Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo No. 625 June, 1981

A Preview of Act 1

Henry Lieberman

Abstract

The next generation of artificial intelligence programs will require the ability to organize knowledge as groups of active objects. Each object should have only its own local expertise, the ability to operate in parallel with other objects, and the ability to communicate with other objects. Artificial Intelligence programs will also the require a great deal of flexibility, including the ability to support multiple representations of objects, and to incrementally and transparently replace objects with new, upward-compatible versions. To realize this, we propose a model of computation based on the notion of an *actor*, an active object that communicates by *message passing*. Actors blur the conventional distinction between data and procedures. The actor philosophy is illustrated by a description of our prototype actor interpreter *Act 1*.

Keywords: Actors, object-oriented programming languages, message passing, artificial intelligence, knowledge representation, data abstraction, parallelism

Acknowledgements: This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Office of Naval Research under Office of Naval Research contract N00014-75-C-0522, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

A Preview of Act 1

Henry Lieberman

Artificial Intelligence Laboratory
and Laboratory for Computer Science
Massachusetts Institute of Technology

Section 1. The actor philosophy

1.1 Introduction

The *Message Passing Semantics Group* at MIT has been concerned with developing formalisms for expressing computations which meet the needs of artificial intelligence. Recently, there's been an important change of viewpoint in AI from modeling the kind of intelligence that is found in a single individual, towards modeling the kind of problem solving done in a *society* of people. We believe that organizing programs as cooperating individuals in a society will require a radical departure from the traditionally accepted models of computation.

To address the requirements of AI programming, we have developed a model of computation based on the notion of an *actor*. An actor is an active object which communicates with other actors by *sending messages*. To test out our theory, we have implemented an experimental programming language for constructing actor systems called *Act 1*.

1.2 Actors meet the requirements for organizing programs as societies

What capabilities are needed in a computational model to construct models of intelligent processes as a society of cooperating individuals? This section will present a number of principles which our experience has led us to believe are requirements which should be satisfied by any system claiming to be suitable for constructing intelligent programs. The remainder of the paper will show how these issues have been addressed specifically in our experimental actor implementation *Act 1*.

First, *knowledge must be distributed* among the members of the society, not *centralized* in a global data base. Each member of the society should have only the knowledge appropriate to his own functioning. He should not have to depend upon global knowledge which may not be relevant to his concerns.

We shall show how Act 1 distributes knowledge entirely in individual actors. Each actor has only the knowledge and expertise required for him to respond to messages from other actors. There's no notion of global state in an actor system.

In a society model, *each member should be able to communicate with other members* of the society, to ask for help and inform others of his progress.

In Act 1, all communication and interaction between actors uses message passing. Message passing provides a uniform communications mechanism for all actors. No actor can be operated upon, looked at, taken apart or modified except by sending a request to the actor to perform the operation himself. This protects the integrity of each actor.

Members of a society must be able to pursue different tasks in parallel. Putting many members of a society to work on different approaches to a problem or on different pieces of the problem may speed its solution enormously. Individuals should be able to work independently on tasks given to them by the society, or generated on their own initiative.

We will show how Act 1 allows a high degree of parallelism. Act 1 uses the object-oriented, message passing philosophy to provide exceptionally clean mechanisms for exploiting parallelism while avoiding the pitfalls of timing errors. Many actors may send messages simultaneously without interfering with each other. We try to assume a minimum of constraints on the ordering of events in an actor system to allow maximum parallelism. Eventually, we intend to implement Act 1 on a large integrated network of parallel processors such as the Apiary [10].

Different subgroups of a society must be able to share common knowledge and resources, to avoid duplicating common resources in every individual that needs them.

Act 1 uses the technique of *delegating* messages, which allows concentrating shared knowledge in actors with very general behaviour, and creating extensions of these actors with idiosyncratic behaviour more suited to specific situations.

1.3 Actors are active objects which communicate by message passing

The basic ideas of the actor model are very simple. There's only one kind of object in our model - an *actor*. Everything, both procedures and data, is uniformly represented by actors.

There's only one kind of thing that happens in an actor system - an *event*. An event happens when a *target* actor receives a *message*. Messages are themselves actors, too. We like to think of each actor as being like a person, which communicates with other people in the society by sending messages.

What does each actor have to know and be able to do to fulfill his role in the society?

Each actor in the system is represented by a data structure with the following components:

Each actor has his own behaviour when he receives a message, his own ways of responding to different situations. The *script* of an actor is a program which determines what that actor will do when he receives a message. When a message is received, the script of the target of the message is given control. If the script recognizes the message, he can decide to *accept* the message. If the script doesn't recognize the message, he *rejects* it.

Each actor knows about another actor to whom he can *delegate* the message if his script decides to reject the message. The *proxy* of an actor might be capable of responding to a message on the basis of more general knowledge than the original recipient had. Alternatively, the code for the script may also decide to explicitly delegate the message to some other actor.

Each actor has to know the names of other actors so that he can communicate with them. The *acquaintances* of an actor are the local data, or variables associated with each actor. Think of the acquaintances like telephone numbers of people. Each actor can call (send a message to) other actors, providing he knows their telephone number. Each actor starts out with a set of known telephone numbers, and can acquire new ones during his lifetime. We say that an actor *knows about* each of his acquaintances.

This simple framework is general enough to encompass almost any kind of computation imaginable. We shall discuss how the more traditional concepts used in

programming can be expressed within our actor model, and the advantages of doing so. Later, we shall make the model more concrete by describing how Act 1 is implemented in Lisp, and we will show how we fool Lisp into regarding ordinary data and procedures as active objects.

1.4 Why do we insist that everything be an actor?

Several previous and subsequent systems have embodied some form of active objects and message passing. What makes Act 1 different?

Act 1 takes the radical view that *all* computation should occur in the message passing paradigm. This results in an exceptionally uniform and elegant framework for computation. Act 1 is intended to provide a vehicle for an uncompromising test of the actor philosophy.

Less radical systems like Simula-67, Clu, Lisp Machine Lisp, [25], [29], [22] and others generally provide a special *data type* (or means of constructing data types) to represent active objects defined by a user, along with a special message passing *procedure* which operates on the special data type. However, the predefined components of the system such as numbers, arrays, and procedures are not considered as active objects. In a non-uniform system, a program must *know* whether it has been given an actor data type in order to use the message send operation. A program expecting a system data type cannot be given a newly defined actor. This limits the extensibility of such systems. (Smalltalk ([15] - [18]) is another language which shares our philosophy of uniform representation of objects.)

