An Actor-Based Programming System

Roy J. Byrd
Stephen E. Smith
S. Peter de Jong*

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT: A programming system is described with which applications are built by defining collections of communicating objects, called actors. The actor programming system provides a uniform environment in which distributed applications can be automated in a highly modular and efficient manner. The system's design is based on the formal theory of actors, with certain modifications made for the sake of efficiency. We describe our view of the actor system, and an implementation of that view. We also discuss applications built on, and contemplated for, the actor system.

1. Introduction.

This paper describes a programming system developed within the context of the System for Business Automation (SBA) project (de Jong (1980), Zloof and de Jong (1977), de Jong and Byrd (1980)). In SBA, "boxes" represent business objects like documents, files and memos. Initially humans operate on the objects. Gradually the business processes are automated by imparting intelligence to the objects through the 2-dimensional SBA programming language. Eventually one obtains an automated business system in which intelligent business objects communicate among themselves without human intervention. These systems are naturally distributed, and we envision a network of processing nodes representing the functional components of a business organization, with objects such as memos and invoices flowing between these nodes.

Instead of directly implementing SBA in PL/I, we decided to first develop a programming system which provides the primitives needed by SBA. Such a programming system should support intelligent objects which communicate via messages, the ability to create instances of objects which have common structure and behavior, and the ability to easily modify the behavior of an object.

Our approach was to begin with the actor formalism developed at the M.I.T. Artificial Intelligence Lab (Baker (1978), Lieberman (1981)), and adapt if for use as a programming system. The actor formalism has the following properties, which are ideally suited to our objective.

1. Instances of actor types are the executing and communicating objects in the system. Their behavior is specified by a "script" and they have memory which persists between invocations.
2. An actor communicates with other actors, known as it's "acquaintances", via messages. It is transparent to an actor sending or receiving a message whether the other actor is at the same node or a different node in a network of computers.
3. Actors execute asynchronously, upon receipt of a message.
4. Actors are highly modular with well defined interfaces. One cannot look inside an actor. One only knows its behavior in terms of the types of messages it accepts and the way it responds to those messages.
5. Every object is an actor; in particular, the messages are actors.

The actor-based programming system intentionally deviates from a pure actor formalism in the granularity of the actor objects, and in the way in which they are defined. In a pure actor system absolutely everything is an actor, actors are defined in terms of other actors, i.e. there are no numbers or arrays. In the actor-based programming system the objects implementing an application behave as actors. However, the internal specification of an

* Current address: Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

actor is PL/I-like and contains PL/I-like data types. While an actor specification can be as small as two lines it can legitimately grow to be hundreds of lines.

Our break from a pure actor formalism is basically pragmatic, but is vital to the success of the actor formalism as a programming discipline. At its origins, actors were presented as a new model of computation, not as a new programming language (cf. Baker(1978)). Users were expected to program in LISP, and use a compiler to generate computations in terms of actors. Our objective is a system in which users can directly program the actors participating in an application.

In addition, a pure actor system implies an overwhelming amount of parallelism and communication. While this is appropriate for an array of microprocessors (cf. Hewitt(1980)), we do not foresee this kind of processing configuration becoming readily available in the near future. Instead, we expect a local network of conventional single processor computers. The cost of a message transmission in this environment will be large enough to preclude the level of message communication in a pure actor system. By defining fewer larger actor objects one can perform more computation within each actor and reduce the degree of communication between actors. This makes it possible to achieve a match between the amount of parallelism and communication in the system and the capabilities of the available processors.

There are other object oriented programming disciplines which have properties similar to actors. In particular, SIMULA (Birtwistle, et al. (1973)), Smalltalk (Ingalls (1978)), and "abstract data type" languages like CLU (Liskov, et al. (1977)). A thorough comparison of these programming languages would be of interest, but would take a considerable amount of effort, and is outside the intent of this paper. We adopted the actor formalism because it most closely matched SBA's requirements, it provided a sound theoretical basis to build upon, and it promised to yield the most compact and efficient implementation of the programming system that we sought.

Section 2 introduces an example which will be referred to throughout the paper. It will be used to demonstrate both how actors are defined, and how applications can be implemented as a collection of communicating actor objects. Section 3 describes the basic constructs in the actor language, and completes the definition of the major actors in the example application. Section 4 describes additional features not needed in the example. Section 5 deals with the implementation, in particular, those aspects of the

