

We will discuss two implementation strategies for Cboxes: method overloading and templates. We first discuss the method-overloading version and then show how the interface would change if templates were used.

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

```
Cbox(Cbox_Type).
```

The argument to the constructor specifies one of the three types of Cboxes available in ACT++. The three types are denoted by the macros `FIRST`, `LAST`, and `QUEUE`. The Cbox of type `FIRST` stores only the very first reply message and, until that message is extracted by the requestor, ignores any subsequent reply messages. The Cbox of type `LAST` stores only the latest reply message by overwriting any existing reply message. The Cbox of type `QUEUE` enqueues all the reply messages. The different types of Cboxes not only serve as vehicles for information exchange but can also be used to implement different types of data-based synchronization schemes among actors. For example, Cboxes of the FIRST and LAST types can be used to implement OR synchronization and the QUEUE type can be used to implement AND synchronization. The LAST type Cbox can also be used in situations where the stream of reply messages contain more timely status information or more accurate approximation of a desired result. In these cases only the most current reply message is of importance to the receiving actor. Note that receiving a reply message from a Cbox is a destructive process - the reply message is extracted from the Cbox and the Cbox is marked empty; a QUEUE type Cbox is marked empty only if it has no reply messages left in it.

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

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

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

The sender of the reply message uses one of the following six overloadings of the `send` method and to send a pointer to a structure, the `SENDSTR` macro.

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

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

`QUEUE` type Cboxes can be used to receive multiple items either from the same or different replying actors. However, a `QUEUE` Cboxe must be used to store *multiple reply messages of only one type*. If the Cbox is not used to store homogeneous items then the `receive` operation could result in an invalid assignment.

Since receiving a reply message removes the message from the Cbox and marks it as empty, the FIRST and LAST type of Cboxes can be reused immediately after a `receive` operation. If it is ever desired to ignore all the reply messages accumulated in a QUEUE type Cbox before receiving the next reply message then the `flush` method must be used whose signature is as follows:

```
void flush()
```

An example showing how a receiver and a sender of reply messages would use the different types of Cboxes to exchange information is shown in *Figure 3.*

In the receiving behavior:

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

cb3->receive(ret_char);
cb4->receive(ret_double);
cb5->receive(ret_cb);
cb6->receive(ret_struct);
cb7->receive(ret_act);
```

In the sending behavior:

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

Figure 3: An example showing the use of Cboxes in a pair of behaviors exchanging reply messages.

In a template implementation, Cboxes can be used as follows:

```
Cbox<int>* cb1 = new Cbox<int>(QUEUE);
Cbox<char>* cb2 = new Cbox<char>(LAST);
Cbox<Actor*>* cb3 = new Cbox<Actor*>(FIRST);
int ret_int1;
...
cb1->receive(ret_int1);
...
cb2->send('a');
```

The type of data that a Cbox will hold is part of the declaration. As a result, the use of a Cbox in the behavior in which it is declared is type safe, for example, the compiler would have complained if `ret_int1` was not an integer variable in the above code segment. Thus, templates yield a more type safe implementation of Cboxes. However, because pointers to Cboxes are exchanged which cannot be checked for type consistency, the user must be careful to use a given Cbox consistently in both the receiving and the sending behaviors.

Another advantage of using templates is the homogeneous way in which reply messages carrying structures can be specified. In the overloaded method implementation a reply message containing a structure is received using the `receive` method but is sent using the `SENDSTR` macro. Moreover, since the name of the structure is provided by the user, a structure pointer is cast as a character pointer before sending and received as a `void*` in `receive`. With templates, the name of the structure can be used directly in the class definition which leads to a better implementation and simplifies the usage as shown below.

In the receiving behavior:

```
struct xyz { int a; float b; };
struct xyz s2;
Cbox<xyz>* cb4 = new Cbox<xyz>(FIRST);
cb4->receive(s2);
```

In the sending behavior:

