MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

A.I. MEMO 436A July 1977

ACTORS and CONTINUOUS FUNCTIONALS

by

Carl Hewitt and Henry Baker

This report describes research conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Office of Naval Research of the Department of Defense under Contract N00014-75-C-0522.

Actors and Continuous Functionals

Carl Hewitt and Henry Baker
M.I.T. Artificial Intelligence Laboratory
Cambridge, Mass. 02139
(617) 253-5873

SECTION I --- ABSTRACT

This paper presents precise versions of some "laws" that must be satisfied by computations involving communicating parallel processes. The laws take the form of stating plausible restrictions on the histories of computations that are physically realizable. The laws are very general in that they are obeyed by parallel processes executing on a time varying number of distributed physical processors. For example, some of the processors might be in orbiting satellites. The laws are justified by appeal to physical intuition and are to be regarded as falsifiable assertions about the kinds of computations that occur in nature rather than as proved theorems in mathematics. The laws are intended to be used to analyze the mechanisms by which multiple processes can communicate to work effectively together to solve difficult problems.

The laws presented in this paper are intended to be applied to the design and analysis of systems consisting of large numbers of physical processors. The development of such systems is becoming economical because of rapid progress in the development of large scale integrated circuits.

We generalize the usual notion of the history of a computation as a sequence of events to the notion of a partial order of events. Partial orders of events seem better suited to expressing the causality involved in parallel computations than totally ordered sequences of events obtained by "considering all shuffles" of the elementary steps of the various parallel processes [21,22]. The utility of partial orders is demonstrated by using them to express our laws for distributed computation. These laws in turn can be used to prove the usual induction rules for proving properties of procedures. They can also be used to derive the continuity criterion for graphs of functions studied in the Scott-Strachey model of computation. The graph of a function is simply the set of all input output pairs for the function. We can prove that the graph of any physically realizable procedure p that behaves like a mathematical function is the limit of a continuous functional F such that

$$\text{graph}(p) = \bigcup_{i \in N} F^i(\{\})$$

In other words the graph of p is the limit of the n-fold compositions of F with itself beginning with the empty graph.

SECTION II --- INTRODUCTION

In programming languages such as SIMULA-67 [17], SMALLTALK [18], and CLU [20], the emphasis has changed (compared to Algol-60) from that of procedures acting on passive data to that of active data processing messages. The actor model is a formalization of these ideas that is independent of any particular programming language. Instances of SIMULA and SMALLTALK classes and CLU clusters are actors. However, actors have been designed to include the added effects of parallelism so that instances of monitors[42,41], envelopes[43], and serializers[34] are also actors.

The actor message passing theory can be used to model networks of communicating processes which may be as close together as on the same LSI chip or as far apart as on different planets. It can be used to model processes which communicate via shared memory[12], packet-switched networks[13,24], ring-networks[23], boolean n-cube networks[44], or Batcher sorting nets[25].

SECTION III --- ACTORS and EVENTS

The theory presented in this paper attempts to characterize the behavior of procedural objects called actors [active objects] in parallel processing systems. Actors and events are the fundamental concepts in the theory. Actors interact with each other through one actor sending a messenger to another actor called the target. The arrival of a messenger at a target is an event, and these events are the basic steps in this model of computation. A key point in the actor model of computation is that messengers are themselves actors. The actor model is therefore an un-typed theory which is a generalization of the λ-calculus of Church.

Actors can be created by another actor as part of the second actor's behavior. Indeed, almost every messenger is newly created before being sent to a target actor.

Events mark the steps in actor computations; they are the fundamental interactions of actor theory. Each event is instantaneous and indivisible taking no duration in time. Every event E consists of the arrival of a messenger actor, called **messenger**(E), at a target actor, called **target**(E).

We will often use the notation:

E: [T <~~ M]

to indicate that E has messenger M and target T.

The time of an event is the arrival of the messenger of the event rather than the sending of the messenger because a messenger cannot affect the behavior of another actor until that actor receives it. If the sender wishes a reply, an actor (called the continuation) to whom any reply should be sent should also be carried by (as a component of) the messenger.

Intuitively, the arrival of the messenger M at the target T makes M's information available to the target

for the purpose of activating additional events. The arrival of M at T does not in itself cause any change to either M or T.

For each event E we define $\text{acquaintances}_E(T)$ and $\text{acquaintances}_E(M)$ to be the vector of immediate acquaintances of T and M, respectively. The immediate acquaintances of an actor x are the other actors x directly "knows about" at a given instant. The relation is asymmetric in the sense that it is possible for an actor x to know about an actor y without it being the case that y knows about x. An actor may or may not "know about" itself; if it does, it can directly send itself messages!

Law of Finite Acquaintances: For all actors x and events E such that x is the target or messenger of E, the vector $\text{acquaintances}_E(x)$ has finite length.

The above law states that an object can only be directly connected to finitely many other objects.

All of the actors which are definable within the lambda calculus of Church have the property that their acquaintances cannot change with time; i.e. if x is defined by a lambda expression, then for all events E_1 and E_2 in which x is the target or messenger, it will be the case that

$$\text{acquaintances}_{E1}(x) = \text{acquaintances}_{E2}(x)$$

In order to implement interprocess communication between parallel processors it is necessary to use actors whose vector of acquaintances changes over time. The purpose of this paper is to axiomatize the fundamental laws which govern the behavior of such actors.

An important example of an actor whose immediate acquaintances change with time is a cell. A cell is an actor which at any given time has exactly one immediate acquaintance--its contents. When the cell is sent a messenger which consists of the message, "what is your contents?", and a continuation--another actor which will receive the contents--the cell is guaranteed to deliver its contents to that continuation (while also continuing to remember them). All this might be very boring if the contents of the cell were constant. However, upon arrival of a messenger which has the message "update your contents to be x" and a continuation, the cell is guaranteed to update its contents to be the actor x (whatever that may be) and inform the continuation that the update has been performed. The behavior of cells will be axiomatized later in this paper after we have presented enough of the actor model to make this possible.

The target(E) and the messenger(E) and their immediate acquaintances will be called (immediate) participants of an event E. The immediate participants of an event are exactly those actors which can be accessed without sending any messages.

$$\text{participants}(E) \equiv \{\text{target}(E), \text{messenger}(E)\} \cup \text{acquaintances}_E(\text{target}(E)) \cup \text{acquaintances}_E(\text{messenger}(E))$$

Finite Interaction Law: For each event E, the immediate participants in E are finite.

The above law, which is intended to capture the physical intuition that only finitely many objects can interact in a single event, is an immediate corollary of the Law of Finite Acquaintances.