The actor theory requires that *everything* in the system, functions, coroutines, processes, numbers, lists, databases, and devices should be represented as actors, and be capable of receiving messages. This may seem at first a little dogmatic, but there are important practical benefits that arise from having a totally actor-oriented system. There's a strong kind of *modularity* that results from having a system which is made up completely of actors. This kind of modularity cannot be realized without sticking to our principles of organizing systems as actors.

Modularity is the ability to treat a part of a complex system as a *black box*. A system is modular if it can be broken up into a large number of small parts or *modules*, each of which is easily understood in isolation, independent of all the others. A user of a particular module should be able to rely on the *behaviour* of

that module without having to know details of the *physical representation* or *implementation* which do not concern him.

Since in an actor system all communication happens by message passing, the only thing that's important about an actor is how the actor behaves when he receives a message. To make use of an actor, the user just has to know what messages the actor responds to and how the actor responds to each message, not details like specific storage formats which may be irrelevant to the user's application.

Relying on actors and message passing makes systems very *extensible.* Extensibility is the ability to add new behaviour to a system without modifying the old system, providing the new behaviour is compatible. In an actor system, the user may always add a new actor with the same message passing behaviour, but perhaps with a different internal implementation, and the new actor will appear identical to the old one as far as all the users are concerned. We can also extend a system by introducing a new actor whose behaviour is a *superset* of the behaviour of the old actor. It could respond to several new messages that the old one didn't, but as long as the new actor's behaviour is compatible with the old, no previous user could tell the difference.

Conventional languages like Lisp tend to be very weak on introducing *new data types.* A conceptually new object must be introduced using pre-defined data objects like lists. The user must be aware of the format of the list to make use of it, and rewrite the program if the format changes.

Suppose we wanted to implement an actor representing a *matrix* of numbers. Matrices might accept messages to ACCESS an individual element given indices, INVERT themselves, MULTIPLY themselves with other matrices, PRINT themselves and so on.

Traditionally, a matrix might be represented as a two-dimensional array, and elements accessed by indexing. Multiplication and inversion would be functions which worked on the array representation. We can implement an actor which stores the matrix in this form.

(Descriptions of programs will be given in English, to avoid introducing the details of Act 1's syntax at this point. Important identifiers will be capitalized, and the text indented to correspond to the structure of the program.)

```
Create an actor called ARRAY-MATRIX.
 with one acquaintance named ELEMENT-ARRAY,
  which is a two-dimensional array of the size of the matrix.

If I'm an ARRAY-MATRIX and I receive an ACCESS message asking for an element,
 I look up the element in my ELEMENT-ARRAY.
```

But now, suppose we have the *identity* matrix, which is more efficiently representable as a procedure than wastefully storing elements of zeros and ones.

```
Create an actor called IDENTITY-MATRIX.

If I'm an IDENTITY-MATRIX and I get an ACCESS message,
 If all the indices are equal, return 1.
 Otherwise, return 0.
```

Alternatively, suppose the matrix is in a data base which resides at a remote site. A message to the matrix actor might result in communication over a computer network to retrieve it. The user wouldn't have to worry about the actual physical location of the data, or network protocols, as long as the elements appear when he needs them. Another plausible use for a different data representation would be a *sparse matrix*, where it would be more compact to encode the elements of the matrix as a list of indices of non-zero elements and their contents, since most elements would be zero. Here the matrix needs both a data structure and a procedure for accessing elements in its representation.

Many different representations of matrices may be present in a system, and implementing them as actors means that users can be insensitive to implementation decisions which do not affect behaviour. Since all users of matrices access them by sending messages, and all kinds of matrices respond to ACCESS messages, an IDENTITY-MATRIX can be used interchangeably with an ARRAY-MATRIX. A calling program doesn't have to know whether the matrix is represented as a data structure or as a procedure. In a more conventional language, introducing a new representation usually means the code for the users of the representation must be changed.

As well as being able to define multiple representations for new data types introduced by the user, it also makes sense to allow multiple representations for built-in system data types as well. In order to allow multiple representations for system data types, it must be possible for a user-defined data type to *masquerade* for a system data object like a number. This requires that the system treat objects like

numbers as full-fledged actors, with the ability to respond to messages. If a user designs a new object which obeys the message passing protocol of numbers, programs designed to operate on system numbers can use the new object as well. This ability to extend built-in objects is an area where all of the less radical languages such as CLU or Scheme are deficient.

In our matrix example, it might be desired to treat certain matrices as if they were scalars, as is often done in mathematics. The identity matrix could be represented as 1, and any matrix with the same element N in all diagonal entries and zero elsewhere could be represented as the constant N. Act 1 would allow the definition of these matrices to respond to many of the same messages as scalars by passing messages sent to the matrix along to its diagonal element. This would enable any program which used scalars to accept these matrices as well.

1.5 A uniform actor system makes managing parallelism easier

The actor model holds many advantages for systems which make large-scale use of parallelism.

Since knowledge is extremely localized in actor systems, it becomes easier to isolate subsystems of actors which can be run in parallel without interfering with each other. Since each actor has only the data and procedures relevant to his own operation, this avoids needless communication with global resources, which becomes a bottleneck in parallel systems.

Actors may be distributed across many processors running in parallel and message passing may involve communication across a network to reach other actors living on different physical processors. Since all communication is performed by message passing, a user need not know whether the actor resides on the same processor or on another.

Because of the principle that actors are defined by their behaviour and are independent of representation, we can create actors which allow or restrict parallelism, but otherwise behave identically to their serial counterparts. In other languages, a user program must be aware of whether it is dealing with an ordinary value, a parallel process, or a synchronized resource. This makes it more difficult to exploit parallelism.

Futures are actors which represent the values computed by parallel processes. They can be created dynamically and disappear by garbage collection when they are no longer needed. Other actors may use the value of a future without concern for the fact that the value is being computed in parallel. Synchronization is provided by *serializers*, which protect actors with internal state from timing errors caused by interacting processes. An actor may send a message to a resource protected by a serializer, in exactly the same manner as the message would be sent to the resource itself.

