In Object-Oriented Concurrent Programming, A. Yonezawa and M. Tokoro, eds., MIT Press, 1987.

Concurrent Object-Oriented Programming in Act 1

Henry Lieberman

To move forward to the next generation of artificial intelligence programs, new languages will have to be developed that meet the demanding requirements of these applications. Artificial intelligence software will require unprecedented flexibility, including the ability to support multiple representations of objects, and the ability to incrementally and transparently replace objects with new, upward-compatible versions. As advances in computer architecture and changing economics make feasible machines with large-scale parallelism, artificial intelligence will require new ways of thinking about computation that can exploit parallelism effectively.

To realize this, we propose a model of computation based on the notion of actors, active objects that communicate by message passing. Actors blur the conventional distinction between data and procedures. For parallel computation, actors called *futures* create concurrency, by dynamically allocating processing resources. Serializers restrict concurrency by constraining the order in which events take place, and have changeable local state. The actor philosophy is illustrated by a description of our prototype actor interpreter Act 1.

1. 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?

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 (or her) own functioning. We shall show how *Act I* 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. 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.

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 I* allows a high degree of parallelism. *Act I* uses the object-oriented, message passing philosophy to provide exceptionally clean mechanisms for exploiting parallelism while avoiding the pitfalls of timing errors. These ideas should be especially suited for implementation on a large integrated network of parallel processors such as the Apiary [Lieberman 1983].

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 behavior, and creating extensions of these actors with idiosyncratic behavior more suited to specific situations.

2. 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—an *actor*. Everything, including 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. [We will sometimes "anthropomorphize" actors by referring to "he" instead of "it". We could also say "she".]

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 behavior when he receives a message. 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 I* is implemented in Lisp, and we will show how we fool Lisp into regarding ordinary data and procedures as active objects.

3. Why do we insist that everything be an actor?

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.

Less radical systems like Simula-67 [Birtwistle et al. 1973], Clu [Liskov et al. 1977], C++, Lisp Machine Lisp [Moon and Weinreb 1984] 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 [Goldberg and Robson 1984] is another language which shares our philosophy of uniform representation of objects.)

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 behavior to a system without modifying the old system, providing the new behavior is compatible. In an actor system, the user may always add a new actor with the same message passing behavior, but perhaps with a different internal implementation, and the new actor will appear identical to the old one as far as an users are concerned. We can also extend a system by introducing a new actor whose behavior is a *superset* of the behavior 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 behavior 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 woefully storing elements of zeros and ones.

Create an actor called IDENTlTY-MATRlX.
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 database 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 behavior. 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. To do this, it must be possible for a user-defined data type to *masquerade* for a system data object like a number. 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.

4. An inventory of messages: EVAL and MATCH

There's no monolithic, centralized interpreter for *Act I* as there is for Lisp. *Act I* 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 interpreter is driven by messages which ask actors to evaluate themselves. Because we send *EVAL* messages rather than have an *EVAL* function as does Lisp, the code for responding to these messages is distributed throughout the system. 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*).

In place of Lisp's argument lists, an actor receiving a message has a *pattern* to which the incoming message is matched. 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, or 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 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*.

5. Equality is in the eye of the beholder

The fact that actors are defined only by their behavior in response to messages is important because it allows many different implementations of the same concept to co-exist in a single system. 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 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, not by bitwise comparison on their storage formats.

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 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.

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 I* with *ARE-YOU* messages. There are no data types in *Act I* 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 behavior, or compare him with other actors. 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?".*

6. Continuations implement the control structure of functions

The message sending primitive in *Act I* 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 I* 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 behavior necessary to continue a computation after the question is answered. A continuation is the actor analogue of a *return address* for subroutines [Abelson 1985] .

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 I* 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 on 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.

One use of continuations 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 behavior to be taken when an *error* condition is encountered. 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.

7. 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 behavior 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 behavior 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. 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. Delegation allows patterns of communication between objects sharing knowledge to be determined at the time a message is received by an object, rather than when the object is created, or its definition compiled.

8. Peaceful co-existence between actors and Lisp

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 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. 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! The ground rules are that the implementation is allowed to violate the actor model only when it is guaranteed to be invisible to the programmer.

The simulation of actors on conventional hardware incurs a certain cost, but the overhead must be kept down to a reasonable level, so that it is not prohibitive even for the simplest operations. 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 behavior.

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.

How does an actor interpreter perform some computation like adding two numbers? 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. 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. 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. Since there are only a fixed number of types in the implementation language and they are 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 fast 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.
y 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. 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 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. 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. Care must be taken to 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. A *ROCK-BOTTOM* message asks an actor to supply a Lisp object which can take the place of the actor when applying Lisp functions.