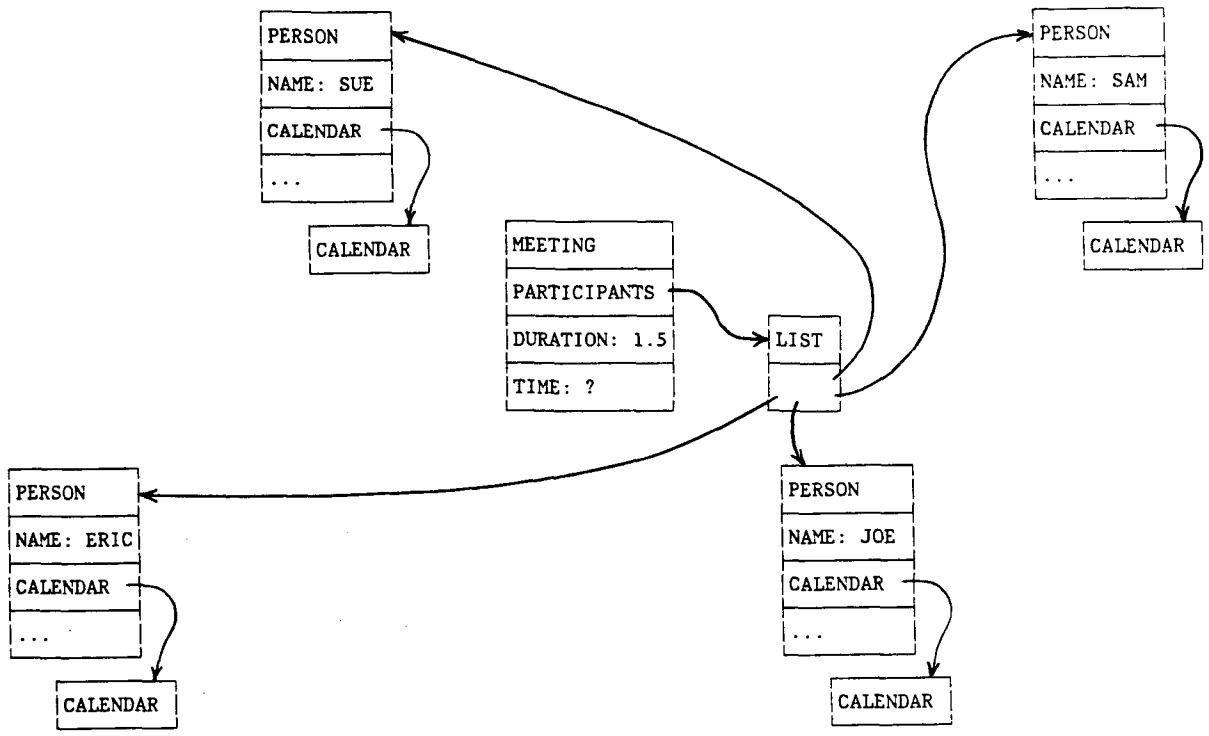

Figure 1. Scheduling a Meeting.

implementation concerning performance. Section 6 summarizes applications built with the current implementation. Section 7 summarizes the paper and describes the current status of the project.

2. An Example.

As an example of a business application implemented as a collection of actors, we consider the process of scheduling a meeting. A MEETING can be partially characterized by a list of its PARTICIPANTS, its DURATION and the TIME of the meeting. (We ignore other aspects of meetings such as topic, location, etc.) A meeting can be scheduled by inspecting the CALENDAR of each PERSON on the participant list, and scheduling the meeting for the earliest time period of the required duration which all calendars indicate to be currently unscheduled.

We will implement the scheduling process with the following actor types: MEETING, PERSON, CALENDAR, TIME_INTERVAL, TIME_INTERVAL_LIST, LIST, FIRST, and NEXT. Figure 1 depicts the major actors involved, with the arrows emanating from each actor representing its "acquaintances"--other actors known to it, and with which it can communicate. The scheduling process could proceed as follows:

1. A MEETING actor is created with a LIST of PERSON actors as its PARTICIPANTS acquaintance. The duration field is initialized.
2. The MEETING actor cycles over the PERSON actors on its LIST of PARTICIPANTS. It (asynchronously) broadcasts itself to all of the CALENDAR actors belonging to those PERSONs.
3. The CALENDAR actors, upon receipt of an unscheduled MEETING, create and return a TIME_INTERVAL_LIST containing the TIME_INTERVALs which a) are currently free, and b) exceed the DURATION required for the MEETING.
4. When a returned TIME_INTERVAL_LIST is received, the MEETING actor sends it to the POSSIBILITIES acquaintance, which is also a TIME_INTERVAL_LIST. POSSIBILITIES updates it's state to the intersection of its current state with the state of the TIME_INTERVAL_LIST message.
5. When all CALENDAR actors have responded, the MEETING sets its TIME field to be the TIME of the first TIME_INTERVAL in its POSSIBILITIES acquaintance. The newly scheduled MEETING actor again broadcasts itself to the original set of CALENDARs for inscription.

This solution to the scheduling problem exploits many of the properties of actors we feel to be useful. The PERSON and CALENDAR actors can be resident on separate personal computers, and distribution is obtained by merely sending messages to those actors. When MEETING broadcasts itself to the CALENDARs it initiates concurrent execution of the CALENDARs, thus taking advantage of natural parallelism in the application.

3. Actors.

Defining Actor Types.

Actor types are defined using the actor language, which is built on a subset of PL/I. An actor type definition has the following structure:

1. An ACTOR statement, giving the name and attributes of the actor type.
2. Zero or more INSTORE statements declaring instance variables.
3. Zero or more DECLARE statements declaring automatic variables.
4. Zero or more ACCEPT statements defining the actor's responses to messages.
5. Subroutines which can be invoked from within the ACCEPT clauses.
6. An END statement.

Figure 2 shows the overall structure of the definition of the MEETING actor described in the preceding example.

```
MEETING: ACTOR SERIAL;
  INSTORE
    PARTICIPANTS ACQ,
    DURATION FIXED,
    TIME FIXED,
    COUNT FIXED,
    POSSIBILITIES ACQ;
  DECLARE
    P ACQ;
ACCEPT('CREATE')
  ...
ACCEPT('TIME_INTERVAL_LIST')
  ...
END;
```

Figure 2. Outline of MEETING actor script.

One of the attributes which can be specified in the ACTOR statement is SERIAL. When specified, it forces the messages sent to an instance of the actor to be processed sequentially. Specifying NONSERIAL allows concurrent invocations of an actor.

The data types supported for automatic and instance variables are the PL/I data types plus the ACQUAINTANCE (ACQ) data type. An acquaintance variable can

contain the name of an actor. Having the name of an actor allows reference to that actor in a message transmission. The internal form of the name is not revealed to the programmer.