One of the primary goals of the Act 1 effort has been to explore the actor ideas in the context of parallel programming for AI. This topic merits a lengthy discussion, so we have chosen to defer an exposition of parallelism in Act 1 to a companion paper [2]. This companion paper discusses in detail the use of future and serializer actors to implement parallel control structures, as well as the implementation details of Act 1's parallel constructs.

1.6 An inventory of messages: EVAL and MATCH

There's no monolithic, centralized interpreter for Act 1 as there is for Lisp. Act 1 has a *distributed interpreter*, consisting of a set of predefined actors which respond to messages which correspond to the actions of a conventional interpreter. The best way to describe Act 1 is through looking at the common messages used in the system, and conventions for how the actors initially supplied with the system respond to these messages.

The interpreter is driven by messages which ask actors to evaluate themselves. These EVAL messages work like the EVAL function of Lisp, except, of course, that the code for responding to these messages is distributed throughout the system, and the user can define new kinds of actors which respond to EVAL messages differently. A list is defined to respond to EVAL by considering the first element of the list as a target, the rest of the elements of the list as a message, then sending the message to the target.

Symbols respond to EVAL by looking up their values as variables. There are also APPLY messages, which bear the same relationship to EVAL messages as the EVAL function does to APPLY in Lisp.

Some actors can be defined to handle the EVAL or APPLY message specially to control

evaluation of arguments in the message, replacing the mechanisms for FEXPRS and MACROS in MacLisp. They can decide to evaluate some arguments and not others (like FEXPRS), or return another actor to receive the EVAL message (like MACROS). EVAL and APPLY messages include an ENVIRONMENT, an actor which can be sent messages to LOOKUP the values of variables.

Act 1 allows the recipient of a message to describe the message he wants to receive using *pattern matching* to say what the message should look like. In place of Lisp's argument lists, an actor receiving a message has a pattern to which the incoming message is matched. Pattern matching is used in place of argument lists in Lisp to let an actor receiving a message Pattern actors receive MATCH messages, which ask if an object included in the match message will satisfy the description in the pattern. The MATCH message includes an environment, and matching can result in the binding of variables to the message or its parts.

Pattern matching is used to *name* messages, by matching an object to an identifier pattern which binds a variable to the message. Pattern matching is used to *test* objects for equality or data type. Objects can be used as patterns and will match only objects equal to themselves. There are patterns which will match only those objects belonging to a certain class. Pattern matching is used to *break up* composite data structures, to extract pieces from the data and work with them separately. A list of patterns used as a pattern will match objects which are lists, and recursively match each element of the pattern to each element of the object. New patterns can be defined by creating new actors which respond to MATCH messages. Pattern matching by MATCH messages constitutes another kind of distributed interpreter which is complementary to EVAL.

1.7 Equality is in the eye of the beholder

The fact that actors are defined only by their behaviour in response to messages is important because it allows many different implementations of the same concept to co-exist in a single system. Implementations using different representations to achieve various efficiency characteristics can be used interchangeably provided their message passing behaviour is compatible. Allowing multiple representations requires some flexibility in the definition of *equality*.

Testing objects for equality is done by sending actors EQUAL messages asking them whether they are willing to consider themselves equal to other objects. Matching

relies on these equality tests. Ours is a different kind of equality relation than appears in most systems. Since actors can have code for handling EQUAL messages, two actors are equal only by their mutual consent. Equality can be made to depend upon the circumstances, and the context in which the question is asked. Two actors which have different definitions can claim to be equal to each other if one considers its behaviour sufficiently similar to the other to warrant calling them equal.

Suppose we have actors for CARTESIAN-COMPLEX-NUMBERS represented with acquaintances who are the real and imaginary parts of the complex number. We might also like to have POLAR-COMPLEX-NUMBERS which are represented with which are represented with acquaintances for the angle and magnitude of the number. A CARTESIAN-COMPLEX-NUMBER must be able to consider itself EQUAL to an equivalent POLAR-COMPLEX-NUMBER.

```
Define an actor called CARTESIAN-COMPLEX-NUMBER:
 with acquaintances REAL-PART and IMAGINARY-PART.

If I'm a CARTESIAN-COMPLEX and I'm asked if I'm EQUAL to ANOTHER-NUMBER:
 I ask the actor in the EQUAL message ARE-YOU a COMPLEX-NUMBER?
 If he says no, I answer NO.
  If he says yes, I ask him for his REAL-PART, call it HIS-REAL-PART.
   Then I ask my REAL-PART whether he's EQUAL to HIS-REAL-PART.
   and if so, I ask my IMAGINARY part
    whether he's EQUAL to the other's IMAGINARY-PART.
    and if both parts are equal, I answer YES.
   If either part is different, I answer NO.
```

This code says that the CARTESIAN-COMPLEX-NUMBER will consider himself equal to any other actor who also thinks he is a complex number and has suitable real and imaginary parts. We assume the code for POLAR-COMPLEX-NUMBER can figure out its real and imaginary parts from the angle and magnitude. CARTESIAN-COMPLEX-NUMBER should also be able to furnish its angle and magnitude for the benefit of actors like POLAR-COMPLEX-NUMBER.

A slightly unusual characteristic of the equality relation as we have it here is that is *asymmetrical*. Since one actor gets a chance to field the message before the other does, asking the question in the other order may have different results, although in practice that almost never happens. A more symmetric way to set up this example would be for both CARTESIAN-COMPLEX and POLAR-COMPLEX to delegate messages to a more general COMPLEX actor where knowledge about how to convert between the various representations would reside.

The equivalent of *type checking* is performed in Act 1 with ARE-YOU messages. There are no data types in Act 1 in the sense of conventional *typed* languages like Pascal. Variables can name objects of any type, just like Lisp. But it is useful to be able to ask an actor what kind of actor he is, to help predict his behaviour, or compare him with other actors. The philosophy we adopt is that each actor will know what type or types he considers himself to belong to.