```
struct pqr {int a; float b;};
struct pqr s2;
s2.a = 4;
s2.b = 3.2;
c4->send(s2);
```

Therefore, templates provide a much more elegant implementation of Cboxes and it will be incorporated into ACT++ when PRESTO is upgraded to run with an AT&T C++ compiler supporting templates. The above implementation scheme has been tested separately using the g++ compiler.

3.6 Input and Output In ACT++

Input and output in a concurrent environment (henceforth concurrent I/O) is complicated by the presence of multiple, simultaneously executing threads of control which can execute I/O requests independently to and from the same file. To ensure consistency of data, simultaneous I/O requests must be executed in critical sections. The absence of any low-level synchronization mechanisms such as locks in ACT++ does not allow the construction of such critical sections for doing concurrent I/O. Even if such mechanisms were present in ACT++ they would have interfered with the inheritance mechanism of C++ causing the Inheritance Anomaly. Another solution to the concurrent I/O problem is to use a central I/O server to handle I/O to all files. The complexity of such a central server increases rapidly with the number of files and the server could easily become a bottleneck in the presence of a large number of independent threads of control.

A second problem of doing I/O in a UNIX system is the blocking effects of low-level I/O system calls. In a concurrent application which does not use a central I/O server, every process might block on I/O calls thereby defeating the purpose of using multiple processes to run the application. The latter event might also cause a real-time application to miss important events in the external world. Though UNIX provides asynchronous I/O facilities for terminal special files only, the user has to explicitly specify a signal handler, set up the file descriptor for asynchronous I/O, and mark the process

receiving the asynchronous signal in order to use the facility. A language feature hiding all these aspects would be easier to use.

The implementation of the concurrency management issues in ACT++ using the PRESTO threads package introduces a third problem for concurrent I/O. Since *threads* in PRESTO can migrate from one process context to another in its lifetime (in order to balance the load), the meaning of a file descriptor obtained as a result of opening a file on one process is lost when the *thread* migrates to another process and tries to do I/O using the same descriptor.

All the problems mentioned above has been resolved in the current implementation of ACT++. For a detailed discussion of the issues, the ways they have been tackled, and for a detailed account of the implementation of the I/O facility in ACT++ refer to [Mukherji 92]. In the following we will discuss the features available in ACT++ for doing concurrent I/O.

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

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

An IA is a special type of actor that is responsible for managing I/O to a single file using a unique file descriptor. It serializes all I/O requests to a particular file. There must be only one IA managing I/O to a given file. An IA encapsulates all low level details for performing I/O and relieves the user from managing file descriptors explicitly. IAs are capable of doing both synchronous and asynchronous I/O. If a request to a terminal special file cannot be satisfied immediately then the IA prepares the file descriptor for asynchronous I/O and interacts with the relevant signal handler when the I/O is ready. The introduction of IAs has enabled the modeling of I/O in terms of actor operations thereby presenting a uniform environment to the users of ACT++. Currently only null-terminated string I/O can be done using IAs.

To do I/O on a particular file (either a special file corresponding to a terminal or an ordinary character file) an instance of an IA is created using the following constructor:

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

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

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

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

Rboxes and Wboxes are character buffers that are used to read and write data respectively. In their absence, the transfer of data between an IA and an actor requesting I/O would have to be done using data members of the behavior of the requesting actor. That would violate the encapsulation of behavior objects. Hence Rboxes and Wboxes are used as shared resources for information transfer between client actors and IAs.

All Rboxes in ACT++ are instantiations of the predefined class `Rbox`. There are two constructors in the `Rbox` class:

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

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

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

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

The first overloading is used to delete the existing buffer in the Rbox and create a new one having the same size as the previous buffer. The second overloading does the same and also resizes the buffer in the Rbox by using the integer argument as the new size. These two methods provide some flexibility in reusing existing Rboxes without having to create new ones.

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

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

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

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

```
int size().
```

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

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

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

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

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

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

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

using the first constructor. The second overloading provides the ability to refill a Wbox from a Rbox - the size and content are extracted from the Rbox.

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

```
void refresh().
```

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

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

```
int size().
```

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

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

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

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

```
void wait().
```

Before a `Read` opeartion is requested, a behavior creates a Rbox object. When the data is required a `wait` operation is invoked on the Rbox. The operation blocks if the Rbox has not yet received the data. Otherwise the call returns immediately. The `wait` on Wboxes is used to determine whether the `Write` operation has completed. If the `Write` has

completed the call returns immediately. Otherwise the operation blocks. Automatic awakening of the blocked behavior is ensured by `wait` for both types of boxes.

When a message for an IA is to be created, the `FILEACT` macro name must be used to obtain the address of the `Read/Write` method of the `File_Beh` class. For example:

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

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

The final I/O operation is the `Close` method defined in the `File_Beh` class which has the following signature:

```
void Close().
```

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

We give several examples below showing how to use the different components for doing I/O in ACT++.