9. Actors accept messages asking them to identify themselves

Conventional languages have a fixed set of data types, and establish conventions about how data types are input and output for communication with human users. Since *Act 1* allows the user to introduce new data types at any time by defining new actors, we need conventions for how they can 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.

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 last-ditch heuristic for printing actors is just to show the user who the sit, proxy and acquaintances are, since this is usually enough information to identify the actor.

10. 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 I* 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.

11. Thinking about lots of things at once without getting confused

The next part of this paper will try to accomplish several goals (in parallel):

We will argue that the *actor* model is an appropriate way to think about parallel computation. Since many actors may be actively sending or receiving messages at the same time, actors are inherently well suited to modelling parallel systems.

We will present some specific actors which we feel should be included in the programmer's tool kit for writing parallel programs. We will show examples illustrating the use of these primitives. *Futures* are actors which represent the values computed by parallel processes. They can be created dynamically and disappear when they are no longer needed. Other actors may use the value of a future without concern for the feet that it was computed in parallel. Synchronization is provided by *serializers,* which protect actors with internal state from timing errors caused by interacting processes.

We will show how these primitives have been implemented in *Act 1. Act I* has been implemented on a serial machine, but it simulates the kind of parallelism that would occur on a real multiprocessor machine. Discussion of the implementation will give a more concrete picture of the mechanisms involved and will also show what would be needed for an implementation on a real network of parallel processors.

12. Traditional techniques for parallelism have been inadequate

Any language which allows parallelism must provide some way of *creating* and *destroying* parallel activities, and some means of *communicating* between them. Most of the traditional techniques for parallelism which have grown out of work in operating systems and simulation share these characteristics:

Usually, only *a fixed* number of parallel processes can be created, and processes cannot be created by programs as they are running. Processes usually must be *explicitly* destroyed when no longer needed. Communication between processes takes the form of assignment to memory cells shared between processes.

We propose that parallel processes be represented by actors celled futures [Baker and Hewitt 1977]. Futures can be created *dynamically* and disappear by *garbage collec*tion rather than explicit deletion when they're no longer needed. Communication between processes takes place using shared actors called *serializers,* which protect their internal state against timing errors.

13. Dynamic allocation of processes parallels dynamic allocation of storage

Act I solves the problem of allocating processes by extending Lisp's solution to the problem of allocating storage.

Languages like Fortran and machine languages take the position that storage is only allocated statically, in *advance* of the time when a program runs. The inflexibility of static storage allocation led Lisp to a different view. With Lisp's *CONS,* storage magically appears whenever you need it, and the garbage collector magically recovers storage when it's no longer accessible. Even though the computer only has a finite number of storage locations in reality, the user can *pretend* that memory is practically infinite.

Futures are actors which represent parallel computations. They can be created when needed, and when a future becomes inaccessible, it gets *garbage collected,* as any Lisp object does. The number of processes need not be bounded in advance, and if there are too many processes for the number of real physical processors you have on your computer system, they are automatically *time shared.* Thus the user can *pretend* that processor resources are practically infinite.

Fortran procedures sometimes communicate through assignment to shared variables. This causes problems because a memory location shared between several users can be inadvertently smashed by one user, violating assumptions made by other users about the memory's contents.

Lisp uses the control structure *of function calls and returns,* procedures communicating by passing arguments and resuming values. A process creating a future actor communicates with the future process by passing arguments, and the future process communicates with its creator by resuming a value. We discourage explicit deletion of processes for the same reason we discourage explicit deletion of storage. If two users are both expecting results computed by a single process, then if one user is allowed to destroy the process unexpectedly, it could wreak havoc for the other user.

14. Futures are actors representing the results of parallel computations

A future is like *a promise or I.O.U.* to deliver the value when it is needed. Act l's primitive *HURRY* always returns a future actor immediately, regardless of how long the computation will take. *HURRY* creates a *parallel process* to compute the value, which may still be running after the future is resumed. The user may pass the future around, or perform other computations, and these actions win be overlapped with the computation of the future's value.

From the viewpoint of a user program, the future actor is indistinguishable from the value itself. The only difference is that, on a parallel machine, it can be computed more quickly. The behavior of a future actor is arranged so that if computation of the value has been completed, the future will act identically to the value. If the future is still running, it will delay the sender long enough for the computation to run to completion.