SECTION IV --- PARTIAL ORDERINGS on EVENTS

In order to develop a useful model of parallel computation, we have found it desirable to generalize the usual notion of the history of a computation as a sequence of events. In this paper a history of a computation will be expressed as a partial order which records the causal and incidental relations between events. The partial orders constrain the maximum amount of parallelism that can be used in an implementation. Any two events which are unordered can be executed concurrently using separate processors. However, there is no requirement that an implementation do this. Events can be executed in any time sequence that is consistent with the partial order.

IV.1 -- ACTIVATION ORDERING

One important strict partial ordering on events in the history of a computation is derived from how events activate one another. Suppose an actor x_1 receives a messenger m_1 in an event E_1 and as a result sends a messenger m_2 to another actor x_2. Then the event E_2, which is the arrival of the messenger m_2 at x_2, is said to be activated by E_1. We call the transitive closure of this "activates" relation the activation ordering and if E_1 precedes E_2 in this ordering then we write:

$$E_1 \text{ -}act\text{-> } E_2$$

In general -*act*-> is only a partial ordering because an event E might activate several distinct events $E_1,\ldots,E_n$, thereby causing a "fork".

IV.1.a -- Primitive Actors

A simple example which illustrates the use of -*act*-> is a computation in which integers 3 and 4 are added to produce 7. We suppose the existence of a primitive actor called + which takes in pairs of numbers and produces the sum. In this case + receives a messenger of the following form:

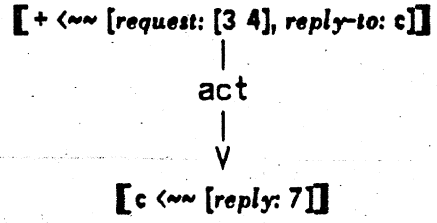
[*request:* [3 4], *reply-to:* c]

which specifies that the message in the request is the argument tuple [3 4] and the reply which is the sum should be sent to the continuation c when it has been computed. Thus the history of the computation contains two events:

1: a request event with target + and messenger that specifies the numbers to be added and an actor c to which the sum should be sent;

2: a reply event with target c and messenger that specifies the sum of the numbers.

These two events are related as follows in the activation ordering:



The activation ordering can be used to define the notion of a simple primitive actor as follows:

Definition: An actor x will be said to be a simple[1] primitive actor if whenever an event E_1 of the form

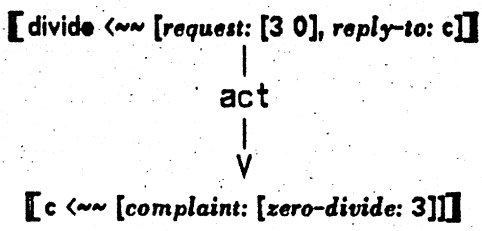
[x <~~ [*request:* m, *reply-to:* c]]

appears in the history of a computation then there is a unique event E_2 of the form

[c <~~ [*reply:* r]]

such that E_1 -*act*-> E_2 and there are no events E such that E_1 -*act*-> E -*act*-> E_2.

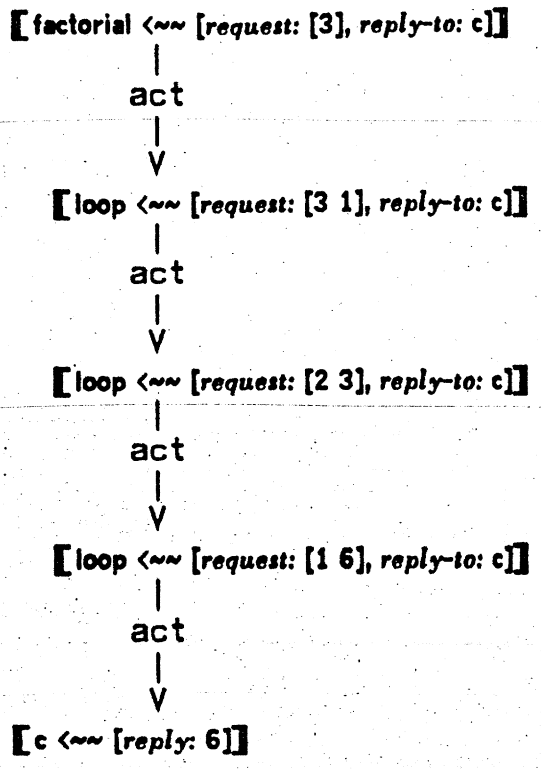
Complaint processing can easily be incorporated into the scheme. The history that results from divide[3 0] which attempts to divide 3 by 0 is shown below:



Since complaint processing does not have any profound implications for the results in this paper, we will not say anything more about the matter.

The history of the computation of factorial[3] using an iterative implementation of factorial illustrates how the activation ordering can be used to illustrate properties of control structures. We will suppose that factorial knows about an actor called loop which is sent tuples of the form [index product] where the initial index is 3 and the initial product is 1. Whenever loop receives a tuple [index product], where index is not 1, then it sends itself the tuple [(index - 1) (index * product)].

1: Later in this paper we will see examples of primitive actors such as fork and join primitives which are not simple.



The actor loop is iterative because it only requires the amount of working store[2] needed to store the index and product. Note that only one reply is sent to the continuation c even though c appears as the continuation in several request events.

IV.1.b — Laws for the Activation Ordering

It is not possible for there to be an infinite number of events in a chain[3] of activation between two given events in the activation ordering of the history of a computation. This law implies the existence of primitive actors. Stated more formally,

Law of Finite Activation Chains between two Events: **If C is a chain of events in the activation ordering from E_1 to E_2, then C is finite.**

The laws of finite activation chains is intended to eliminate "Zeno machines"--i.e. machines which compute infinitely fast. For example, consider a computer with your favorite instruction set which executes its first instruction in 1 microsecond, its second in 1/2 microsecond, its third in 1/4 microsecond, and so on. This machine not only could compute everything normally computable in less than 2

2: The careful treatment of the storage required for this example is given in [26].

3: A chain is a totally ordered sequence of events

microseconds, but could also solve the "halting problem". It could do this by simulating a normal computer running on some input, and if the simulation were still running after 2 microseconds, it could conclude that the simulated machine does not halt on that input.

Intuitively each event can directly activate only a finite number of other events. The events directly activated by an event E are called immediate successors of E (under the activation ordering -*act*->). The immediate successor set of E in the -*act*-> ordering, written immediate-succ$_{\text{-}act\text{->}}$(E), can be defined formally:

$$\text{immediate-succ}_{\text{-}act\text{->}}(E) \equiv \{E_1 \mid E \text{ -}act\text{-> } E_1 \text{ and } \neg\exists\, E_2 \text{ such that } E \text{ -}act\text{-> } E_2 \text{ -}act\text{-> } E_1\}$$

Then we have the following law:

Law of Finite Immediate Successors in the Activation Ordering:
For all events E, the set immediate-succ$_{\text{-}act\text{->}}$(E) is finite.

We define immediate predecessors in the activation ordering in a manner similar to that used for immediate successors. We postulate that an event is either an initial event, in which case it has no predecessors, or it is activated by a unique predecessor event.

