Proceedings of IEEE Southeastcon '92, April 12-15, 1992, Birmingham, AL, pp. 600-603.

CONCURRENT PROGRAMMING MODEL DEVELOPMENT IN C++

Viswanathan Vaidyanathan, Barrett R. Bryant
Dept. of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294
InterNet: vaidyana@cis.uab.edu, bryant@cis.uab.edu

ABSTRACT

Blending the object-oriented and concurrent programming paradigms will provide a strong programming platform of multiple, independent, concurrent, and interacting objects which can model real world problems in a natural fashion. To achieve this aim, we are working on the development of an Actor-like model in C++. The Actor model is based on the message passing mechanism and provides a conceptual foundation for massively concurrent object-oriented programs. The interest in this project is about how an inheritance-oriented language like C++ aids in the development of the actor model and how Unix interprocess communication facilities are useful in realizing the message passing mechanism. This model gives an additional set of constructs in C++ providing active objects.

1. INTRODUCTION

Object-oriented programming is a relatively new programming paradigm which facilitates the software engineering principle of reusability of code. It provides for capabilities of programmer-defined objects, classes, inheritance, data abstraction or information hiding, strong typing, and polymorphism. Objects are program modules which model and implement conceptual and physical entities that appear in problem domains. Derived classes inherit members of their parent classes and objects are instances of classes. Objects are self contained entities.

In a concurrent programming environment, participating entities execute concurrently. There is a necessity for interaction between the entities and this can be provided by shared memory, message passing or remote procedure calls [Andr83]. Message passing can be classified according to the send and receive mechanism, call and reply mechanism and synchronous and asynchronous mechanisms [Toml89].

In object-oriented programming a problem is represented in terms of autonomous objects which function independently of each other. In principle, these self contained objects may carry out computations in parallel [Hewi87]. So it is a natural extension to include concurrent programming capabilities in the object-oriented metaphor whose very aim is the development of multiple, independent and interacting entities [Yone87].

In this research we are working on the development of an actor-like model in C++. An actor model is a conceptual model for concurrent object-oriented programming based on the message passing paradigm. It is seen that an inheritance-oriented language like C++ aids in the development of the actor model. We find that the various Unix interprocess communication facilities are useful in realizing the message passing mechanism.

2. THE ACTOR MODEL

The Actor model is a concurrent object-oriented model proposed by [Agha86]. In this model, everything in the system is taken to be an actor. Actors are computational agents which carry out their actions in response to incoming communications. In principle, the concept of actors provides a conceptual foundation for massively concurrent object-oriented programs.

All actors are characterized by an *identity* and a *current behavior*. Once an actor is created, its identity is not changed but its behavior is changed in response to messages. The identity of the actor is represented by a *mail-address* and a current behavior is composed of a set of *acquaintances* and a *script* that defines the actions that the actor will perform upon receipt of the next message. Associated with the mail-address of an actor is a mail-queue that is used to buffer messages that have been sent to the actor but not yet received by some behavior.

In response to processing a communication targeted to an actor, the behavior of an actor consists of three kinds of actions:

a) An actor may send communications to specific actors whose mail address it knows, i.e., sending communication to its acquaintances.

b) An actor may specify a replacement behavior which

0-7803-0494-2/92 $3.00©1992 IEEE

will accept the subsequent communications.

c) An actor may create new actors.

3. DESIGN OF ACTORS AND THEIR ACTIONS USING C++

This section explains how we have developed the characteristics of the Actor model in C++ by also making use of the Unix interprocess communication facilities.

Rationale for the Development of Actor Model in C++

C++ is an extension of the C language which provides for object-oriented facilities [Stro86]. It includes features such as inheritance, data abstraction, function prototypes, etc. [Pohl89]. We can use C++'s higher level constructs with a good runtime efficiency. Development of the characteristics of an actor is aided by the basic class structure in C++ which can define member variables and functions. The inheritance mechanism for creating class hierarchies found in C++ makes it possible to create abstractions that support actor concurrency. The constructor facility in C++ is useful in instancing active objects. The message passing between the actors are implemented using the Unix interprocess communication facility, sockets [Sun88].