Futures are especially useful when a problem can be broken up into *independent subgoals.* If the main problem requires several subgoals for its solution, and each goal can be pursued without waiting for others to finish, the solution to the problem can be found much faster by allocating futures to compute each subgoal. Since the computation of each subgoal will presumably take a long time, the computation of the subgoals will overlap with each other and with the procedure combining their results.

How do we know when the value that the future returns will really be needed by someone else? In the actor model, that's easy - the only way another actor may do anything with the value is to send it a message. So, whenever any other actor sends a message to a future, we require that the *future finish* computing before a reply can be sent. If the value is requested before the future is ready, the caller must *wait* for the future to finish before getting the answer. When the future does finish, it stashes the answer away inside itself, and thereafter behaves identically to the answer, passing all incoming messages through to the answer.

A nice aspect of using futures is that the future construct automatically matches creation of parallel processes and synchronization of results computed by the processes. This promotes more structured parallel programs than formalisms in which you describe the creation of parallel processes and their synchronization independently.

The property of being able to transparently substitute for any actor whatsoever a future computing that actor is crucially dependent on the fact that in *Act 1*, everything is an object and all communication happens by message passing. This can't be done in less radical languages like Clu and Simula, which do have some provision for objects and message passing, but which don't treat everything that way. A future whose value is a built-in data type like numbers or vectors could not be used in a place where an ordinary number or vector would appear.

Futures can be used in *Act I* in conjunction with Lisp-like list structure, to represent *generator* processes. Suppose we have a procedure that produces a sequence of possibilities, and another procedure that consumes them, and we would like to overlap the production of new possibilities with testing of the ones already present.

We can represent this by having the producer come up with a list of possibilities, and the consumer may pick these off one by one and test them. This would work fine if there were a finite number of possibilities, and if the consumer is willing to wait until all possibilities are present before trying them out. But with futures, we can simply change the producer to create futures for the list of possibilities, creating a list which is *growing* in time, while the possibilities are being consumed.

Define PRODUCER:
if there are no more POSSIBILITIES.
return the EMPTY-LIST.
If some possibilities remain,
Create a list whose FIRST is:
a FUTURE computing the FIRST-POSSIBILITY,
and whose REST is:
a FUTURE computing the rest of the possibilities
by calling PRODUCER

Define CONSUMER, consuming a list of POSSIBILITIES:
Test the FIRST possibility on the POSSIBILITIES list,
Then call the CONSUMER
on the REST of the POSSIBILITIES list.

The consumer can use the list of possibilities as an ordinary list, as if the producer had produced the entire list of possibilities in advance. We could get even more parallelism out of this by having the consumer create futures, testing an the possibilities in parallel.

On a machine with sufficiently many processors, the most radical way to introduce parallelism would be to change the interpreter to *evaluate arguments in parallel.* Making this *eager beaver* evaluator would require just a simple change to create a future for the evaluation of each argument to a function. In our current implementation of *Act 1,* we require explicit specification of futures because processes are still a bit too expensive on our serial machine to make it the default to create them so frequently.

Futures are more powerful than the alternative *data flow* model proposed by Dennis [Dennis 1979]. In the dataflow model, arguments to a function are computed in parallel, a function is applied only when all the arguments have finished returning values. Let's say we're trying to compute the *SUM* of *FACTORIAL* of *10, FACTORIAL* of *20* and *FACTORIAL* of *30,* each of which is time consuming. In dataflow, the computations of the factorial function can all be done in parallel, but *SUM* can't start computing until *all* the factorial computations finish.

If futures are created for the arguments to a function, as can be done in *Act 1,* the evaluation of arguments returns immediately with future actors. The function is applied, with the future actors as arguments, without waiting for any of them to run to completion. It is only when the value of a particular argument is actually needed that the computation must wait for the future to finish.

In our sum-of-factorials example, imagine that the first two factorials have finished but the third has not yet returned. *Act I* allows *SUM* to begin adding the results of *FACTORIAL* of *10* and *FACTORIAL* of *20* as soon as they both return, in parallel with the computation of *FACTORIAL* of *30.*

15. Explicit deletion of processes considered harmful

Notice that there are some operations on futures that we *don't* provide, although they sometimes appear in other parallel formalisms. There's no way to ask a future whether he has finished yet. Such a message would violate the property of futures that any incoming message forces him to finish, and that wrapping futures around values is completely transparent. It would encourage writing time and speed-dependent programs.

There's no way to *stop* a future before the future has returned a value. Continuing the analogy with list storage, we believe that explicitly stopping or destroying processes is bad for the same reason that deleting pointers to lists would be bad in Lisp. Deleting a process that somebody else has a pointer to is just as harmful as deleting a list pointer that is shared by somebody else. It's much safer to let deletion happen by garbage collection, where the system can automatically delete an object when it can be verified that it's no longer needed by anybody.