Since an actor can respond to messages by delegating the message to a proxy, the actor conceptually inherits the type of his proxy. A CARTESIAN-COMPLEX-NUMBER might delegate messages to a proxy which holds information common to COMPLEX-NUMBERs, which might in turn delegate to a NUMBER actor. So the CARTESIAN-COMPLEX should answer yes when asked "ARE-YOU a COMPLEX-NUMBER?", or "ARE-YOU a NUMBER?".

1.8 Continutations implement the control structure of functions

The message sending primitive in Act 1 is *unidirectional.* Once a target receives a message, the script of the target has complete control over everything that happens subsequently. There needs to be some way of sending a *request* to a target actor, and receiving a *reply* to answer the question asked. This bidirectional control structure is like functions or subroutines in conventional languages.

The Act 1 mechanism for implementing function call and return control structure uses *continuation* actors. A continuation is an actor which receives the answer to a question, and which encodes all the behaviour necessary to continue a computation after the question is answered. A continuation is the actor analogue of a *return address* for subroutines [26] .

When an actor sends a REQUEST message (corresponding to a function call), the message includes a component called the REPLY-TO continuation, which tells the target who to send an answer to. When the target decides to furnish an answer, he sends a REPLY message (corresponding to returning from a function) to the REPLY-TO continuation received as part of the REQUEST message. The answer is included in the REPLY message.

Lest the reader worry that writing out requests and replies explicitly would be a burden on the user, rest assured that it is seldom necessary. REQUEST and REPLY messages are automatically supplied by the Act 1 interpreter whenever the user writes code with the function call syntax of Lisp.

Continuation actors are usually freshly created whenever a request message is sent. Replies are not usually sent directly to the actor who made the request, but to a new actor whom the sender creates to receive the answer. An important optimization is that when the *last argument* to a function is evaluated, the caller's continuation is passed along instead of creating a new continuation. This allows so-called *tail recursive* calls (where the last action in a function's definition is a call to that function itself) to be as efficient as *iteration.*

Nested function calls produce a *chain* of continuations, each of which knows about another continuation, like a *control stack* for Lisp. However, since the lifetime of a continuation may extend beyond the time after a REPLY message has returned as answer, continuations cannot be stored on a conventional stack.

Having continuations becomes important in situations where programs need to get hold of an object which represents the behaviour of the program following a return. Such a situation arises with communication in parallel systems, where an activity running concurrently with another may need to wait for some condition to become true. The program can store away the continuation of the waiting activity, wait for the condition to become true, then issue a reply to the stored continuation, resuming the activity.

Complaint continuations are another kind of continuation which represent the behaviour to be taken when an *error* condition is encountered. A *complaint* is an actor which receives error messages and takes corrective action, or calls a debugger. By explicitly managing the *complaint* continuation, a user can set up *error handlers* which can look at the error message and decide to take action, or delegate the message to more general error handlers. Separating the complaint continuation from the reply continuation was done only for implementation convenience. Instead, one continuation actor could serve to collect both replies and complaints.

Section 2. Delegation

2.1 Knowledge is shared by delegating messages

Whenever an actor receives a message he cannot answer immediately on the basis of his own local knowledge and expertise, he *delegates* the message to another actor, called his *proxy.* Delegating a message is like "passing the buck". The actor

originally receiving the message, whom we will call the *client*, tells his proxy, "I don't know how to respond to this message, can you respond for me?".

Many client actors may share the same proxy actor, or have proxies with the same script. Very general knowledge common to many actors may reside in a proxy, and more specific knowledge in each client actor which shares that proxy. This avoids the need for duplicating common knowledge in every client actor.

Delegation provides a way of *incrementally extending* the behaviour of an actor. Often, actors existing in a large system will be *almost correct* for a new application. Extension is accomplished by creating a new client actor, which specifically mentions the desired differences, and falls back on the behaviour of the old actor as his proxy. The client actor gets first crack at responding to messages, so he can catch new messages or override old ones.

Delegation replaces the *class, subclass and instance* systems of Simula, Smalltalk and Lisp Machine Lisp [25], [16] It provides similar capabilities for sharing common knowledge among objects, but since delegation uses message passing instead of a low level built-in communications mechanism, delegation allows more flexibility.

2.2 New actors are created with CREATE messages

We will give each actor the ability to create new actors similar to himself. For creating new objects, we introduce a message called CREATE, which we expect all actors to be able to respond to. When an actor receives a CREATE message, he produces a new copy of himself.

However, the new copy doesn't have to be exactly like the old copy. We can include in the CREATE message a list saying how we would like the new copy to differ from the old copy. The differences are specified in the form of new values for acquaintances of the actor.

Usually, the target of the CREATE message responds by creating a new actor whose script is the same as the script of the target. Actors with the same script are of the same "type" because the script of an actor determines his behaviour.

For each acquaintance of the actor, if a value for that acquaintance is specified in the CREATE message, that value appears in the newly created actor. If no value is specified in the CREATE message, the value is copied from the target.

Each actor is like a *prototypical* member of a set. When we are asked to create a new object, we can use as much specific information about what the new object should look like as is available in the request. Any other information which is necessary to create the new object is taken from the prototypical object.

Individual actors can intercept the CREATE message, so that special action can be taken when an actor is created. This is a convenient way to *initialize* newly created actors.

2.3 Extending the behaviour of actors with EXTEND messages

As large systems evolve, it's often useful to be able to improve on the expertise of existing actors, creating new ones which respond to additional messages or which modify the behaviour for existing messages. This is implemented by creating a new actor which has a pre-existing actor as his proxy, or a proxy with a pre-existing script.

The convention we adopt for extending actors is that actors respond to an EXTEND message. The EXTEND message creates a new script for the new actor which can have handlers for new messages. The EXTEND message also contains the names and values of new acquaintances for the new actor, and values of acquaintances of his proxy.

2.4 Implementing default behaviour for messages

When an actor doesn't understand a message, he delegates the message to his proxy, who might be able to respond to the message based on more general knowledge. The proxy may then delegate the message to his proxy, and so on. Eventually, the process of delegation must stop somewhere.

We introduce a distinguished actor called OBJECT, which contains the most general knowledge common to all actors. The script for OBJECT can contain default responses for all messages that should be understood by every actor even if there's no code in the script of the actor or any of the actor's proxies to explicitly handle that message.