The INSTORE variables define the contents of **instance storage** for actors of the type being defined. The instance storage of an actor is a piece of storage which identifies the actor type, and contains variables whose values represent the current state of the instance. This state persists from one invocation of the actor to the next. Instance storage, therefore, is the permanent representative of the actor instance within the computing system. When an actor serves as the message in a transmission, it is the instance storage which is actually transmitted.

Each ACCEPT statement identifies a message type to be accepted, and the code to be executed upon receipt of a message of the specified type. If the code for an accept clause comprises more than one statement, it is enclosed in a DO-END block.

The source language specification for an actor type is compiled. This compilation process produces an executable **script** file and a template for the instance storage. Instance storage for actor's of type MEETING would be formatted as shown in Figure 3.

0	SCRIPT_ACQ
4	PARTICIPANTS
8	POSSIBILITIES
12	0
16	DURATION
20	TIME
24	COUNT

Figure 3. Instance Storage for MEETING Actors.

The SCRIPT_ACQ field is a required acquaintance variable which identifies a system defined SCRIPT actor. All instances of a given type point to a common SCRIPT actor which identifies the name of the type and the location of the executable script. Instance storage is allocated from a heap, and a garbage collector is used to reclaim the instance storage of actors which can no longer be accessed (i.e. those which no other actors have as acquaintances). Acquaintance variables are separated from the other variables in instance storage to facilitate garbage collection.

Creating Actor Instances.

An instance of an actor is created explicitly with a CREATE operation.

```
CREATE(type,parameter)
```

CREATE causes instance storage for an actor of type "type" to be allocated. If the script for that type contains an ACCEPT('CREATE') clause, the script is invoked and the create clause executed. The create clause specifies processing to be performed at the time of creation. The "parameter" in the CREATE operation is an actor which can be referenced in the create clause. In the absence of an ACCEPT('CREATE') clause, the instance storage for an actor is created with uninitialized instance variables. CREATE is a function which returns the new actor as its value.

Actor Communication.

An actor communicates with it's acquaintances by sending them messages. **Send** operations are provided for initiating **message transmissions.** A message transmission has four components which are set by the send operations. The first two are the **target,** which is the actor to receive the message, and the **message** actor itself. A third component, the **customer,** is the actor which should receive the reply message if one is generated as a result of the message transmission. The customer is frequently, but not necessarily always, the actor initiating the send operation. The last component in a message transmission, called the **client,** is a system defined actor used for synchronization, as will be described later. Note that all four components of a message transmission are necessarily acquaintances of the actor performing the send operation (the "sender"). The sender itself is not automatically a component of the resulting message transmission, although it may function as the target, message or customer.

Asynchronous communication is provided by a SENDC (Send-Continue) operation,

```
SENDC(target,message,customer,client).
```

The SENDC operation provides a "fork" capability, asynchronously sending the message to the target. Control returns immediately to the sender (i.e., the message will not necessarily have been processed), and the target and sender execute concurrently. In a SENDC operation one can specify all four components of the message transmission. If either the customer or client is not specified it defaults to the current customer or client, respectively, of the invoking actor.

The system provides three send operations for synchronous message communication. For all three, the sender specifies only the target and message components of the message transmission. The customer component is set according to the type of send operation performed. The client component is always set to the client of the sender.

```
SENDR(target,message)
```

A SENDR (Send-Reply) operation, provides a synchronous transfer of control similar to the procedural call-return mechanism. The sender serves as the customer, and waits for a reply message. The SENDR operation returns an acquaintance identifying a reply message.

```
SEND(target,message)
```

A SEND operation, is identical to SENDR except it does not expect a return message, and if one is generated, it is ignored.

```
SENDE(target,message)
```

A SENDE (Send-End) operation, differs from the others in that the invocation of the sender is terminated. The sender's current customer serves as the customer for the message transmission, rather than the sender itself, as in the other two synchronous send operations. As a result, any reply message generated by this transmission will be returned directly to this customer. A SENDE operation is functionally equivalent to

```
RETURN(SENDR(target,message))
```

but avoids an unnecessary change of contexts during the return processing. It is a convenient means of forwarding a message to another actor for processing.

Between any pair of actors, messages are guaranteed to be processed in the order in which they are sent. For any one actor the messages are processed in the order of arrival.

Invocation of Actors.

Every message transmission causes a new invocation of the target actor. If the target's instance storage is currently residing on secondary storage, it is read into main memory. If the script for actors of the target's type is not resident, it is dynamically loaded. Automatic storage is provided on a stack, and control is transferred to the actor's script. The script executes the accept clause corresponding to the type of the message received.

Special acquaintance values may be accessed during execution. These values are constant during any given invocation of an actor, although they may change from one invocation to the next. These special values are:

SELF - the name of the executing actor instance.

MESSAGE - the name of the actor whose receipt, as a message, caused the current invocation. In an ACCEPT clause for a "CREATE", MESSAGE is the name of a parameter actor.

CUSTOMER - the name of the actor which will receive any reply from the current invocation.

CLIENT - the client for the current invocation.

NULL - the empty actor reference.

NOTHING - the value which, when used in a RETURN statement, will suppress the sending of a reply to the customer.

When the target actor completes processing of the received message it returns either an actor acquaintance identifying an actor which constitutes the return message, a NULL acquaintance, or NOTHING.

The processing performed by an actor in response to the messages it accepts is independent of the manner in which they are sent. Its processing is the same whether messages are sent via SENDR, SEND, SENDE or SENDC operations. The actor which sends a message, however, must be aware of the kind of response generated by the target actor when it receives messages of the type being sent. In particular, it is an error to send a message via a SENDR operation if the processing of that message will not eventually produce a valid reply message.