We don't exclude the possibility of providing such lower level operations as examining and destroying processes for the purposes of debugging and implementation, but they should not be routinely used in user programs. A safer way of being able to make a decision such as which of two processes finished first is to use *Act l's* serializer primitives which assure proper synchronization.

16. How does *Act 1* implement futures?

When the future receives a message intended for its value, there are two cases, depending on whether the computation is still running or not. The future needs a flag to distinguish these cases. If it is running, the sender must wait until the computation finishes. When the future finishes, it sets the flag, and remembers the answer to the computation in a memory cell. Any messages sent to the future are then relayed to the stored answer.

Define HURRY, creating a FUTURE evaluating a FORM:
Create a CELL which initially says that the future is RUNNING.
Create a CELL for the ANSWER
to which the form will eventually evaluate.
Create a PROCESS, and start it to work
computing the value of the FORM,
Then send the VALUE of the FORM
to the continuation FINISH-FUTURE.

If I'm a FUTURE, and I get a request:
I check my RUNNING cell.
If it's TRUE, the sender waits until it becomes FALSE.
Then, I pass the message along to my ANSWER cell.

Define FINISH-FUTURE, receiving a VALUE for the FORM:
I receive the ANSWER to the computation started by HURRY.
Update the RUNNING cell to FALSE,

indicating the future has finished.
Put the VALUE in the FUTURE's ANSWER cell.
Cause the process to commit SUICIDE, since it is no longer needed.

17. Aren't futures going to be terribly inefficient?

Advocates of more conservative approaches to parallelism might criticize our proposals on the grounds that futures are much too *inefficient* to implement in practice. Allocating processes dynamically and garbage collecting them does have a cost over simpler schemes, at least on machines as they are presently designed. Again, we make the analogy with list structure, where experience has shown that the benefits of dynamic storage allocation are well worth the cost of garbage collection.

Trends in hardware design are moving towards designs for computers that have many, small processors rather than a single, large one. We think the challenge in designing languages for the machines of the near future will come in trying to make effective use of massive parallelism, rather than in being excessively clever to conserve processor resources.

One source of wasted processor time comes from processes that are still running using up processor time even though they are no longer needed, before they are reclaimed by the garbage collector. This is analogous to the fact that in Lisp, storage is sometimes unavailable between the time that it becomes inaccessible and the time it is reclaimed by the garbage collector. This is a cost that can be minimized by a smart *incremental* real time garbage collector for processes like that of Henry Baker, or the one we proposed in [Lieberman and Hewitt 1983].

We intend that processes be cheap and easy to create, a basic operation of the system just like message passing. We have taken care to see that a process doesn't have a tremendous amount of state information or machinery associated with it. The state of a process is *completely* described by the *target* actor, the *message* actor, and the *continuation* actors. If this information is saved, the process can be safely interrupted or may wait for some condition to happen while other processes are running, and be resumed later.

This makes processes more *mobile.* It is easy to move a process from one processor to another, or time many processes on a single processor. Multiple processor systems may need to do dynamic load balancing or time sharing when there are more conceptual processes than physical processors.

18. Serializers are needed to protect the state of changeable actors

There's a whole class of errors which arise in parallel programming which don't show up in sequential programming: *timing errors.* Timing errors occur when one process looks at the state of an actor, takes some action on the implicit assumption that the state remains unchanged, and meanwhile, another process modifies the state, invalidating the data. Timing errors are possible when parallel programs use *changeable* actors incorrectly. An actor is changeable if the same message can be sent to him on two different occasions and result in different answers.

To protect against misuse of actors with changeable state, we will not let actors modify others directly. Instead, actors with changeable state must be sent messages *requesting* them to make a state change. An actor who receives state change requests should insure that each state change operation completes without outside interference before another request is handled. This facility is provided by an actor called *ONE-AT A-TIME.*

ONE-AT-A-TIME is a kind of *serializes,* an actor which *restricts* parallelism by forcing certain events to happen serially. *ONE-AT-A-TIME* creates new actors which are protected so that only one process may use the actor at a time. A *ONE-AT-A-TIME* actor holds his state in a set of *state variables* which are defined locally to himself and are inaccessible from outside. He also has a script for receiving messages, and as a result of receiving a message, he may decide to change his state. When a message is received, he becomes *locked* until the message is handled and possibly, the state is changed, then he becomes *unlocked* to receive another message.