Law of Uniqueness of Immediate Predecessors in the Activation Ordering:
For all events E, the set immediate-pred$_{\text{-}act\text{->}}$(E) has at most one element

This law is based on the physical intuition that two distinct events cannot both be the immediate cause of another event. This is because an event which immediately activates another event must have been the sender of the messenger for that second event. Thus each event E has at most one activator[4] which if it exists will be denoted as activator(E).

Note that the activation ordering analyzes the causality of the classical "fork-join" structure of parallel computations in an asymmetric manner. The reason is that the last event to arrive at the join is the one which activates the remainder of the computation. Later in this paper we will introduce another partial order on events [called the continuation order] which treats "fork-join" control structures in a symmetric fashion.

4: This usage of the term "activator" is somewhat in conflict with the usage of the term in Greif and Hewitt[40]. The usage here has the advantage that it is more firmly grounded in the physics of computation.

IV.2 — ARRIVAL ORDERINGS

Intuitively, the activation ordering can be identified with "causality" in which each event is "caused" by its activator. However, the activation ordering is not enough to specify the actions of actors with "side-effects", such as cells. For this reason, we introduce the arrival ordering $\text{-}arr\text{->}_x$ for an actor x whose behavior depends on the order of arrival of the messengers sent to x. The physical basis for defining the order of arrival is a hardware device called an arbiter. Note that there are only a few primitive actors such as cells and synchronization primitives whose behavior actually depends on the order in which messengers arrive.

Due to the totality of the order of arrival of messengers at an actor x (which will be discussed in more detail below), the notion of a "local time" for x is well-defined. Therefore, when talking about a single actor, we can talk rigorously about the changes in its vector of acquaintances over time.

IV.2.a — Laws for Arrival Orderings

The arrival ordering for each actor x is required to be a total ordering on all events which have x as their target. This policy is enforced by arbitration in actors such as synchronization primitives which need to observe the in order in which their messages arrive.

Arrival Ordering Law: If $E_1 \neq E_2$ and $target(E_1)=target(E_2)=x$,
then either $E_1 \text{ -}arr\text{->}_x E_2$ or $E_2 \text{ -}arr\text{->}_x E_1$

This law says that the messenger of E_1 arrives at x before the messenger of E_2 or vice-versa.

Note in connection with arrival orderings that there is no necessary relation between the arrivals of two messengers at a target and the ordering of their activator events. Suppose that events E_1 and E_2 have the same target x. Then, in general, the circumstance that $E_1 \text{ -}arr\text{->}_x E_2$ does not imply that $E_1 \text{ -}act\text{->} E_2$ since E_1 and E_2 might be distinct events of two asynchronous processes that both happen to send messengers to the same actor. Furthermore, the fact that $activator(E_1) \text{ -}act\text{->} activator(E_2)$ is no guarantee that $E_1 \text{ -}arr\text{->}_x E_2$; i.e. the messenger of E_2 might still arrive at the target actor before the messenger of E_1.

Given an event E_1 of the form $[\![T \langle\sim\sim M_1]\!]$ and an event E_2 of the form $[\![T \langle\sim\sim M_2]\!]$, there are only a finite number of events between these two events in the arrival ordering $\text{-}arr\text{->}_T$. Stated more formally:

Corollary: Law of Finite Chains between two Events in an Arrival Ordering:
For all events E_1 and E_2 such that $target(E_1) = target(E_2) = x$,
$\{E \mid E_1 \text{ -}arr\text{->}_x E \text{ -}arr\text{->}_x E_2\}$ is finite.

This eliminates anomalous behavior like the following: a cell receives the infinite sequence of "store" messages: [*store:* 1], [*store:* 1/2], [*store:* 1/4], [*store:* 1/8], etc. and then receiving a "contents?" message. What is it to reply? Zero? But zero was never explicitly stored into the cell!

The law of Finite Chains in the Arrival Ordering allows us to define immediate predecessors and immediate successors for the arrival ordering in a manner similar to the one used for the activation ordering. Since the Arrival Ordering Laws guarantee that the arrival ordering for each actor is total over its domain, successors and predecessors are unique when they exist. If an event E has an immediate predecessor in $\text{-}arr\text{->}_{target(E)}$ then it will be called the precursor of E and will be denoted by **precursor(E)**.

SECTION V --- CREATION of ACTORS

Intuitively the creation of an actor x must precede any use of x. In order to precisely state the above intuition as a law we must be more precise about when actors are created. For each actor x which is created in the course of a computation, we shall require that there is a unique event **creation(x)** which caused x to be created.

Let **created(E)** be the set (possibly empty) of actors created by the event E--i.e. the set of actors which claim E as their creation event. Note that x is not a participant in **creation(x)** because x does not come into existence until after **creation(x)** has occurred.

Definition: **created(E) ≡ {x| creation(x)=E}**

The intuition that a single event can only create finitely many objects is formalized as follows:

Law of Finite Creation: **For each event E, created(E) is finite.**

If an actor x is created in the course of a computation, then prior to any given message which it receives, it could only have received finitely many messages:

Law of Finitely Many Predecessors in the Arrival Ordering of a Created Actor:

If an actor x is created in the course of a computation and target(E) = x then

$\{E' |\ E' \text{-}arr\text{->}_x E\}$ **is finite.**

The above law is used in the next section to guarantee that our axiom which characterizes the behavior of a cell is well defined. The law guarantees that the process of repeatedly taking the precursor of an event with target t will find the creation event for t in a finite number of steps.

SECTION VI --- CELLS

VI.1 --- Axiom for Cells

The axiom for cells has two parts: involving their creation and use which can be stated as follows:

Creation: There is a simple primitive actor, called **create-cell**, such that whenever it is sent a tuple of the form [i], it creates an actor **s** which is a new storage cell with initial contents the actor i. More formally, for each event E_1 of the form E_1: [[**create-cell** <~~ [*request:* [i], *reply-to:* c]]] there is a unique event E_2 of the form E_2: [[c <~~ [*reply:* s]]] such that **s** is a newly created simple primitive actor and E_1 = **activator**(E_2). Furthermore **created**(E_1) = {**s**} which says that the only actor created by the event E_1 is the storage cell **s**. Thus each storage cell that is returned by **create-cell** differs from all previously created cells. The storage cell **s** always has exactly one acquaintance which is initially i. If E is an event which has **s** as its target, we will use the notation $\text{contents}_E(s)$ to denote this acquaintance at the time of the event E.

Use: A storage cell **s** can only be sent messages of the form [*contents?*] which requests the "current" contents and [*update:* x] which updates the contents to be x.