Set and Get References.

The code in an actor's script may refer to the instance variables of another actor by using the following syntax.

```
acquaintance->type-name.instance-variable-name
```

The meaning of such a reference depends upon where it appears in a statement. If it is the target of an assignment, the meaning is that a new value is to be placed in the named actor's instance storage. This is called a **set** reference. When the expression appears in another context, it is a **get** reference, and means that the current value of the variable in the named actor's instance storage replaces the expression in that context.

The use of set and get references appears to violate the rule that actors may only inspect and modify their own instance storage. However, the violation is only apparent. These expressions are compiled into object code which (when necessary) creates and sends one of a special set of message actors to the actor which owns the instance storage. These messages are processed by accept clauses which are implicitly included in the actor's script because of its instance variables. These implicit accept clauses simply update or return the value of an instance variable. Thus, the actor itself performs the requested modification or returns the requested value. The parenthetical reference is to the fact that the object code may directly access the actor's instance storage when the results of a direct reference are indistinguishable from those obtainable by sending a message. Notice that this system does not inherently provide the benefits of "information hiding", as found in the abstract data type languages. Users who wish to approximate those benefits may replace the implicit

```
MEETING: ACTOR SERIAL;
 INSTORE
  PARTICIPANTS ACQ,
  DURATION FIXED,
  TIME FIXED,
  COUNT FIXED,
  POSSIBILITIES ACQ;
 DECLARE
  P ACQ;
ACCEPT('CREATE')
 DO;
  (Initialize PARTICIPANTS and DURATION.
   Set POSSIBILITIES to a TIME_INTERVAL_LIST
    encompassing all of the future.
   Initialize TIME and COUNT to zero.)
  DO P = SENDR(PARTICIPANTS,CREATE('FIRST'))
     REPEAT SENDR(PARTICIPANTS,CREATE('NEXT'))
     WHILE(P ¬= NULL);
   SENDC(P->PERSON.CALENDAR,SELF);
   COUNT = COUNT + 1;
  END;
 END;
ACCEPT('TIME_INTERVAL_LIST')
 DO;
  SEND(POSSIBILITIES,MESSAGE);
  COUNT = COUNT - 1;
  IF COUNT = 0 THEN
   DO;
    TIME = SENDR(POSSIBILITIES,CREATE('FIRST'))
              ->TIME_INTERVAL.TIME;
    DO P = SENDR(PARTICIPANTS,CREATE('FIRST'))
      REPEAT SENDR(PARTICIPANTS,CREATE('NEXT'))
      WHILE(P ¬= NULL);
     SENDC(P->PERSON.CALENDAR,SELF);
    END;
   END;
 END;
END;
```

Figure 4. Specification for MEETING actors.

accept clauses for the special set and get messages with clauses which explicitly filter or prohibit access to instance variables.

Completion of the example.

We now return to the problem of scheduling a meeting. With the facilities described so far, we can show how some of the actors constituting the solution might be implemented.

First, Figure 4 completes the definition of the MEETING actor which was outlined in Figure 2. The overall functioning of this actor has been described in section 2, and will not be repeated here. Notice, however, MEETING's use of the SENDC operation to asynchronously broadcast itself to the CALENDARs of all PERSONs on its list of PARTICIPANTS. The first broadcast occurs in the ACCEPT('CREATE') clause to request available time intervals. The second one announces the time of the meeting after receipt of the final time interval list.

Figure 5 shows the actors involved in the operation of time interval lists. Such a list consists of a main TIME_INTERVAL_LIST actor plus a singly-linked chain of ELEMENT actors. TIME_INTERVAL_LIST actors must have the SERIAL attribute because the instance variables, HEAD and CURRENT, can be modified during an invocation, and there is no guarantee that multiple users won't simultaneously send to a TIME_INTERVAL_LIST. (Note, in contrast, that an ELEMENT actor is only known to the actor which created it. Because of the pattern of messages sent to ELEMENT actors, it is clear that they do not need to be SERIAL, even though they, too, modify their instance storage.)

In the ACCEPT('CREATE') clause, a TIME_INTERVAL_LIST initializes its instance storage, and returns its own name to its customer. Upon receipt of a TIME_INTERVAL message, it creates and chains a new ELEMENT actor. (The chaining process is assisted by the "CREATE" code in the ELEMENT script.) It then sets the VALUE acquaintance of the new ELEMENT actor to the name of the new TIME_INTERVAL with the statement:

```
HEAD->ELEMENT.VALUE = MESSAGE;
```

A FIRST message causes a TIME_INTERVAL_LIST to position its CURRENT acquaintance at the HEAD of its list of ELEMENTs, and then to send itself a NEXT message. A NEXT message causes it to update its CURRENT acquaintance, and then to return the result of a **get** operation for the VALUE field of its old CURRENT ELEMENT with the statement:

```
RETURN(TEMP->ELEMENT.VALUE);
```

4. Additional Facilities.

Actor Names.