The *ONE-AT-A-TIME* actor embodies the same basic concept as Hoare's *monitor* idea [Hoare 1975]. *ONE-AT-A-TIME* has the advantage that it allows creating protected *actors dynamically* rather than protecting a lexically scoped block of procedures and variables. The actors created by *ONE-AT-A-TIME* are *first-class citizens.* They may be created interactively at any time by a program, passed around as arguments, or returned as values, in a manner identical to that of any actor.

19. GUARDIANs can do more complex synchronization than ONE-AT-A-TIME

There are some kinds of synchronization which are not possible to achieve simply with *ONE-AT-A-TIME. ONE-AT-A-TIME* has the property that a reply must be given to an incoming message before possession is released and another message from a different process can be accepted. Sometimes a bit more control over *when* the reply is sent can be useful. The reply might sometimes have to be delayed until a message from another process gives it the go-ahead signal. In response to a message, it might be desired to cause a state change, release possession and await other messages, replying at some later time.

Imagine a *computer dating service,* which receives requests in parallel from many customers. Each customer sends a message to the dating service indicating what kind of person he or she is looking for, and should get a reply from the dating service with the name of his or her ideal mate. The dating service maintains a file of people, and matches up people according to their interests. Sometimes, the dating service win be able to fill a request immediately, matching the new request with one already on file. But if not, that request win join the file, perhaps to be filled by a subsequent customer.

The dating service is represented by an actor with the file of people as part of its state. If an incoming request can't be filled right away, the file of people is updated. The possession of the dating service actor must be released so that it can receive new customers, but we can't reply yet to the original customer because we don't know who his or her ideal mate is going to be!

The actor *GUARDIAN* provides this further dimension of control over how synchronization between processes takes place. When a message cannot be replied to immediately, the target actor will save away the means necessary to reply, continue receiving more messages, and perform the reply himself when conditions are right.

To do this, *GUARDIAN* makes use of the actor notion of *continuations.* Since the continuation encodes everything necessary to continue the computation after the reply, the *GUARDIAN* can remember the continuation, and reply to it later. *GUARDIAN* is like *ONE-AT-A-TIME,* but the continuation in messages sent to him is made explicit, to give the *GUARDIAN* more control over when a reply to that continuation may occur. *ONE AT-A-TIME* can be easily implemented in terms of *GUARDIAN.*

Here's the code for the computer dating service.

Define COMPUTER-DATING-SERVICE:
Create a GUARDIAN actor,
whose internal state variable is a FILE-OF-PEOPLE.

If I'm a COMPUTER-DATING-SERVICE and
I get a message from a LONELY-HEART
with a QUESTIONNAIRE to help find an IDEAL-MATE:
Check to see if anyone in the FILE-OF-PEOPLE
matches the QUESTIONNAIRE.
If there is, reply to the LONELY-HEART
the name of his or her IDEAL-MATE,
and reply to the IDEAL-MATE the name of the LONELY-HEART.
Otherwise, enter the LONELY-HEART in the FILE-OF-PEOPLE,

And wait for a request from another LONELY-HEART.

What really happens in the implementation if more than one process attempts to use a *GUARDIAN* actor at once? Each such actor has a *waiting line* associated with him. Messages coming in must line up and wait their turn, *in first-come-first-served* order. If the *GUARDIAN* is not immediately available, the process that sent the message must wait, going to sleep until his turn in line comes up, and the *GUARDIAN* becomes unlocked. Then, the message is sent on through, and the sender has no way of knowing that his message was delayed. Each message may change the internal state of the actor encased by the guardian.

Define GUARDIAN, protecting a RESOURCE,
which has its own internal state:
Each GUARDIAN has a WAITING-LINE, and
A memory cell saying whether the RESOURCE is LOCKED,
initially FALSE.

If I'm a GUARDIAN, and 1 get a MESSAGE,
If I'm LOCKED, the sender must wait on the WAITING-LINE
until I'm not LOCKED and
the sender is at the front of the WAITING-LINE.
Set the LOCKED flag to TRUE.
Send the RESOURCE the incoming MESSAGE,
Along with the REPLY-TO continuation of the MESSAGE,
so the RESOURCE can send a reply for the MESSAGE.
The RESOURCE might update his internal STATE
as a result of the MESSAGE.
Then, set the LOCKED flag to FALSE,
letting in the next process on the WAITING-LINE.

20. Waiting rooms have advantages over busy waiting

Several situations in implementing the parallel facilities of *Act I* require a process to *wait* for some condition to become true. If a message is received by a future before it finishes computing, the sender must wait for the computation to finish. If a message is received by a guardian while it is locked, the sender must wait until the guardian becomes unlocked, and the sender's turn on the queue arrives. If a message is received by some actor which provides *input* from some device like a terminal or disk, the input requested may not be available, so the requesting process must *wait* until such time as the input appears.