The behaviour we described above for common system messages like CREATE can be

implemented by putting that behaviour in the script for OBJECT. Every actor will then respond to a CREATE message with the default behaviour. It is very important for messages like PRINT to have such default behaviour, so that every actor can be printed, even if the user writes no code for PRINT. Any particular actor who disagrees with the default behaviour in OBJECT may put code in his script to intercept the message and produce different behaviour.

Alternatively, we could put the default behaviour with *each message* instead of with a universal proxy like OBJECT. Messages could be actors who know their default behaviour if a target fails to accept them. The script for OBJECT would then simply turn around and ask the message if he had any default behaviour and execute that. Associating default behaviour with messages is preferable since it makes it easier to incrementally add a new kind of message to an existing system. The default behaviour is introduced when the new message is defined, instead of modifying OBJECT.

2.5 Implementing turtle graphics illustrates delegation

Using delegation, we will develop some actors to draw pictures using *turtles* as in the language Logo [30], [31]. Turtles are objects which keep track of a position on the screen, and a heading. They respond to FORWARD and RIGHT messages, and have a PEN which can draw when the turtle moves.

First, let's develop the notion of a POSITION, which keeps track of X and Y coordinates. We'll allow positions to MOVE to different places on the screen. The actor POSITION is created by extending OBJECT, since POSITION need not have any special behaviour other than that common to all actors. He has acquaintances named X and Y, given initial values in the center of the screen.

```
Create an actor called POSITION by
 Sending to OBJECT a message to EXTEND himself, with new acquaintances:
  named X, with value 0,
  and Y, with value 0.
```

The script will contain one handler for a MOVE message, and rely on OBJECT for all other messages.

```
If I'm a POSITION and I get a message to MOVE to a NEW-X and a NEW-Y,
 I update my X to the NEW-X and my Y to the NEW-Y.
```

Messages other than MOVE messages, such as messages to access and update acquaintances, will be delegated to OBJECT. New positions are created by sending the actor POSITION a CREATE message.

```
Create an actor called ANOTHER-POSITION by
 Sending to POSITION a CREATE message
  with X value 100,
  and with Y value 200.
```

If we hadn't included the values for X and Y, they would default to 0, just like in POSITION. Changes to the acquaintances of ANOTHER-POSITION don't affect POSITION. Any changes to the script are reflected in both since they share the same script.

Now, let's extend the notion of a position to keep track of a heading and be able to accept FORWARD and RIGHT messages.

```
Create an actor named TURTLE-POSITION by:
 Sending to POSITION an EXTEND message,
  with a new acquaintance named HEADING, initially 0.

If I'm a TURTLE-POSITION and I receive a message to turn RIGHT some DEGREES:
 I update my HEADING to be the sum of my present HEADING and DEGREES.

If I'm a TURTLE-POSITION and I receive a message to go FORWARD some STEPS:
 I increase my X coordinate by,
  the product of the number of STEPS and the SINE of my HEADING.
 I increase my Y coordinate by,
  the product of the number of STEPS and the COSINE of my HEADING.
 and I return my new position
```

TURTLE-POSITION is now an actor whose script has handlers for FORWARD and RIGHT messages. He has one acquaintance, HEADING, and his proxy is a POSITION, which has acquaintances named X and Y. If the message is FORWARD or RIGHT, the script for TURTLE catches the message, otherwise the message is delegated to the POSITION, which can respond to MOVE. In all other cases, the message is passed to OBJECT.

We can further extend TURTLE-POSITION to TURTLE by adding an acquaintance, the

PEN, which draws lines on the screen when the turtle moves. TURTLE intercepts the FORWARD message to draw a line between the TURTLE's old and new positions. TURTLE would delegate the FORWARD message to his TURTLE-POSITION to compute the new position.

2.6 The expertise of different actors can be combined

Several extensions to the same actor can be *orthogonal.* If we create two different extensions to the same actor, each of which responds to a new message, and the two messages don't interfere with each other, we might like to create an actor which can respond to both of the two new messages. Each extension has his own expertise, we should be able to combine the expertise of several actors in cases where their behaviour is compatible. An actor can accept a message to COMBINE himself with another actor, returning a new actor which combines the expertise of both.

The CONTRAST message allows us to construct a new actor selecting just those features of an already existing actor which are desired for a new actor. CONTRAST messages are supposed to return an actor representing the *differences* in behaviour between a given actor and the target of the CONTRAST message.

Suppose we extended TURTLE to VISIBLE-TURTLE, which displays a marker to indicate his position on the screen. Suppose also we have LIZARD, a different actor which also would like to display his position on the screen. We can extract the feature we need from TURTLE and combine it with LIZARD.

```
Create an actor called VISIBLE-LIZARD by:
 Sending a COMBINE message to:
  LIZARD, and
  the result of sending a message to:
   VISIBLE-TURTLE to CONTRAST himself
   with TURTLE.
```

See [19] for some discussion of combining orthogonal capabilities of objects in Smalltalk.

Section 3. Implementation issues

3.1 Peaceful co-existence between actors and Lisp

The process of constructing a concrete implementation for the actor ideas has helped clarify the issues involved. The remainder of this paper will describe the solutions that have been adopted in *Act 1*, our experimental interpreter written in MacLisp [21] for the PDP-10. Another implementation is currently in progress for the MIT Lisp Machine [22].

The actor model is attractive for its simplicity and elegance. But can it made to work as a practical tool? There are a number of questions that have to be resolved in implementing the actor theory. Can the kinds of computation done in more traditional formalisms be easily expressed in the actor model? How does an actor system keep from getting caught in an infinite loop of sending messages to actors, causing more messages to be sent to other actors, without any useful computation being performed? The recursion of actors and messages must stop somewhere, some primitive data types and procedures are needed. Yet the implementation should remain faithful to the theory, which says that all components of the system are treated as actors and obey the message passing protocol.

Ideally, we might like to have an *actor machine*, which deals with everything as actors, right down to the lowest level of the hardware. However, the design of most machines available today is poorly suited for implementing actors. The machine thinks in terms of numbers, registers, and instructions rather than active objects and message passing. We expect that in the near future, machines can be designed which offer a congenial environment for implmenting actor languages.

Meanwhile, we must make compromises with existing machine architectures. In implementing Act 1 in Lisp, we must make it appear to the user *as if* everything in the system is an actor, and all interaction happens via messages. But at some level, the implementation must get down to dealing with ordinary Lisp objects, which aren't actors. How do we create the illusion of actors on a machine which doesn't believe in them?