```
char* fname = "/dev/ttyp9";
IActor* my_iact = new IACTOR(fname, FILEBEH);
      ...
Rbox* rb_empty = new Rbox();              //Create an empty Rbox
Rbox* rb_n_empty = new Rbox(72); //Create an empty Rbox with a buffer of size 72
char* mess1 = "Please enter a line of text";
char* mess2 = "Have you finished using the program?"
Wbox* wb_empty = new Wbox();              //Create an empty Wbox
Wbox* wb_n_empty = new Wbox(mess1);       //Create a non-empty Wbox
wb_empty->refill(mess1);                  //Refill empty Wbox with mess1
wb_n_empty->refill(mess2);                //Refill non-empty Wbox with mess2
rb_empty->refresh(72);                    //Allocate buffer of size 72
Message* read_mess = new Message(FILEACT::Read, rb_empty, rb_empty->size());
                                          //Message for IA
read_mess->send(my_iact);                 //Send Read message to IA
rb_empty->wait();                         //Wait for read to be over
wb_empty->refill(rb_empty);               //Refill Wbox from Rbox
Wbox* wb_from_rb = new Wbox(rb_empty);    //Create Wbox from Rbox
char first_char = rb_empty->get(0);       //Read first character
char thirty_first = rb_empty->get(30);    //Read 31st character
rb_empty->refresh();
Message* write_mess = new Message(FILEACT::Write,wb_from_rb,wb_from_rb->size());
                                          //Message for IA
write_mess->send(my_iact);
```

4.0 Conclusion

We have implemented the ACT++ class library for the Sequent Symmetry multiprocessor using PREST 0.4. This library is meant to be used with AT&T C++ version 1.2 and any other compatible compilers. We are currently porting PRESTO onto SUN 3/80 and DECstation 5000/240. Our next target is to use PRESTO 1.0 to implement ACT++. This new version of PRESTO uses AT&T C++ 2.1.

We are also trying to extend the ACT++ class library in different ways so as to make it a better tool for concurrent object-oriented programming. An interesting issue we are addressing currently is the interaction between the explicit specification of the `become` operation in the methods of a behavior and the inheritance mechanism. Since the `become` operation synchronizes the initiation of the processing of the next message to an actor, like explicit synchronization primitives, it interacts with the inheritance mechanism to cause Inheritance Anomaly. A solution to this problem is to allow the `become` operation from the method which is requested in a message and suppress it in all methods - in the class of the requested method or in any superclass - which are invoked by it. An alternative solution is to disallow the explicit specification of the `become` operation from the methods of a behavior. We are considering the different possibilities and will implement it in a future version of ACT++.

Another issue which we want to address is extending the capabilities of interface actors to handle typed I/O. We have introduced the capability of handling null-terminated string I/O only so that we could concentrate on the details of implementing the asynchronous I/O capabilities. Now we will use the `ifstream` and `ofstream` classes available in the `iostream` library of C++ to handle typed I/O.

Another long-term goal which we have is to implement a garbage collector that would execute concurrently with an ACT++ application. Initial steps have already been taken in this regard [Kafura et al 90]. But the dynamic reconfiguration of object interconnections in an ACT++ application poses some inherent problems which preclude a straightforward implementation of a garbage collector. Those problems have to be addressed and a garbage collector should be implemented for ACT++.

4.1 Acknowledgements

We would like to thank Keung Hae Lee for his implementation of an earlier version of ACT++ [Lee 90]. We would also like to thank Vikul Khosla for an initial design of the I/O system.

References

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

[Agha and Hewitt 87] Agha, Gul, and Hewitt, Carl, "Concurrent Programming Using Actors," in Object-Oriented Concurrent Programming, A. Yonezawa and M. Tokoro (eds.), MIT Press, Cambridge, MA, 1987, pp. 37-53.

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

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

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

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

[Briot and Yonezawa 90] Briot, J-P, Yonezawa, A., "Inheritance and Synchronization in Object-Oriented Concurrent Programming," in ABCL: An Object-Oriented Concurrent System, (ed. A. Yonezawa), MIT Press, Cambridge, MA, 1990, pp. 107-118.

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

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

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

[Kafura et al 90] Kafura, D., Washabaugh, D., and Nelson, J., “Garbage Collection of Actors,” OOPSLA/ECOOP '90 Proceedings, October, 1990, pp. 126-134.

[Lavender and Kafura] Lavender, G. and Kafura, D., “Specifying and Inheriting Concurrent Behavior in an Actor-Based Object-Oriented Language,” Technical Report TR 90-56, Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, 1990.

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

[Matsuoka et al 90] Matsuoka, S., Wakita, K., and Yonezawa, A., "Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages," paper distributed at OOPSLA/ECOOP '90 Workshop on Concurrent Object-Oriented Programming.

[Mukherji 92] Mukherji, M., The implementation of ACT++ on a shared memory multiprocessor, M.S. Project Report, February, 1992, Department of Computer Science, Virginia Tech.

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

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