One way to implement this behavior is by *busy waiting,* repeatedly testing a condition until it becomes true. Busy waiting is a bad idea because it is subject to needless *deadlock.* If the condition becomes true for a while, then false again, it's possible the condition won't be checked during the time it is true. Since one process which is waiting might depend upon another's action to release it, failing to detect the release condition for one process can cause a whole system containing many processes to grind to a halt. Another disadvantage of busy waiting is that repeated checking of conditions wastes time.

A preferable technique for implementing wait operations is to have *waiting rooms.* When too many people try to visit the dentist at the same time, they must amuse themselves sitting in a waiting room listening to muzak and reading magazines until the dentist is ready to see them, then they can proceed. Except for the time delay caused by the interlude in the waiting room, their interaction with the dentist is unaffected by the fact that the dentist was busy at first.

Our *waiting rooms* are lists of processes which are waiting for some condition to happen. When a message is sent to some actor who wants to cause the sender to wait, he places the sending process in a waiting room, including the sender's continuation, which contains all information necessary to reply to the sender. When the condition becomes true, everybody waiting for that condition in a waiting room receives a reply.

Waiting rooms do introduce a problem with garbage collection, however. In order for an actor with waiting room to wake up a waiting process, it must *know about* (have a pointer to) that process. If the only reason a process is held onto is that it has requested something which isn't immediately available, that process is really no longer relevant. But the waiving room pointer protects the process from garbage collection. Waiting rooms should be implemented using *weak pointers,* a special kind of pointer which doesn’t protect its contents from garbage collection [Lieberman and Hewitt 1983].

21. RACE is a parallel generalization of Lisp's list structure

For creating an ordered sequence of objects, Lisp uses the elegant concept of *list structure,* created by the primitive *CONS.* For creating sequences of objects which are *computed in parallel,* and which are ordered by the time of their completion, we introduce an actor called *RACE.* *RACE* is a convenient way of collecting the results of several parallel computations so that each result is available as soon as possible.

CONS produces lists containing elements in the order in which they were given to *CONS.* *RACE* starts up futures computing all of the elements in parallel. It returns a list which contains the values of the elements *in the order in which they finished computing.* Since lists constructed by *RACE* respond to the same messages as those produced by *CONS,* they are *indistinguishable* from ordinary serial lists as far as any program which uses them is concerned.

If we ask for the *FIRST* or *REST (CAR* or *CDR)* of the list *befor e* it's been determined who has won that race, the process that sent the message *waits* until the outcome of the race becomes known, then gets the answer. This is just like what happens if we send a message to a future before the future has finished computing a value. (The *RACE* idea is similar to *the ferns* of Friedman and Wise [Friedman and Wise 1980].)

Using *RACE,* we can easily implement a parallel version of *OR,* helpful if we want to start up several heuristics solving a problem, and accept the result of the first heuristic to succeed in solving the problem. *PARALLEL-OR* starts evaluating each of a set of expressions given to it, in parallel, and returns the first one which evaluates *TRUE,* or returns *FALSE* if none of the expressions are *TRUE.*

We just create a *RACE* list, whose elements are the evaluated disjunct expressions. This searches all the disjuncts concurrently, and returns a list of the results. A simple, serial procedure runs down the list, resuming the first *TRUE* result, or *FALSE* if we get to the end of the list without finding a true result. All evaluations which might still be going on after a *TRUE* result appears become inaccessible, and those processes can be garbage collected. We don't need to explicitly *stop* the rest of the disjuncts.

Define PARALLEL-OR of a set of DISJUNCTS:
Start up a RACE list evaluating them in parallel
using EVALUATE-DISJUNCTS.
Examine the results which come back
using EXAMINE-DISJUNCTS.

Define EVALUATE-DISJUNCTS, of a set of DISJUNCTS:
If the set is empty, return the empty list.
Otherwise, break up the disjuncts
into FIRST-DISJUNCTS and REST-DISJUNCTS.

Start up a RACE between
testing whether FIRST-DISJUNCT evaluates to TRUE, and
doing EVALUATE-DISJUNCTS on the REST-DISJUNCTS list.