Users of the system only know that an acquaintance variable contains the "name" of an actor with which it can communicate. The internal form of the name is not revealed. The implementation, however, recognizes three different kinds of names, **local names, global names,** and **external names.** The local name of an actor is simply the address of it's instance storage. It provides direct access to an actor, but only exists while an actor is resident in main memory. Local names are not permanently assigned, and if an actor moves from one node to another or to secondary storage and back to main memory, it will at different times have different local names. A global name

```
TIME_INTERVAL_LIST: ACTOR SERIAL;
 INSTORE
  HEAD        ACQ,
  CURRENT     ACQ;
 DECLARE
  TEMP        ACQ;
ACCEPT('CREATE')
 DO;
  HEAD,CURRENT = NULL;
  RETURN(SELF);
 END;
ACCEPT('TIME_INTERVAL')
 DO;
  HEAD = CREATE('ELEMENT',HEAD);
  HEAD->ELEMENT.VALUE = MESSAGE;
 END;
ACCEPT('FIRST')
 DO;
  CURRENT = HEAD;
  SENDE(SELF,CREATE('NEXT'));
 END;
ACCEPT('NEXT')
 DO;
  IF CURRENT = NULL THEN RETURN(NULL);
  TEMP = CURRENT;
  CURRENT = CURRENT->ELEMENT.FPTR;
  RETURN(TEMP->ELEMENT.VALUE);
 END;
ACCEPT('TIME_INTERVAL_LIST')
 DO;
  (Modify current state by intersecting
   with the state of the message.)
 END;
END;
```

```
ELEMENT: ACTOR;
 INSTORE
  FPTR     ACQ,
  VALUE    ACQ;
ACCEPT('CREATE');
 DO;
  FPTR = MESSAGE;
  RETURN(SELF);
 END;
END;


TIME_INTERVAL: ACTOR;
 INSTORE
  TIME FIXED,
  DURATION FIXED;
END;


FIRST: ACTOR;
END;


NEXT: ACTOR;
END;
```

Figure 5. Additional actor specifications for meeting problem.

identifies an actor at another node. It is implemented as the local name of a system defined actor, of type GLOBAL, which contains the remote node's identifier and the local name of the actor within that node. An external name acquaintance value may be used to identify an actor which has been assigned an external name (a character string). It is implemented as the local name of a system defined actor, of type EXTERNAL, which contains the actual character string name. An actor identified by it's external name may be resident in main memory, on secondary storage, or at another node.

When an actor is created it only has a **local name.** It can be assigned an external name by means of a TITLE operation.

```
TITLE(acquaintance,external name)
```

where "acquaintance" identifies the actor instance to be named, and "external name" is a character string of the form "type-name.instance-name". An external name must be unique within each node. The system does not enforce uniqueness of external names throughout the whole network, and use of an external name will always result in a reference to a local actor with that name before it results in a reference to an actor at a remote node with the same name.

The external name of an actor can be used in scripts as an **acquaintance constant** which references the actor. For example, the phrase

```
'PERSON.ERIC'->PERSON.CALENDAR
```

can be used as a set or get reference to the CALENDAR instance variable of the actor instance ERIC of type PERSON. An acquaintance constant may appear anywhere in a script that an acquaintance variable is allowed, except as the target of an assignment operation.

Storing Actors.

Actors which have been assigned an external name may be saved on secondary storage with a STORE operation,

```
STORE(acquaintance)
```

Stored actors persist across session boundaries, and exist until explicitly destroyed. A stored actor is accessible via it's external name, and will be automatically read into main memory and assigned a local name when referenced.

It is not sufficient to store just the specified actor. Take for instance a named TIME_INTERVAL_LIST actor which uses a linked list of unnamed ELEMENT actors to identify the elements of the list. One could not store just the list actor, but would have to also include the list of element actors. Because of this, a STORE operation results in storing the following collection of actors.

1. The specified named actor.
2. All unnamed acquaintances accessible from that actor without an intervening named actor.
3. For each accessible named actor, a system EXTERNAL actor containing the actor's external name.

The collection of actors is stored in the form of a single variable length record called a **stored actor record.**

Remote Actor Invocations.

If the target of a message transmission identifies an actor at a remote node, then a system REMOTE actor is invoked in place of the target. The REMOTE actor interfaces with the network to cause the remote invocation of the target actor. The message actor is either moved to the remote node or a GLOBAL actor identifying the message is sent in its place. At the remote node the GLOBAL actor is interpreted as a global name acquaintance.

When an actor is moved to another node, it is done by forming a **network actor record.** A network actor record is the same as a stored actor record, except that the originating actor need not be named. Additionally, the acquaintances identified by local names, can either be included in the record or be represented by system GLOBAL actors. A network actor record is a portable form of an actor which can be sent around the network. Upon arrival at a node it is read into memory by the same mechanism used to read stored actor records.

Synchronization.

Several mechanisms are provided for synchronizing actor execution. Together, these mechanisms constitute a general architecture within which any desired synchronization control structure can be built.

At the lowest level, actors can be marked **non-serial** or **serial.** A message transmission to a non-serial actor can cause the invocation of that actor to occur immediately, even when the actor is already executing (with a message from a separate transmission). In order to support such concurrent processing, the code in all actor scripts is reentrant. Transmissions to non-serial actors are never queued. On the other hand, a transmission to a busy serial actor is placed on a FIFO queue of transmissions associated with that actor. This queue supports the guarantee, mentioned earlier, that messages will be processed by an actor in the order in which they are received.

The second level of synchronization is the **client.** Clients provide the "join" that corresponds to the "fork" capability given by send-continue (SENDC). CLIENT is a system actor type. When a non-NULL client actor is specified for a transmission started by a SENDC operation, it keeps track of the asynchronous execution path started by that SENDC. In particular, it keeps a count of such execution paths, incrementing the count for each new SENDC with that client, and decrementing the count when each path terminates. When the count reaches zero--i.e., all asynchronous execution paths begun with that client have ended--the client actor initiates a transmission with target and message actors which were specified at the creation of the client.

```
  ...
  MYCLIENT = CREATE_CLIENT(SELF,
                    CREATE('RESUME'));
  SENDC(ACTOR1,MESSAGE1,,MYCLIENT);
  SENDC(ACTOR2,MESSAGE2,,MYCLIENT);
  RETURN(NOTHING);
  ...
ACCEPT('RESUME')
  ...
```

Figure 6. Using Clients for Synchronization.