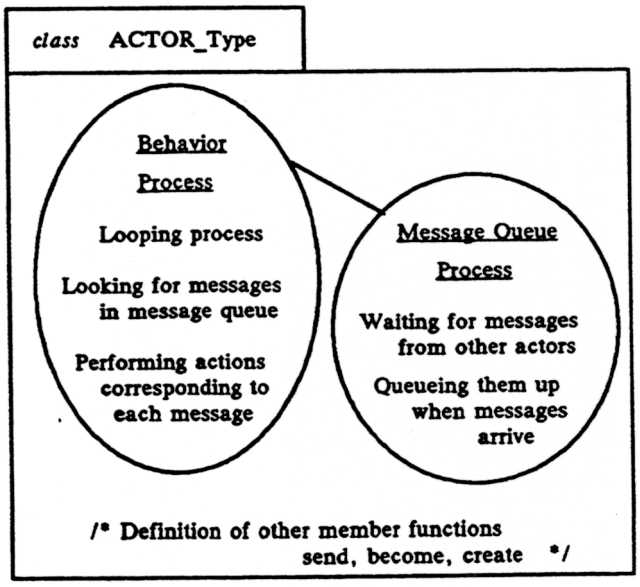
Basic Actor_Type Class Definition

An actor, essentially has a two-tuple representation as follows:

Actor_name(identity, behavior)

The basic structure of the actor is defined as a class ACTOR_Type. This class has various member functions for defining the behavior, message queue and mail address.

Since the actor is made of some active components like message queue and behavior, we require active Unix processes to define them.



Defining the Basic Actor Structure

At this level, the basic ACTOR_Type class is defined with two member functions for the behavior and message queue, which are to be forked as Unix processes. Other member functions which are part of this ACTOR_Type class are the ones which define the primitive actions of an actor - send, become and create.

```
class   ACTOR_Type {
  public :
    virtual   messages();
    virtual   script();
    virtual   behavior(/* script of actions */);
                        /* Unix process */
    virtual   mail_addr();
    virtual   message_queue();
                        /* Unix process */
    virtual   acquaintance_list();
    ACTOR_Type();        /* Constructor */
  protected :
    virtual   send(destination,message);
    virtual   become(what,message);
    virtual   create(who,message);
}
```

User level definitions

```
class      check_acc :
      public      ACTOR_Type
{
      public   :
        messages()  {
            withdraw . . .
            deposit . . .
        }
        script()      {
            action on withdraw . . .
            action on deposit . . .
        }
        mail_addr();
        acquaintance_list();
        check_acc();   /* Constructor */
}
```

Instance of the derived class

```
check_acc  check1;
```

Hierarchical Structure of the Actor Definition

User Level Definitions

In this subsection, we will examine what are the definitions needed by the user for using the base ACTOR_Type class

- The user derives a subclass of actor, specific to the problem, from the base ACTOR_Type class.
- The identity of the actor is established by defining the mail address as a socket to the message queue process.

- Different messages that can be received by that subclass actor are declared as member functions.
- The script of actions corresponding to these messages must be defined here.
- The different acquaintance actors of this actor class are listed.

Identity of the actor

The message queue process of each actor creates a stream socket, binds a name to it, and waits for receiving communication requests from other processes. This is the identity of that actor. The behavior process of actors which intend to send messages to other actors, connects with the socket created by the message queue process of the target actor and sends the message.

Instancing of an Active Object

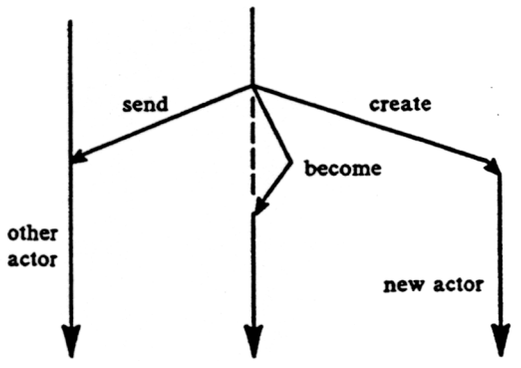
To instance an actor of the user derived class, which is an active object, we make use of the constructor facility in C++. In the base ACTOR_Type class, we have a constructor ACTOR_Type. Inside this constructor there are statements for forking the behavior and message queue processes. Various details defined at the user level, such as the messages that can be received and the script of actions corresponding to them, are passed back as parameters to this ACTOR_Type constructor using the modified constructor at the derived class level. So when an instance of this class is created, the corresponding active processes are forked thereby giving an active object.