Define EXAMINE-DISJUNCTS, of a list of clauses:
If the list is empty, return FALSE.
If not, split it into FIRST-RESULT and REST-RESULTS.
If REST-RESULT is true,
return it as the value of the PARALLEL-OR.
Otherwise, do EXAMINE-DISJUNCTS on REST-RESULTS

As another illustration of the use of *RACE,* consider the problem of *merging* a sequence of results computed by parallel processes. If we have two lists constructed by *RACE* whose cements appear in parallel, we can merge them to form a list containing the elements of both lists in their order of completion. The first element of the merged list appears as the first of a *RACE* between the first elements of each list.

Define MERGE-LISTS of ONE-LIST and ANOTHER-LIST:
Return a RACE between
The first element of ONE-LIST, and
the result of MERGE-LISTS on ANOTHER-LIST
and the rest of ONE-LIST.

The implementation of *RACE* is a bit tricky, motivated by trying to keep *RACE* analogous to *CONS*. *RACE* immediately starts up two futures, for computing the *FIRST* and *REST*. The outcome of *RACE* depends upon which future finishes first. We use a serializer to receive the results of the two futures and deliver the one that crosses the finish line first.

First, let's look at the easy part, when the future for the *FIRST* element finishes before the future for the *REST* finishes. At that time, the race actor can reply to *FIRST* and *REST* messages sent to him. When he gets a *FIRST* message, he should reply with the value of the first argument to *RACE,* the actor that won the race. When he gets a *REST* message, he can reply with *the future computing the second argument ho RACE,* even though that future may *still be running.*

The more difficult case is when the *REST* future finishes first. The trivial case occurs when the *REST* future returns *THE-EMPTY-LIST.* A *RACE* whose *REST* is empty should produce a list of one element just like a *CONS* whose rest is empty.

If the value of the *REST* future isn't empty, then we can assume he's a list, and the trick is to get him to work in the case where he's a list produced by *RACE. So we* ask that list for *his* first element. If the list was produced by *RACE,* this delivers the fastest element among those that he contains. This element then becomes the first element of the *RACE* list node that's being constructed. The *FIRST* future, still running, must *RACE* against the remaining elements of the list to produce the *REST* of the final *RACE* list. When the computation of an element which appears farther down in the list outraces one which appears closer to the front of the list, he *percolates* to the front, by *changing places* with elements that are still being computed.

Define RACE, of a FIRST-FORM and REST-FORM: Create
futures evaluating FIRST-FORM and REST-FORM. Return
whichever of these FINISHES-FIRST: Either the
FIRST-FUTURE-WINS, or the REST-FUTURE-WINS.

Define FIRST-FUTURE-WINS:
If the FIRST-FUTURE finishes first,
Return a list whose first is the value of the FIRST-FUTURE
and whose rest is REST-FUTURE,

which may still be running.

Deject REST-FUTURE-WINS, helping RACE:
If the REST-FUTURE finishes before FIRST-FUTURE,
Then, look at the value of the REST-FUTURE.
If it's empty, wait until the FIRST-FUTURE returns,
and return a list of FIRST-FUTURE.
If it's not empty,
Return a list whose first is
the first of the list returned by REST-FUTURE,
and whose rest is
a RACE between
The value returned by FIRST-FUTURE, and
The rest of the list returned by REST-FUTURE.

22. Previous work and acknowledgments

Carl Hewitt originally developed the notion of an actor, and has guided the development of both the actor theory and implementation since its inception. 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 explicitly support object-oriented programming [Birtwistle et al. 1973]. 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 [Moon 1981], [Moon et al. 1984].

Alan Kay's *Smalltalk* replaced Simula's Algol base with a foundation completely built upon the notion of objects and messages [Goldberg and Robson 1984], *Smalltalk* is the closest system to ours in sharing our radical approach to building a totally object oriented language. 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.

Kenneth Kahn has developed an actor language called *Director,* as an extension to Lisp [Kahn 1980], [Kahn 1979]. 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.

Guy Steele and Gerald Sussman implemented a dialect of Lisp called *Scheme,* which compromises between traditional Lisp and actors [Abelson et al., 1985]. 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 *FOILS* implements list structure with parallel evaluation of elements and synchronization as does our RACE. [Freidman and Wise 1979], [Friedman and Wise 1980].

We would like to extend thanks to Jon White and Richard Greenblatt for helpful discussion of critical systems programming issues.

We would like to thank Giuseppe Attardi, Maria Simi, Luc Steels, Kenneth Kahn, Carl Hewitt, William Kornfeld, Daniel Friedman, David Wise, Dave Robson, Gerald Barber, Dan Halbert, David Taenzer, and Akinori Yonezawa for their helpful comments and suggestion on earlier drafts of this paper.