In the example in Figure 6, MYCLIENT is a client actor whose target acquaintance contains the name of the actor which created it (because of the reference to SELF), and whose message acquaintance is an actor of type RESUME. Before control is relinquished by the RETURN statement, the two SENDC operations will have caused the client's count to be set at 2. When the processing initiated by MESSAGE1 and MESSAGE2 has been completed, the client's count will have been decremented to zero. The client will then send the RESUME actor as a message to the actor which created it. Notice, finally, that the target actor has an accept clause for messages of type RESUME; this is the point at which the "join" operation occurs.

Activities are provided at the third level of synchronization. They are similar to clients in that they can monitor the progress of asynchronous execution paths, and they allow the specification of a transmission which is to occur at the termination of those paths. In addition, activities may be cancelled and are capable of being hierarchically nested within one another. If an actor executing as part of one activity issues a SENDC operation with another activity specified as client, then the transmission issued at the

completion of the child activity is considered to be part of the parent activity. Parent activities may execute concurrently with their children, but they are never considered complete until all of their children have either terminated or been cancelled.

```
  ...
  ACTIVITY1 = CREATE_ACTIVITY(SELF,
                    CREATE('RESUME1'));
  ACTIVITY2 = CREATE_ACTIVITY(SELF,
                    CREATE('RESUME2'));
  SENDC(ACTOR1,MESSAGE1,,ACTIVITY1);
  SENDC(ACTOR2,MESSAGE2,,ACTIVITY2);
  ...
ACCEPT('RESUME1')
 DO;
  CANCEL(ACTIVITY2);
  ...
 END;
ACCEPT('RESUME2')
 DO;
  CANCEL(ACTIVITY1);
  ...
 END;
 ...
```

Figure 7. Using Activities for Synchronization.

An example of the use to which the ability to cancel activities may be put appears in Figure 7. When either activity completes, the first action of the "join" code is to cancel the other activity. (Note that the join for the cancelled activity will never be executed.) If ACTOR1 and ACTOR2 were, for example, trying to produce the same result using different algorithms, this type of control structure would allow the cancellation of the losing algorithm as soon as the other one succeeds.

The activity mechanism is entirely implemented with the system actor type, ACTIVITY. The script for activity actors is illustrated in Figure 8. The instance variables of activity actors include a fixed point child count, as well as target, message, and parent activity. The child count and the parent activity acquaintance form the basis for the ability to nest activities. Every new activity becomes a child of the activity under which it is created. Part of the creation process for activities involves sending a YOU__HAVE__A__CHILD message to the parent, causing it to increment its child count. When an activity and all of its children have finished, it transmits its message acquaintance to the target, and sends a YOU__LOST__A__CHILD message to its parent. The parent, in turn, decrements its child count.

When created, each activity actor creates a client actor to manage its own actor invocations (as opposed to those of its children). Upon completion of these invocations, the client sends a YOUR__WORK__IS__DONE message to the activity actor which created it. When this happens, the activity will terminate as soon as it detects that its child count has reached zero.

```
ACTIVITY: ACTOR SERIAL;
 INSTORE
  TARGET ACQ,
  MSG ACQ,
  ORIGINATOR ACQ,
  PARENT_ACTIVITY ACQ,
  MY_CLIENT ACQ,
  MY_WORK_IS_DONE BIT(1),
  CHILD_COUNT FIXED;
ACCEPT('CREATE')
 DO;
  (Initialize the TARGET and MSG acquaintances.
  Set PARENT_ACTIVITY to the name of the current
  activity, or NULL if there is none.)
  IF PARENT_ACTIVITY ¬= NULL THEN
    SEND(PARENT_ACTIVITY,
         CREATE('YOU_HAVE_A_CHILD'));
  ORIGINATOR = CUSTOMER;
  MY_WORK_IS_DONE = '0'B;
  CHILD_COUNT = 0;
  MY_CLIENT = CREATE_CLIENT(SELF,
            CREATE('YOUR_WORK_IS_DONE'));
  RETURN(MY_CLIENT);
 END;
ACCEPT('YOU_HAVE_A_CHILD')
 CHILD_COUNT = CHILD_COUNT+1;
ACCEPT('YOU_LOST_A_CHILD')
 DO;
  CHILD_COUNT = CHILD_COUNT-1;
  IF CHILD_COUNT = 0 & MY_WORK_IS_DONE THEN
   DO;
    IF PARENT_ACTIVITY ¬= NULL THEN
      SEND(PARENT_ACTIVITY,
           CREATE('YOU_LOST_A_CHILD'));
    SENDC(TARGET,MSG,ORIGINATOR,NULL);
   END;
 END;
ACCEPT('YOUR_WORK_IS_DONE')
 DO;
  MY_WORK_IS_DONE = '1'B;
  IF CHILD_COUNT = 0 THEN
   DO;
    IF PARENT_ACTIVITY ¬= NULL THEN
      SEND(PARENT_ACTIVITY,
           CREATE('YOU_LOST_A_CHILD'));
    SENDC(TARGET,MSG,ORIGINATOR,NULL);
   END;
 END;
END;
```

Figure 8. Definition of the ACTIVITY actor.

5. Performance Considerations.

It must be possible to develop actor-based applications which meet acceptable levels of performance, i.e., performance comparable to a direct implementation in a high

level language. The following summarizes those aspects of the actor programming system directed at performance.

1. Actor scripts are compiled, rather than interpreted. Compiling will generally yield superior performance. The ability to define larger actors, of a size comparable to a procedure, allows one to capitalize fully on compiler optimization techniques.