The answer is, we *cheat*, but cheating is allowed as long as we can't get caught! At the implementation level, everything isn't really an actor, and we must be able to get some real computation (like adding or printing) done by invoking primitive machine

operations, not by sending messages. The ground rules are that the implementation is allowed to violate the actor model only when it is guaranteed to be invisible to the user.

Efficiency is an important consideration. Since conventional machines work on the Von Neumann model, actors must be simulated. This simulation incurs a certain cost, but the overhead must be kept down to a reasonable level. Message passing must be efficient enough so that its cost is not prohibitive even for the simplest operations. Can we employ shortcuts for efficiency without compromising the integrity of the actor model?

In addition to cheating to make sure the actor interpreter is well founded, cheating can also be done to improve efficiency. As long as an actor behaves according to the message passing rules, the implementor is always free to use more efficient procedures behind the scenes to accomplish that behaviour.

It's also important to provide a smooth interface with the host language, Lisp. Lisp functions should be callable from actor programs, and Lisp data usable without requiring explicit conversion to a different representation. This means that we can build upon all the existing facilities in Lisp without having to duplicate them in our actor language.

3.2 Rock-bottom actors prevent infinite regress of actors and messages

How does an actor interpreter perform some computation like adding two numbers? The actor interpreter must be grounded in both the procedures and data of the implementation language. Numbers can only be added using the primitive addition operation of the implementation language, which only works on the machine's representation of numbers.

We can have actors whose acquaintances are actors, who in turn know about other actors, but the actor data structure must terminate in the primitive data of the implementation language and do not have pointers to any other actors. We can perform computations by sending messages to actors which in turn send messages to other actors, but some actors must have the ability to reply to messages they receive without sending any more messages.

There are a set of actors called *rock-bottom actors* which are allowed to cheat on the

actor model and use the primitive data and procedures of the implementation language. Some data objects in the implementation language can be considered as actors in their own right, and can receive messages. Which actors are considered rock-bottom actors will, of course, depend on the base language, but in Lisp they are *numbers*, atomic *symbols*, and *lists*. Instead of representing a number by an actor with a stored NUMBER-SCRIPT and the value of the number as an acquaintance, we represent the number actor using just the machine representation of the number itself.

We don't have to store the script or the proxy of the actor with the data, because the interpreter can be fixed to know specially about these data types. The object itself is like an acquaintance of the actor - it contains all the information necessary for its script to respond to messages. Since there are only a fixed number of such types known in advance, the interpreter can always find the script corresponding to a particular rock-bottom actor by looking in a table indexed by the type of the object.

Actors which have explicitly stored scripts and proxies we will call *scripted* actors. These can be implemented as a vector, record structure, or one-dimensional array containing script, proxy, and acquaintances. The implementation must have some way of being able to tell whether an actor is a rock-bottom actor or a scripted actor just by looking at it.

We are now in a position to describe how the fundamental loop of the Act 1 interpreter works:

```
Here's what happens when an EVENT occurs:
The EVENT consists of a TARGET receiving a MESSAGE.
 Check to see if the TARGET actor is a rock-bottom actor.
  If so, find the script of the actor by
   Looking up the type of the actor in the table of ROCK-BOTTOM-SCRIPTS.
   and invoke the script.
    The script may access the TARGET actor like an acquaintance.
  If the actor is a SCRIPTED actor,
   Extract the script and invoke it.
   The script can access the acquaintances and proxy,
   which are stored in the actor itself.
 If the SCRIPT REJECTs the MESSAGE,
  the MESSAGE is DELEGATED to the TARGET's PROXY.
 The SCRIPT causes a new EVENT, with a new TARGET and a new MESSAGE.
  The new TARGET and MESSAGE may come from:
   The ACQUAINTANCES of the TARGET, or
   the MESSAGE, or
   an actor newly CREATED by the script.
```

There are a special set of scripts, *rock-bottom scripts*, which are allowed to directly operate on an actor without sending messages. When is this safe? Since the script of an actor determines how it will behave, recognizing the script of an actor allows a rock-bottom script to ascertain what that actor is supposed to do and take appropriate action with primitive operations. The code for rock-bottom scripts is written in the implementation language, and these scripts are supplied with the initial system. A compiler may also convert user-written scripts to rock-bottom scripts for efficiency.

We will illustrate the relationship between rock-bottom actors and scripted actors by showing how *numbers* work in Act 1. Numbers respond to messages asking if the number is EQUAL to another number. Numbers are rock-bottom actors, which are represented using Lisp numbers. The user may have defined all kinds of number actors, like complex numbers, or infinite numbers, which may have code to receive EQUAL messages. It must be possible for a user-defined number to consider itself equal to a Lisp number, and vice versa.

When the number is sent a message, it is recognized as a rock-bottom actor by the interpreter. The interpreter finds the script corresponding to Lisp numbers, and invokes it. The script for numbers checks the script for the other number in the message, and sees if he can answer definitely yes or no. If he can't, then he turns around and sends a message to the other number, giving him a chance to respond.

```
Define the rock-bottom script for LISP-NUMBER actors:
for a particular NUMERICAL-VALUE of a number.

If I'm a LISP-NUMBER and I'm asked if I'm EQUAL to ANOTHER-NUMBER:
 I check to see if he's a rock-bottom actor,
  using the primitive Lisp test for rock-bottom actors.
  If so, then if he's EQUAL to my NUMERICAL-VALUE using Lisp's EQUAL,
   Then I answer YES.
  If he's not EQUAL to me, I answer NO.
 If he's not a rock-bottom actor,
  Then he might consider himself equal to me, so I turn around and
  send him an EQUAL message asking whether he thinks he's equal to me.
```

It's also possible to introduce different kinds of equality for different purposes. It would be useful to have a three-valued equality, which could return *yes*, *no*, or *maybe*, indicating the actor didn't have enough information to make a judgment. This would help avoid the situation of two actors unfamiliar with each other getting in a loop, each trying to pass the buck to the other.