The contents of **s** when it receives one of these messages in an event E can be axiomatized using the arrival ordering for **s** as follows:

$\text{contents}_E(s) \equiv$
if E *has a precursor in the arrival ordering for* **s**
then
if **precursor**(E) *is of the form* [[**s** <~~ [*request:* [*update:* x], *reply-to:* ...]]]
then x
else $\text{contents}_{\text{precursor}(E)}(s)$
else i *which is the actor sent to* **create-cell** *to create* **s**

If E is an event of the form [[**s** <~~ [*request:* [*contents?*] *reply-to:* c]]] then there is a unique event E' of the form E': [[c <~~ [*reply:* $\text{contents}_E(s)$]]] such that E = **activator**(E').

VI.2 — Busy Waiting

Busy waiting is the kind of waiting used in some multi-processing systems. In this kind of waiting, the contents of a cell is continually checked and, if it is unchanged, the processor branches back to check it again. This kind of waiting is used when one processor cannot depend upon another to "wake it up" when the contents change. Busy waiting depends upon the property of Finite Chains between Events in the arrival orderings of cells.

For example suppose that a new storage cell **s** is created whose initial contents are 0. Furthermore suppose that the contents of **s** are updated exactly once by a process which sends **s** the message [*update:* 1]. Now another process might busy wait until the contents of the cell **c** change to 1 by executing a procedure of the following form:

```
loop:   if contents(s) = 0
            then goto loop
            else ...proceed...
```

The property of Finite Chains between Events in the arrival ordering for **s**, guarantees that the code ...proceed... will eventually be executed since otherwise there would be an infinite number of "contents?" messages before the [*update:* 1] message in the arrival ordering of **s**.

The use of the arrival ordering in the actor model of computation seems to help overcome one of the major limitations of other theories of the semantics of communicating parallel processes based on the Scott-Strachey model of computation [5,6]. The Scott-Strachey model is a deep mathematical study of functions that are minimal fixed points of "continuous" functionals. As currently developed the Scott-Strachey model seems to be a special case of the actor model in that it only deals with actors which behave like mathematical functions to the exclusion of actors such as cells and synchronization primitives whose behavior depends on the arrival ordering of messages sent to the actor.

SECTION VII --- LAWS of LOCALITY

We would like to formalize the physical intuition that computation is local and there can be no "action at a distance". The laws of locality presented in this section are intended to capture these intuitions.

The initial acquaintances of an actor are a subset of the participants in its creation event and the actors created by its creation event:

Initial Acquaintances Law: If an actor z is the target of an event E such that E is the first event in the arrival ordering of z then,

$$\text{acquaintances}_E(z) \subseteq \text{participants}(\text{creation}(z)) \cup \text{created}(\text{creation}(z))$$

The acquaintances of an actor can increase over its previous acquaintances only by the acquaintances of the messengers which it receives and the actors which it creates.

Precursor Acquaintances Law: If an actor z is the target of an event E
such that E has a precursor in the arrival ordering of z then,
$acquaintances_E(z) \subseteq participants(precursor(E)) \cup created(precursor(E))$

An actor x can only be the target or messenger in an event E if x is newly created or is an immediate participant in activator(E).

Activator Acquaintances Law: For each event E which is not an initial event
target(E) $\in$ participants(activator(E)) $\cup$ created(activator(E))
messenger(E) $\in$ participants(activator(E)) $\cup$ created(activator(E))

SECTION VIII --- COMBINED ORDERING

To make sense out of the activation and arrival orderings, and to relate them to a notion of "time", we introduce the precedes relation "-->":

Definition: --> is a binary relation on events which is the transitive closure of the union of the activation ordering -*act*-> and the arrival orderings $-arr\rightarrow_x$ for every actor x.

In order for --> to function as a notion of precedence, we require that the activation and arrival orderings be consistent. This is guaranteed by the Law of Strict Causality for actor systems which states that there are no cycles allowed in causal chains; i.e. it is never the case that there is an event E in the history of an actor system which precedes itself. Stated more formally the law of causality is that the combined ordering is also a strict partial ordering:

Law of Strict Causality: For no event E does E-->E.

Suppose that we have events in a computation described as follows:

E_1: [[x <~~ m_1]]
E_2: [[y <~~ m_2]]
E_3: [[y <~~ m_3]]
E_4: [[x <~~ m_4]]

E_1 -*act*-> E_2 ;arrival of m_1 at x causes the arrival of m_2 at y
E_2 $-arr\rightarrow_y$ E_3 ;m_2 arrives at y before m_3
E_3 -*act*-> E_4 ;arrival of m_3 at y causes the arrival of m_4 at x
E_4 $-arr\rightarrow_x$ E_1 ;m_4 arrives at x before m_1

The Law of Strict Causality states that the history of the computation given above is physically impossible to realize even though it is locally reasonable in the sense that any proper subset of the orderings can be realized. The above example of an impossible computation is due to Guy Steele.

Now we can define immediate predecessors and successors of an event E under -->. Note that an event

[t <~~ m] has at most two immediate predecessors in the relation --> one of which is the activator of the event and the other is the precursor of the event.

We would like to formalize the intuition, that between any two events which are causally related, that there are only finitely many events in a causal chain that connects the events. This intuition is formalized in the following law:

Law of Finite Chains between events in the Combined Ordering:[5]

There are no infinite chains of events between two events in the strict partial ordering -->.

We can use the combined ordering --> to express an important law about created actors.

Law of Creation before Use:

If an actor x is created in the course of a computation and E is an event with target x then

creation(x) --> E

VIII.1 — NESTED ACTIVITIES

Since one of the aims of actor theory is to study patterns of passing messages, we must identify several common patterns. The two most common types of messengers are requests and replies to requests. A request has two acquaintances: the request message itself, and a continuation actor which is to receive the reply. A reply to a request consists of a message sent to the continuation; this reply usually contains an answer to the request, but may contain a complaint or excuse for why an answer is not forthcoming.

We define the nested activity corresponding to a request event RQ in a computation to be the set of events which follow RQ in the combined order but precede any reply RP to the request. More formally, let E--≥ denote the set of events which follow E (including E itself) and --≥E denote the set of events which precede E (including E) in the computation. Then

$$\text{activity}(RQ) \equiv RQ\text{--}\geq \cap \cup\{\text{--}\geq RP \mid RP \text{ is reply to } RQ\}$$

Activities embody the notion of the nesting of activities that is produced by conventional programming languages, since we only include those events in an activity which contribute to a reply to that request. Note that if no reply is ever made to the request RQ in the computation, then the activity corresponding to RQ is incomplete and therefore vacuous.

If we let concurrent activities be those whose request events are unordered, then concurrent activities may overlap--i.e. share some events. However, this can only happen if the activities involve some shared actor which is called upon by both; if two concurrent activities involve only "pure" actors which by

---

5: This law is a strict generalization of the Law of Finite Chains between events in the Activation Ordering, the Law of Finite Chains between events in each Arrival Ordering, and the Law of Strict Causality. We conjecture that the Law of Finite Chains between events in the Combined Ordering can be proved using the Laws of Locality. If this conjecture is established then we would no longer need it as an an independent law.

definition have no arrival ordering and can be freely copied to avoid arbitration bottlenecks, then activities are properly nested, meaning that two activities are either disjoint, or one is a subset of the other.