2. Messages are passed by reference, and are copied only when it is necessary for transmission across a network. This is particularly important in an actor system where messages are actors and can be large. Actors, such as the MEETING actor in the example, frequently pass themselves to an acquaintance as a message, even though a much smaller message could have sufficed.

3. Symbolic references to actors (external names) are resolved to direct memory addresses. This avoids having to resolve symbolic names on every reference. In addition, a hash table is used to provide quick resolution of external names of actors already resident in main memory.

4. An executing actor-based application will typically create a large number of actors. Thus, actor instances must be inexpensive to create and have a minimal amount of space overhead. The processing cost of creating an actor is essentially the cost of allocating a structure in an area. The storage costs for an actor can be as small as 8 bytes, a 4 byte header and 4 bytes for the obligatory script acquaintance.

5. The large number of send operations which will be performed in a typical application make the cost of a send operation a significant factor in overall performance. Some send operations will be unavoidably expensive, requiring, allocation of a stack, resolving external names, network transmissions, dynamic loading of an actor's instance storage from secondary storage, and dynamic loading of a script. The majority of send operations, however, will be synchronous (SEND, SENDR, SENDE) transmissions where both the target actor's instance storage and script are already resident in main memory. The system design strives to make these transmissions as efficient as a normal procedure call. When the send mechanism is invoked the target acquaintance is the address of the target's instance storage, and the first acquaintance in that instance storage block points to a SCRIPT actor containing the address of the script to be executed. The send mechanism can simply transfer control to the identified script. The script runs on the current stack, i.e. the stack of the sending actor invocation. For SEND and SENDR operations it executes immediately above the sender, and for SENDE operations in replaces the sender (the sender's invocation is terminated). The net effect is that the target actor invocation occurs just above the customer actor invocation, as if the target had been called by the customer. This allows a message to be returned to the customer via a normal procedural return statement.

6. The process of storing an actor on secondary storage, and subsequently reading it back into main memory is a potential performance problem. The process involves a potentially large collection of actors. The format of the actors within a stored actor record is kept the same as in main memory, with acquaintances converted to offsets from the beginning of the record. This makes it possible, when reading a stored actor, to process the entire record as a unit, and avoid separate allocations for each component actor. The stored actor record is read directly into a block of storage within the area used for instance storage. A single pass is made over the the record to convert the acquaintance record offsets to local names. In addition, SCRIPT actors are not stored. Instead, their names are placed at the end of the stored actor record, and they are recreated when the record in read. External name acquaintances are intentionally not resolved when a stored actor is read. The send mechanism dynamically resolves the external name acquaintances upon first use, thus reestablishing connections to named actors which were severed during the storing process. A beneficial by-product of storing and reading an actor is that the actor and its unnamed acquaintances become physically adjacent in main memory.

6. Applications.

Part of the motivation for building the actor system is to explore the usefulness of the actor model of programming for building substantial applications. This exploration naturally involves the programming of actual applications.

An application is a subsystem of actors communicating with each other. The application designer must approach his task differently than if he were using a traditional programming system. For example, he cannot use global data structures, since actors provide none. He must decompose his problem into active entities--which he then implements as actors--rather than passive data structures operated upon by arbitrary control structures. The interfaces among these entities must be well-defined. Distributed applications become easier to define, since distribution is built into the actor model. Asynchrony is much more accessible than in other systems, and the designer should

incorporate it in intelligent ways. As the designer gains experience with the actor model, he is rewarded with increased ease in developing and implementing system structures for his applications.

We summarize some of the applications built, and being built, using actors.

Our first application was an implementation of the OMEGA knowledge representation system (Hewitt, Attardi, and Simi (1980)). OMEGA represents knowledge as a network of "descriptions". At the lowest level, descriptions represent the objects of the world being modelled, and the links in the network represent inheritance and attribution relationships among those objects. The more interesting aspect of OMEGA is that descriptions may also represent axioms which further describe the semantics of the world being modelled.

The implemented system provides a language in which OMEGA statements and requests can be entered. Statements are transformed into their corresponding descriptions and merged with the network of existing descriptions. Requests explore the inheritance and attribution links, under the control of the axioms in existence, and yield information to the user. Some of the description types supported, along with examples of the description language, are given below:

Atomic - `John`
Instance - `(a child)`
With-attribution - `(with sex male)`
Inheritance - `(is John (a child (with sex male)))`
Variable - `=x`
Implication - `(=> (is =x (a child (with sex male))) (is =x (a boy)))`

The implementation represents each description type as an actor type; actor instances, therefore, represent individual descriptions. The acquaintances of a description actor include the other descriptions linked to it in the semantic network. Additional actors provide such functions as user communication, input parsing, output formatting, and triggering of user-supplied axioms. Altogether, about forty actor types, with scripts ranging in size from two lines to four-hundred lines of actor code, were required for the implementation. During an OMEGA session, thousands of actor instances may be in existence at one time.

A second actor application is an end-user window interface to the SBA system. The interface gives the user simultaneous access to multiple overlapping windows on his terminal screen. All windows are active at all times; this contrasts with systems - like Smalltalk - in which the user interacts with a single "active" window, and in which activation of a new window forces a "focus shift" on the part of the user. Individual SBA windows are controlled by instances of a WINDOW actor type. Each one can change its own name, size, color, border, and height with respect to other windows. A window can also initiate office applications, manipulate multiple graphical images belonging to the applications, and scroll over these images, all in response to user commands. The window actors communicate with an actor of type SCREEN by sending it PICTURE messages. The screen actor manipulates the multiple pictures of windows, creating, moving, and overlapping them on command; it also notifies the window actors of user activity at the terminal, by sending modified PICTURE actors.