A final problem concerns calling functions written in the implementation language from Act 1. Lisp functions require standard Lisp objects as arguments, not actors. It is necessary for the interpreter to check when a Lisp function is being applied to an actor argument. The actor can be sent a LISP-APPLY message telling it that it was the argument to a Lisp function. The message includes the function to be applied, the list of arguments, and tells the actor which position it was in the argument list. The actor may decide to convert itself to a rock-bottom Lisp object for the occasion, or ask other arguments to convert themselves. A ROCK-BOTTOM message asks an actor to supply a rock-bottom Lisp object which can take the place of the actor when applying Lisp functions.

3.3 Actors accept messages asking them to identify themselves

Communication between an actor system and a person using the system is accomplished by messages which ask actors to display, or print themselves to the user. An actor usually responds to messages like PRINT with some representation which clearly identifies who the actor is, and prints the values of important components of the actor.

There's an interesting issue which arises concerning conventions for printing actors. Most conventional languages have just a few kinds of data, and the number of different data types is fixed in advance by the language. These languages can establish conventions for printing by defining a different printed representation for each kind of data object. These conventions are used to convert the object to a character string which can be printed out. The reader can create an object of the appropriate type by parsing the input string according to the conventions. In Lisp, parenthesized expressions denote lists, sequences of alphabetic characters are atomic symbols.

Since Act 1 allows the user to introduce new data types at any time by defining new actors, how can they be typed in and printed? Of course, each actor can have message handlers to print itself in a special way, but it is helpful to establish some conventions for printed representations that actors can fall back on.

Our solution is an extension of the printing philosophy of Lisp. In Lisp, the PRINT function is expected to produce a printed representation such that if that printed representation were read back in using the READ function, it would result in an object which is EQUAL to the original object. The fact that READ and PRINT are inverses is especially helpful to programs which need to convert back and forth between objects and their printed representations.

We have an UNREAD message which asks an actor to return a printed representation, suitable for reading back in and creating an actor equal to the original one. The printed representation must be in a form which can be printed on the user's screen with the printing primitives of the implementation language, in our case the Lisp PRINT function.

To be able to read and print arbitrary actors, we devise a way to interpose EVAL between READ and PRINT. EVAL is capable of constructing any actor whatsoever. The reader recognizes a special escape character which causes it to invoke EVAL on the following expression, and return that as the result of the read. Thus, any actor can be typed in by typing the escape character, followed by an expression which evaluates to the desired actor.

Our convention for PRINT then, is that an actor can print starting with the read-time EVAL character, followed by an expression which evaluates to an equivalent actor. If we have actors called TURTLEs, a plausible way to print them might be by printing out a call to a function which creates turtles, say CREATE-TURTLE, along with arguments which would create a turtle with the appropriate state components, such as POSITION, HEADING, PEN.

The ability to use EVAL in reading and printing supplies the needed flexibility for representing actors. Each actor responds to an UNEVAL message requesting that he return an expression which will evaluate to an actor equal to himself. For simple actors like symbols and lists, they will just return QUOTEd objects when sent UNEVAL, but for newly introduced data types, UNEVAL will produce a form which can perform an arbitrary computation. The UNEVAL message also carries an environment, like the EVAL message. This aids printing actors *relative* to the current environment, so that unnecessary detail can be omitted.

Actors can also be defined to respond to different varieties of PRINT messages, to print in different formats which may be more readable in a given context. The last-ditch heuristic for printing actors is just to show the user who the script, proxy and acquaintances are, since this is enough information to determine who the actor is.

3.4 Making decisions

Special care is needed in the treatment of *conditionals*. In conventional languages like Lisp, a conditional can just compare the result of a test to the TRUE and FALSE objects in the language to decide which branch of a conditional to execute. But if we want to adhere to our policy of allowing user-defined actors to appear *anywhere* a system-provided actor can appear, we must provide for the case where the result of a predicate in a conditional is a user-defined actor. The Act 1 interpreter must be prepared to send a message to the value of a predicate to decide how to proceed with a conditional. The IF message asks if a target considers himself to be TRUE for the purposes of making a choice between two branches of a conditional.

3.5 Previous work and acknowledgements

Act 1 is just part of a continuing research effort into the idea of object-oriented message passing systems. Many of the ideas in this paper represent a synthesis of ideas which first appeared in earlier research. We owe a great debt to past efforts in exploring these and related ideas.

Carl Hewitt originally developed the notion of an actor, and has guided the development of both the actor theory and implementation since its inception. Carl has lent much support and encouragement to our implementation effort. We owe

special thanks to those in our group who have worked on previous projects to implement actors, including Marilyn McLennan, Howie Shrobe, Todd Matson, Richard Steiger, Russell Atkinson, Brian Smith, Peter Bishop and Roger Hale.

Act 1's most distant ancestors are the languages Simula and Lisp. Simula was the first language which tried to support object-oriented programming. [25] Simula grafted the notion of objects onto an Algol-like base language, and provided for sharing knowledge among objects by organizing them into classes. Although traditional Lisp does not provide direct support for objects and messages, Lisp's flexibility and extensibility has allowed many in the AI community to experiment with programming styles which capture some of the actor philosophy [20], [21], [22]. Experience in construcing so-called *data-driven* programs in Lisp, and Lisp's uniform control structure of function calls have been valuable to us in designing Act 1 [23].

Alan Kay's *Smalltalk* replaced Simula's Algol base with a foundation completely built upon the notion of objects and messages [15], [16], [17]. Smalltalk is the closest system to ours in sharing our radical approach to building a totally object-oriented language. A major area where Smalltalk and Act 1 differ is in our proposals for parallelism. Smalltalk follows Simula in using coroutines to simulate parallelism. Smalltalk also retains the class mechanism of Simula rather than sharing knowledge by delegating messages as we do. The success of Xerox's Learning Research Group in using the object-oriented programming philosophy in a personal computer for children has been a great inspiration and encouragement to us.

Kenneth Kahn has developed an actor language called *Director*, as an extension to Lisp [11], [12], [13]. Director does not treat everything as an actor, and is quasi-parallel, but has developed extensive dynamic graphics facilities and a means of compiling actors to Lisp. Our mechanism for delegation was strongly influenced by Director. Director and Act 1 have developed concurrently, and throughout our work we have enjoyed our interaction with Ken, for providing much insight and important ideas.