The notion of activities allows one to vary the level of detail in using actors to model a real system. Let us define a primitive activity as the activity of a request which activates exactly one immediate reply, with no events intervening. Thus, a primitive activity always consists of exactly two events. A crude model for a system might represent an actor as primitive, i.e. one whose receipt events are all primitive. However, at a finer level of detail, one might model the internal workings of the actor as an activity in which a group of "sub"-actors participate.

SECTION IX --- CONTINUATION ORDERING

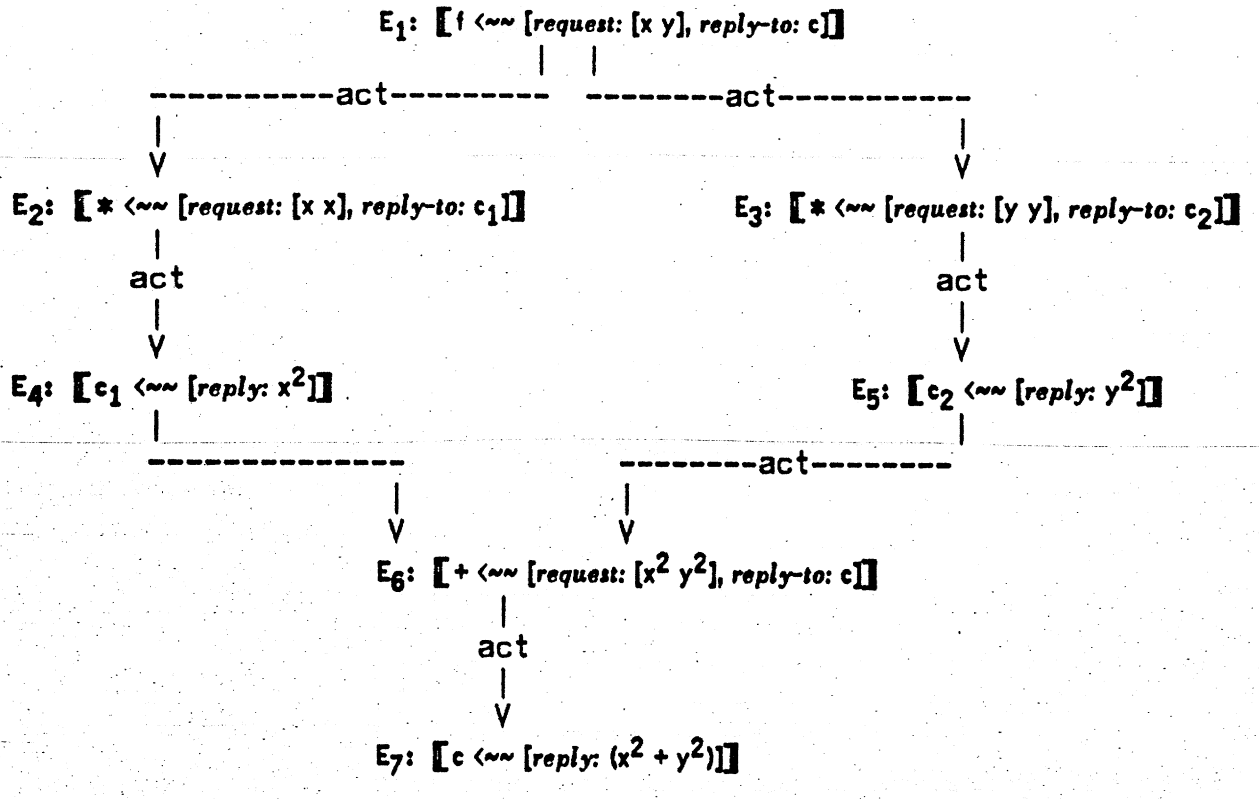
The notion of nested activities can be used to help explicate several of the various notions of "process" that have been used in computer science. In particular it can be used to define an ordering on events that is important to defining the semantics of programming languages for parallel processing. This new ordering is the continuation order and will be denoted by -*cont*->. The continuation ordering is important because it captures the usual operating system notion of "process" in terms of partial orders on events. Later in this paper we will show how to use the continuation ordering to provide a precise characterization of the relationship between the Scott-Strachey model and the actor message-passing model.

Definition: If E and E' are events then E -*cont*-> E' if
1: There is some activity α such that E, E' $\in \alpha$
and
2: E --> E'

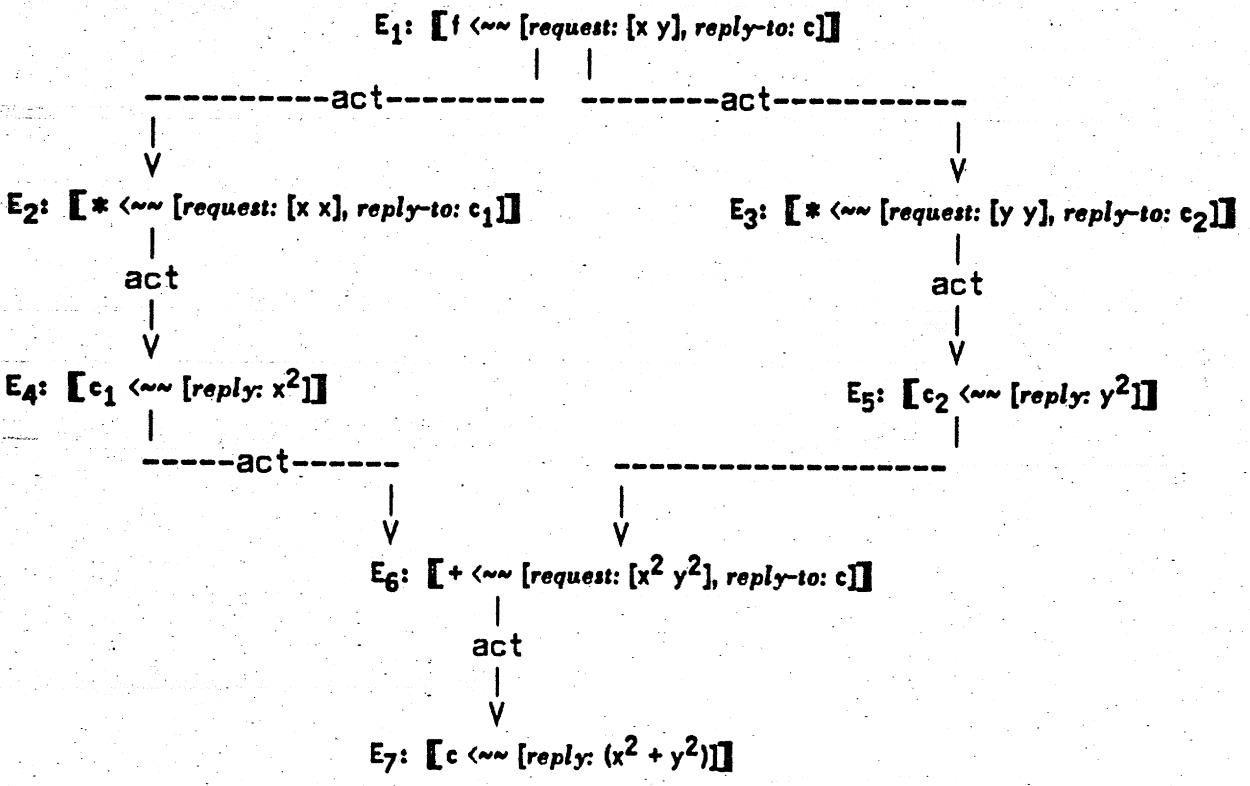
Note that each event has only finitely many predecessors and finitely many immediate successors in the continuation ordering because -*cont*-> is a sub-ordering of -->.