The SBA window system provides a unified application development and execution environment at an office workstation. Additional applications, also built of actors, operate within the window environment. These applications are "graphical" in the sense that they communicate with WINDOW actors about their status by sending and receiving PICTURE messages. They include:

- SBA tables and files which support Query-by-Example operations, and whose underlying relational access method was simulated by an OMEGA network,
- SBA forms,
- a personal calendar,
- a continuously running digital clock, and
- a lightpen-operated pocket calculator.

We must not forget that one of the more important applications of actors to date has been the implementation of the actor system itself. The use of actors to implement clients, activities, scripts, directories, loaders, and names, testifies to the generality of the actor programming model.

7. Summary and Status.

The actor-based programming system takes the formal theory of actors and adapts it for use as a language and system for developing applications. Whereas actor systems have previously been used only in artificial intelligence research, they can now be employed in other areas. In particular, we propose to use ours for programming distributed business applications. An application is implemented as a collection of independently executing actors communicating by messages. They can be distributed over a network of computing nodes, and permanently stored on disk. A naming architecture allows actors to communicate in a uniform way regardless of their location or status.

Actor types are defined through a PL/I-like language which specifies both a storage area to be associated with

the actor and its behavior in response to the messages it accepts. This behavior may involve changing the state of its instance storage area, creating other actor instances, initiating communication with its actor acquaintances, and returning a message to the actor which caused it to be invoked. Mechanisms are provided for synchronizing concurrently executing actors.

While the system retains the desirable properties of actors, performance has been a primary concern, and we have been careful to avoid the inefficiencies usually found in object-oriented message-passing systems. The size of an actor specification can vary from two lines to hundreds of lines of PL/I-like code, and is compiled with an optimizing compiler. Messages are passed by reference, and are only copied when transported across the network. Names of acquaintance actors are dynamically bound to direct memory references upon first use.

Status.

The prototype processor for the actor language was built using an existing optimizing compiler for a subset of the PL/I language together with the standard PL/I macro preprocessor (I.B.M. (1976)). Syntactic and semantic limitations imposed by the macro preprocessor resulted in the implemented language differing somewhat from the descriptions given in this paper (see Byrd (1980)). Continued development of the actor system should include the implementation of a compiler for the actor language.

Some of the described language features, such as the ability to cancel activities, have not yet been implemented. The concept of inheritance, which is similar the Smalltalk notion of **superclass,** is needed, as well as a mechanism by which an actor can delegate the handling of a message to another actor. The information hiding properties of the abstract data type languages, such as CLU, should be made available. The serial/non-serial attribute could be made a function of the message type being processed, and an actor's instance storage could be allowed to dynamically vary in size.

The existing prototype implementation is for a single actor node in a VM/370 virtual machine. It is being extended to multiple actor nodes residing in a network of virtual machines. This will simulate the intended processing configuration of a local network of small computers, and allow experimentation with distributed actor applications. This will be a vehicle for conducting research on the difficult problems confronting distributed applications, including deadlock detection/prevention, crash recovery, and distributed garbage collection. An actor-based system should provide the potential for unique and interesting solutions to problems in these areas.

Acknowledgements.

We wish to thank Carl Hewitt for many long and exciting discussions of the actor formalism and the description system OMEGA, especially during his stay at the IBM T. J. Watson Research Center in the summer of 1980. We also thank John Lucassen for his energetic work during the early stages of the actor system implementation, and Ken Niebuhr, the other member of the SBA group, for his insight during many discussions.

References.

Baker, H. G., Jr. (1978) **Actor Systems For Real-Time Computation,** M.I.T. Ph.D. Thesis. Laboratory of Computer Science Technical Report MIT/LCS/TR-197.

Birtwistle, Dahl, Myhrhaug, and Nygaard (1973) **SIMULA Begin,** Auerbach.

Byrd, R. J. (1980) "Macros and Coding Conventions for SBA Boxes," SBA project memo.

de Jong, S. P. (1980) "The System for Business Automation (SBA): A Unified Application Development System," in **IFIPS Congress 80 Proceedings,** North Holland Publishing Company.

de Jong, S. P., and R. J. Byrd (1980) "Intelligent Forms Creation in the System for Business Automation (SBA)," I.B.M. Research Report RC 8529.

Hewitt, C. (1977) "Viewing Control Structures as Patterns of Passing Messages," in **Artificial Intelligence** Vol. 8, pp. 323-364.

Hewitt, C. (1980) "The Apiary Network Architecture for Knowledgeable Systems," M.I.T. AI Lab Memo.

Hewitt, C., G. Attardi, and M. Simi (1980) "Knowledge Embedding in the Description System OMEGA," in **Proceedings of the First National Annual Conference on Artificial Intelligence** American Association for Artificial Intelligence, August 1980.

I.B.M. (1976) **OS PL/I Checkout and Optimizing Compilers: Language Reference Manual,** I.B.M. form GC33-0009.

Ingalls, D. H. H. (1978) "The Smalltalk-76 Programming System: Design and Implementation," in **Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,** pp. 9-16.

Lieberman, H. (1981) "A Preview of Act 1," M.I.T. AI Lab Memo 625.

Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert (1977) "Abstraction Mechanisms in CLU," in **Communications of the ACM,** Vol. 20, no. 8, pp. 564-576.

Zloof, M. M., and S. P. de Jong (1977) "The System for Business Automation (SBA): Programming Language," in **Communications of the ACM,** Vol. 20, no. 6, pp. 267-280.