Aspects of the actor philosophy have also been explored by others in the context of experimental extensions to Lisp. Guy Steele and Gerald Sussman implemented a dialect of Lisp called *Scheme*, which compromises between traditional Lisp and actors [26]. Active objects can be implemented as Lisp functions, and message passing performed by function call, but Scheme's built-in data types, (numbers, symbols, lists) are not active objects in the same sense as functions are. Sussman and Steele have contributed to understanding the issues of continuation control structure and compilation.

Daniel Friedman and David Wise have a modified Lisp interpreter which uses DELAYed control structure for the Lisp CONS primitive, and their FONS implements list structure with parallel evaluation of elements and synchronization as does our RACE [27], [28].

An interesting area of recent research applies the object-oriented philosophy and parallelism to systems which manipulate *descriptions* such as those of Luc Steels and Giuseppe Attardi [38], [37], [36].

We would like to extend thanks to Jonl White for help with MacLisp, in which Act 1 is implemented, and also to Richard Greenblatt and the MIT Lisp Machine Group.

We would like to thank Luc Steels, Carl Hewitt, Kenneth Kahn, Giuseppe Attardi, Maria Simi, William Kornfeld, Dan Halbert, David Taenzer for their helpful comments and suggestions on earlier drafts of this paper.

Bibliography

[1] Marvin Minsky and Seymour Papert, The Society Theory of Mind, MIT unpublished draft

[2] Henry Lieberman, Thinking About Lots Of Things At Once Without Getting Confused, AI memo 626, MIT AI Lab, 1980

[3] Carl Hewitt, Viewing Control Structures As Patterns Of Passing Messages, in "Artificial Intelligence, An MIT Perspective", Brown and Winston, eds.

[4] Carl Hewitt, et al., A Universal, Modular Actor Formalism For Artificial Intelligence, Third International Joint Conference on Artificial Intelligence, 1973

[5] Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, Specifying And Proving Properties Of Guardians For Distributed Systems, Conference on Semantics of Concurrent Computing, Evian, France 1979

[6] Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, Security And Modularity In Message Passing, First Conference on Distributed Computing, Huntsville, Ala. 1979

[7] The Incremental Garbage Collection Of Processes, Henry Baker and Carl Hewitt, Conference on AI and Programming Languages, Rochester, NY, Aug 1977

[8] Henry Baker, List Processing In Real Time On A Serial Computer, Communications of the ACM, April 1978

[9] Henry Baker, Actor Systems For Real Time Computation, MIT Lab for Computer Science report TR-197

[10] Carl Hewitt and Jeff Schiller, The Design of Apiary-0, 1980 Lisp Conference, Stanford University

[11] Kenneth Kahn, Director Guide, MIT AI lab memo

[12] Kenneth Kahn, How To Program A Society, AISB Conference Proceedings, 1980

[13] Kenneth Kahn, Creating Computer Animation From Story Descriptions, MIT AI Lab memo TR-540, 1980

[14] Kenneth Kahn, Dynamic Graphics Using Quasi-parallelism, ACM SigGraph 78 Conference

[15] Alan Kay, Microelectronics and the Personal Computer, Scientific American, September 1977

[16] Adele Goldberg, Dave Robson and Xerox PARC Learning Research Group, Smalltalk: Dreams and Schemes, Forthcoming, 1981

[17] Daniel Ingalls, The Smalltalk 76 Programming System: Design and Implementation, Fifth ACM Conference on Principles of Programming Languages, 1978

[18] Alan Kay and Adele Goldberg, Personal Dynamic Media, IEEE Computer, March 1977

[19] Alan Borning, ThingLab: An Constraint Oriented Simulation Laboratory, Artificial Intelligence journal, forthcoming 1981

[20] Jon Allen, Anatomy of Lisp, McGraw Hill 1979

[21] David Moon, MacLisp Reference Manual, MIT Lab for Computer Science report

[22] Daniel Weinreb, David Moon, Lisp Machine Manual, MIT Artificial Intelligence Lab report, 3rd edition 1981

[23] Erik Sandewall, Programming In An Interactive Enviroment: The Lisp Experience, Computing Surveys, vol. 10 no. 1

[24] Henry Lieberman and Carl Hewitt, A Real Time Garbage Collector That Can Recover Temporary Storage Quickly, AI Memo 569, MIT AI Lab, April 1980

[25] G. Birtwhistle, O. J. Dahl, B. Myrhaug, K. Nygaard, Simula Begin, Auerbach, 1973

[26] Guy Steele and Gerald Sussman, Revised Report On Scheme: A Dialect Of Lisp, AI memo 472, 1978

[27] Daniel Friedman and David Wise, Cons Should Not Eval Its Arguments, Indiana U. technical report 44

[28] Daniel Friedman and David Wise, A Nondeterministic Constructor for Applicative Programming, 1980 Conference on Principles of Programming Languages, Las Vegas

[29] Barbara Liskov, Alan Snyder, Russell Atkinson, Craig Schaffert, Abstraction Mechanisms in CLU, Communications of the ACM, August 1977

[30] Henry Lieberman, The TV Turtle, A Logo Graphics System For Raster Displays, ACM SigGraph and SigPlan Symposium on Graphics Languages, 1978

[31] Hal Abelson and Andy DiSessa, The Computer As A Medium For Exploring Mathematics, MIT Press, 1980

[32] Seymour Papert, Mindstorms, Basic Books, 1980

[33] Victor Lesser and Lee Erman, A Retrospective View of the Hearsay-II Architecture, Fifth International Joint Conference on Artificial Intelligence, 1977

[34] Hans Moravec, Intelligent Machines: How To Get From Here To There And What To Do Afterward, Stanford AI Lab memo, 1977

[35] C. A. R. Hoare, Monitors: An Operating System Structuring Concept, Communications of the ACM October 1975

[36] Daniel Bobrow and Terry Winograd, An Overview Of KRL, A Langugage For Knowledge Representation, Cognitive Science, vol. 1, no. 1, 3 1977

[37] Luc Steels, Representing Knowledge As A Society of Communicating Experts, AI memo TR-542 , MIT AI Lab, 1980

[38] Carl Hewitt, Giuseppe Attardi, and Maria Simi, Knowledge Embedding In The Descripion System Omega, 1980 AAAI Conference, Stanford University