References

[Abelson et al. 1985] Abelson, H., et al., *Revised Revised Report On Scheme,* Technical Report 848, MIT Artificial Intelligence Laboratory, 1985.

[Abelson and DiSessa 1980] Abelson, H., A. DiSessa, The Computer as a Medium For Exploring Mathematics, MIT Press, Cambridge, MA, 1980.

[Baker 1977] Baker, H., Actor *Systems For Real Time Computation,* Technical Report 197, MIT Laboratory for Computer Science, 1977.

[Baker 1978] Baker, H., *List Processing in Real Time on a Serial Computer,* Communications of the ACM, April 1978.

[Baker and Hewitt 1977] Baker, H., C. Hewitt, *The Incremental Garbage Collection of Processes,* Proc. of Conference on AI and Programming Languages, Rochester, NY, 1977.

[Birtwistle et al. 1973] Birtwistle, G., O-J Dahl, B. Myhrhaug, K Nygeard, Simula Begin, Van Nostrand Reinhold, New York, 1973.

[Borning 1977] Borning, A., *ThingLab* -- An *Object-Oriented System for Building Simulations Using Constraints,* Proc. of UCAI, 1977.

[Dennis 1979] Dennis, J., *The Varieties of Data Flow Computers,* Proc. of 1st International Conference on Distributed Computing, Huntsville, Alabama, 1979.

[Friedman and Wise 1979] Friedman, D., D. Wise, *Cons Should Not Eval Its Arguments,* Technical Report 44, Indiana University, 1979.

[Friedman and Wise 1980] Friedman, D., D. Wise, A *Nondeterministic Constructor for Applicative Programming,* Proc. of Conference on Principles of Programming Languages, ACM SIGPLAN, 1980.

[Goldberg and Robson 1984] Goldberg, A., D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA, 1984.

[Hewitt et al. 1973] Hewitt, C., et al., *A Universal, Modular Actor Formalism for Artificial Intelligence,* Proc. of UCAI, 1973.

[Hewitt 1979] Hewitt, C., *Viewing Control Structures as Patterns of Passing Messages,* Artificial Intelligence, an MIT Perspective, Winston, P., R. Brown, (eds.), MIT Press, Cambridge, MA, 1979.

[Hewitt et al. 1979] Hewitt, C., G. Attardi, H. Lieberman, *Security And Modularity In Message Passing,* Proc, of 1st Conference on Distributed Computing, Huntsville, Alabama, 1979.

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

[Kahn 1978] Kahn, K., *Dynamic Graphics Using Quasi-Parallelism,* Proc. of ACM SIGGRAPH Conference, Atlanta, 1978.

[Kahn 1979] Kahn, K,, *Creation of Computer Animation from Story Descriptions,* 1979.

[Kahn 1980] Kahn, K. *How to Program a Society.* Proc, of the AISB Conference, 1980.

[Krasner 1984] Krasner, G. (ed),Smalltalk-80: Bits of History and Words of Advice, Addison-Wesley, Reading, MA, 1984.

[Lesser and Erman 1977] Lesser, V., L. Erman, *A Retrospective View of the Hearsay-II Architecture,* Proc. Of UCAI, 1977.

[Lieberman 1976] Lieberman, H" *The TV Turtle, A Logo Graphics System for Raster Displays,* Proc. of ACM SIGGRAPH and SIGPLAN Symposium on Graphics Languages, Miami, 1976.

[Lieberman 1983] Lieberman, H., *An Object Oriented Simulator for the Apiary,* Proc. of AAAI, Washington, D. C., August, 1983.

[Lieberman and Hewitt 1983] Lieberman, H., C. Hewitt, *A Real Time Garbage Collector Based on the Life" times of Objects,* Communications of the ACM, Vol. 26, No. 6, June 1983.

[Liskov et al. 1977] Liskov, B., A. Snyder, R. Atkinson, JS. Schaffert, *Abstraction Mechanism in CLU* Communications of the ACM, Vol. 20, No. 8, August 1977.

[Minsky 1986] Minsky, M., The Society of Mind Basic Books, New York, 1986

[Moon 1981] Moon, D., *MacLisp Reference Manual,* MIT Laboratory for Computer Science, 1981.

[Moon et al. 1984] Moon, D, D.Weinreb, et al., Lisp Machine Manual, Symbolics, Inc. and MIT, 1984.

[Papers 1981] Papert, S., Mindstorms, Basic Books, New York, 1981.

[Steels 1979] Steels, L., *Reasoning Modeled as a Society of Communicating Experts,* Technical Report 541 MIT Artificial Intelligence Laboratory, June 1979.