IX.1 -- Fork-Join Behavior

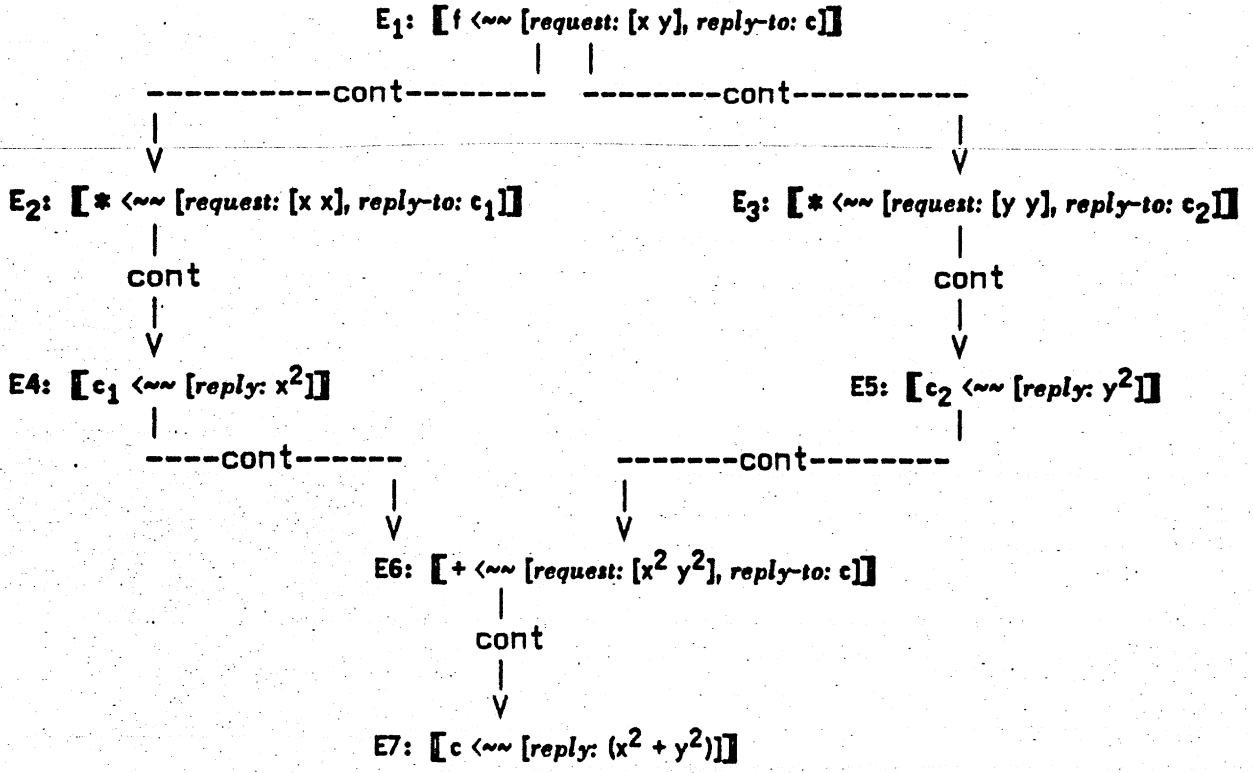
In programming languages for parallel processing, it is quite common to provide primitives by which processing can "fork" creating more parallelism which can later join together. Parallel evaluation of the arguments of a procedure provides a good example of fork-join behavior. All fork-join primitives have basically the same structure. Consider for example, the behavior of a procedure f which computes $(x^2 + y^2)$ given arguments x and y. Below are the two possible histories for an activity of f which produces these results where --> is used for the combined ordering:



Note that in the history given above that E_5 -*act*-> E_6 whereas in the history given below that E_4 -*act*-> E_6.

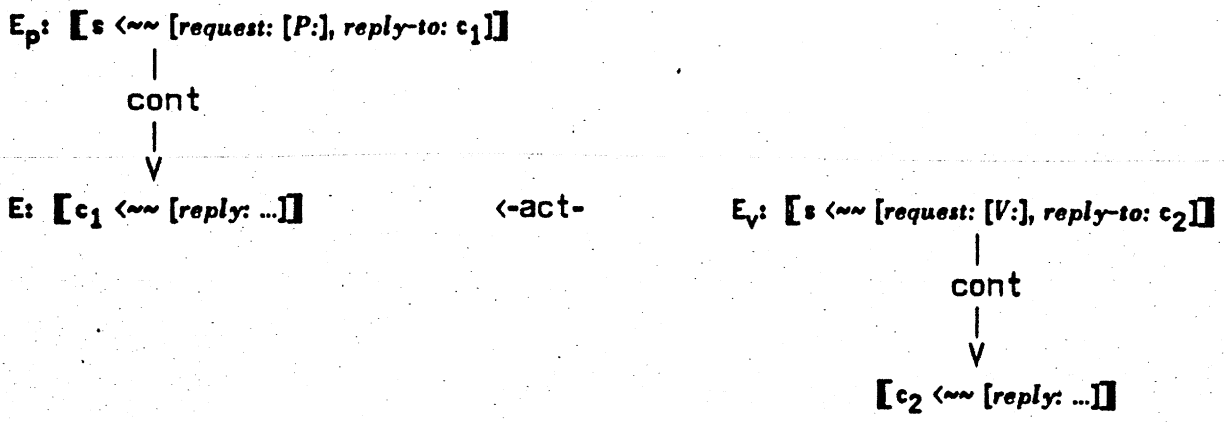


We shall say that E_1 is a fork event and that E_6 is a join event. In the above computation it will necessarily be the case that E_1 -*act*-> E_6 since this is the only way that E_6 can be activated. Therefore it will be the case that either E_4 -*act*-> E_6 or E_5 -*act*-> E_6. The continuation ordering -*cont*-> enables us to present the history of the computation without having to be concerned as to which of the above possibilities actually occured. Using the continuation ordering the symmetry of the above fork-join computation is demonstrated by the fact that the continuation ordering is the same for both of the above histories:



IX.2 -- Synchronization Between Processes

The behavior of semaphores provides a simple example to illustrate the relationship between the activation and continuation orderings. Suppose that s is a newly created semaphore whose capacity (count) is initially 0 so that the first attempt to perform a P operation will wait until a V operation is performed on the semaphore. In order to model the behavior of semaphores using message passing, we will suppose that P and V operations are implemented by sending [*P:*] and [*V:*] requests respectively. Suppose that E_p is the first event in the arrival ordering of s in which s receives a [*P:*] request and E_v is the next event in which s receives a [*V:*] request. The activation and continuation relations between these events is shown below:



Note that E_v --> E since E_v -*act*-> E but it is not the case that E_v -*cont*-> E because there is no activity in they are both elements.

SECTION X --- PROCEDURES

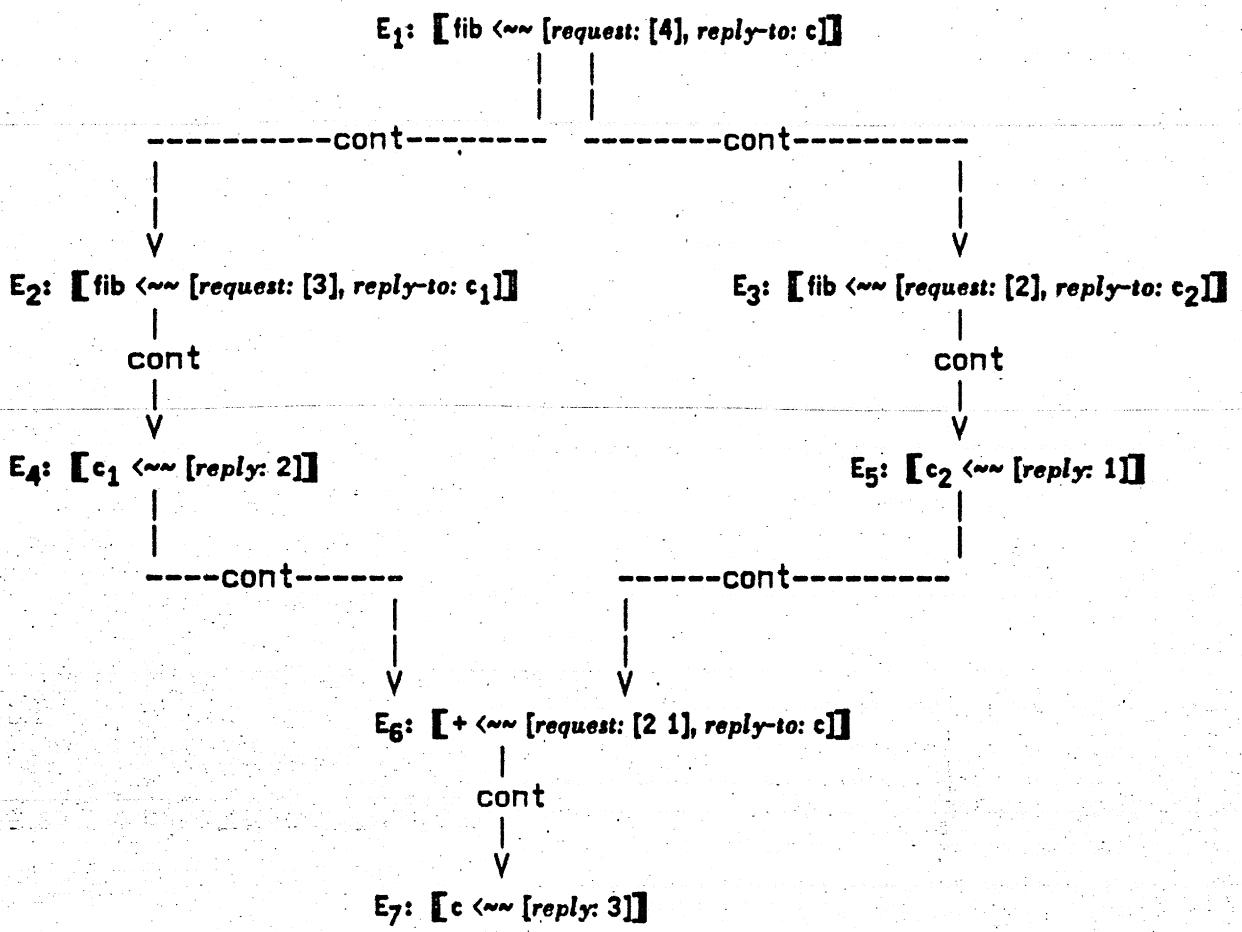
X.1 --- Behavior of Procedures

In this section we would like to characterize the behaviors of actors which behave like procedures. In order to do this we would like to use the notion of an activity.

To make our discussion more concrete we will consider the behavior of an implementation of the Fibonacci function defined as follows:

```
(fib n) ≡
   (if
      (n = 1) then 1
      (n = 2) then 1
      (n > 2) then ((fib (n - 1)) + (fib (n - 2))))
```

The following history is a partial order of some of the events that might result from evaluating (fib 4).