Actions of an Actor

- The action of sending messages to different actors can be effected by the primitive:

send <target_actor> <message>

At the lower level, the behavior process of the sender actor connects through the common socket to the message queue process of the target actor and sends the message which gets queued up in the target actor's message queue.



Actions of an Actor

- Actors may be created by creating an instance of a class in C++. This new actor can inherit the acquaintances of the actor which has created it. When an actor is created, message queue and script processes are also created. For the created actor the mail queue is new. The behavior process of the original actor can proceed after the creation of the new actor.
- A replacement behavior is specified by identifying another actor that will control the processing of further messages. This is visualized as an actor *becoming* another actor. The message queue part of this actor is the same as the previous actor but only the behavior part of the actor is changed. So this actor starts executing all the messages next to the one which has initiated this actor. The behavior process of the original actor exits after specifying the replacement behavior.

4. EXAMPLE USING ACTOR MODEL

Let us consider an example of checking and savings account implemented using our model. The pseudocode for the behavior part of a 'checking account' actor is as follows:

```
check_acc :: script() {
if (message = withdraw(amount)) {
    if (amount > balance)
    become(overdrawn_check_acc,
                withdraw(amount))
    else send(balance, withdraw(amount));
    }
}
```

'Check_acc' is an actor and 'balance' is its acquaintance. If the amount to be withdrawn is less than the balance, a message is sent by the 'check_acc' to 'balance' to that effect. This illustrates the *send* action of an actor. If balance is less than the amount to be withdrawn, then 'check_acc' cannot service that request. Instead it becomes an 'overdrawn_check_acc' actor which responds to that message. This is a case of replacement of behavior.

Consider the situation where we allow the creation of an 'overdraft' actor. When the 'withdraw' request cannot be honored by the 'check_acc', it creates an 'overdraft' actor which tries to withdraw from the 'savings_acc', if enough 'balance' is available there.

```
if (message == withdraw(amount)) {
    if (amount > balance) {
    create(overdraft od_actor,
                withdraw(amount));
    send(self, insensitive);
    }
```

The 'overdraft' actor is created with the message that some 'amount' has to be withdrawn. While the 'overdraft' actor is working on the withdrawal, the 'check_acc' actor, which is indicated as 'self', should be just buffering all the subsequent messages except for the reply message from the 'overdraft' actor. This is achieved by making the 'check_acc' insensitive. The 'overdraft' actor will either succeed or fail in making the withdrawal depending upon the balance available in the 'savings_acc'.

5. CONCLUSION

In this paper we discussed the connection between object-oriented programming and concurrent programming. We discussed the development of the Actor model in C++ to achieve the advantages of both concepts. We gave a high level design of how the actor model is implemented in C++ by making use of the Unix interprocess communication facilities. So as an addition to the passive objects that can be instanced from C++ classes, this model gives an extra set of constructs which provide for active objects. Currently the model is being simulated on a uniprocessor machine. As future work we could extend this model so that it could be executed on different platforms like the shared memory Sequent Balance and other distributed and fine grained machines which would provide a more natural, efficient platform for this model.

6. REFERENCE

[Agha86] Gul A. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, The MIT Press, 1986.

[Andr83] G. Andrews, F. Schneider, 'Concepts and Notations for Concurrent Programming', ACM Computing Surveys, Vol. 15, No. 1, March 1983, pp. 3-43.

[Hewi87] Carl Hewitt, Gul A. Agha, 'Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming', in Research Directions in Object-Oriented Programming, eds. Bruce Shriver, Peter Wegner, The MIT Press, 1987, pp. 49-74.

[Pohl89] Ira Pohl, C++ for C Programmers, Benjamin-Cummings, 1989.

[Stro86] B. Stroustrup, The C++ Programming Language, Addison-Wesley, 1986.

[Sun88] SUNOS Reference Manual, Sun Microsystems, 1988.

[Toml89] C. Tomlinson, M. Scheeval, 'Concurrent Object-Oriented Programming Languages', in Object-Oriented Concepts, Databases and Applications eds. W. Kim, F. Lochovsky, Addison-Wesley, 1989, pp. 79-124.

[Vaid91] Viswanathan Vaidyanathan, 'Development of an Actor-like Model in C++', Proceedings of the 29th Annual ACM Southeast Regional Conference, 1991, pp. 323-328.

[Yone87] A. Yonezawa, M. Tokoro, Object Oriented Concurrent Programming, The MIT Press, 1987.