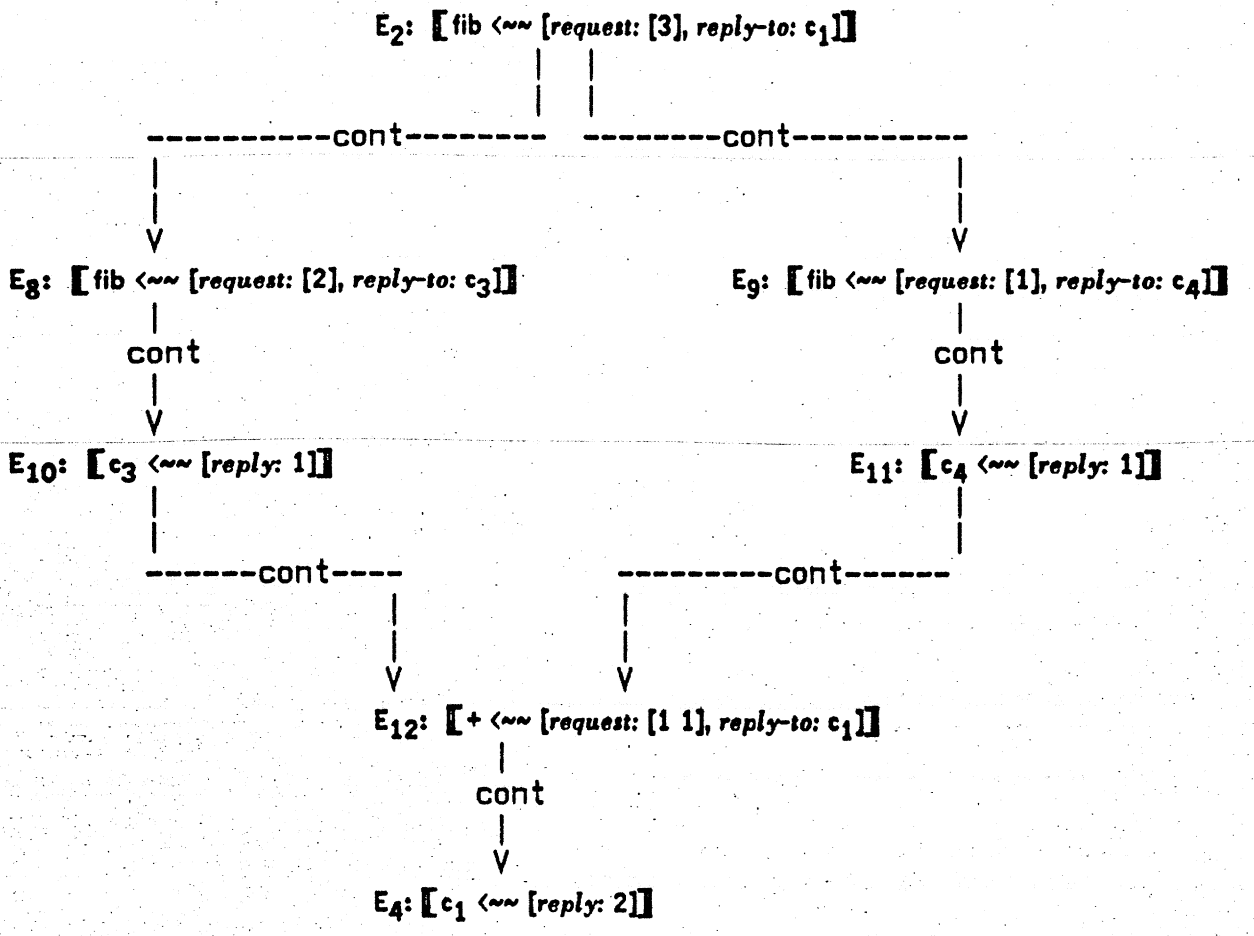


We will use the notation {|(p <= m) --> y|} to partially describe an activity which starts with an event of the form [p <~~ [request: m, reply-to: c]] and finishes with an event of the form [c <~~ [reply: y]].

All of the events shown in the above diagram are contained in one activity (which we will name α) of fib whose starting event is E_1 and whose finishing event is E_7. Thus the activity α is of the form {|(fib <= [4]) --> 3|}. The diagram above shows two sub-activities of α which we will call β and γ such that the following relationships hold.

β: {|(fib <= 3) --> 2|}	start(β) = E_2	finish(β) = E_4
γ: {|(fib <= 2) --> 1|}	start(γ) = E_3	finish(γ) = E_5

The activity β has events which are not shown in the above diagram. Some of these events are shown in the diagram below:



Thus we see that β in turn has sub-activities γ' and δ such that

γ': {|(fib <= [2]) --> 1|}	start(γ') = E_8	finish(γ') = E_{10}
δ': {|(fib <= [1]) --> 1|}	start(δ') = E_9	finish(δ') = E_{11}

Notice that both γ and γ' both satisfy the partial description {|(fib <= [2]) --> 1|} even though they are distinct activities which share no events in common. Uniquely identifying activities has the same problems as uniquely identifying objects and events: no finite local description will serve as a unique identification.

An actor f will be said to behave like a procedure if the following conditions hold for all the histories of f:

1: All of the messengers of events in the history are either requests or replies.

2: There is at most one reply event E to any given continuation actor in a history of f. Furthermore such an event must be an element of the activity of a request event of the form [[... <~~ [*request:* ..., *reply-to:* c]]] and every such request event must be an element of $predecessors_{-act->}(E)$.

3: The activities of f are properly nested. I.E. for any two activities of f it is the case that either one activity is a proper subset of the other or the two activities are disjoint.

X.2 — Limits of Continuous Functionals

The actor model of computation is based on axiomatizing the causal and incidental relations among computational events. The Scott-Strachey model of computation is based on the mathematical analysis of continuous function spaces. Superficially these two models might seem to have little in common. In this section we will analyze the relationship between these models of computation. Our main result is that if an actor behaves like a mathematical function then it is the limit of a continuous functional in the sense of Scott. This result follows from the law that each event has only finitely many immediate successors in the continuation ordering and the law of finite chains between two events in the continuation ordering.

Once again we will make the discussion concrete by considering the behavior of an implementation of the Fibonacci function defined by the following procedure:

```
(fib n) ≡
   (if
      (n = 1) then 1
      (n = 2) then 1
      (n > 2) then ((fib (n - 1)) + (fib (n - 2))))
```

Definition: Suppose f behaves like a mathematical function and that ⟨x y⟩∈f and ⟨x' y'⟩∈f.
Then ⟨x' y'⟩ will be said to be an immediate f-descendant of ⟨x y⟩ if
there is some history of f which has events E_1 and E_2 of the form

E_1: [[f <~~ [*request:* x, *reply-to:* ...]]]

E_2: [[f <~~ [*request:* x', *reply-to:* ...]]]

such that E_1 --> E_2 and it is not the case that there is an event E of the form

E: [[f <~~ [*request:* ..., *reply-to:* ...]]]

such that E_1 -*cont*-> E -*cont*-> E_2

For example ⟨2 1⟩ is an immediate fib-descendant of ⟨3 2⟩.

Definition: Suppose that $\langle x\ y\rangle \in f$

$$\text{immediate-descendants}_f(\langle x\ y\rangle) \equiv \{\langle x'\ y'\rangle \mid \langle x'\ y'\rangle \text{ is an immediate f-descendant of } \langle x\ y\rangle\}$$

$\text{immediate-descendants}_{fib}(\langle 1\ 1\rangle) = \{\}$
$\text{immediate-descendants}_{fib}(\langle 2\ 1\rangle) = \{\}$
$\text{immediate-descendants}_{fib}(\langle 3\ 2\rangle) = \{\langle 1\ 1\rangle\ \langle 2\ 1\rangle\}$
$\text{immediate-descendants}_{fib}(\langle 5\ 5\rangle) = \{\langle 3\ 2\rangle\ \langle 4\ 3\rangle\}$

Lemma: If f behaves like a mathematical function and $\langle x\ y\rangle \in f$ then $\text{immediate-descendants}_f(\langle x\ y\rangle)$ is finite.

Proof: Follows from the Law of Finitely Many Immediate Successors in the Activation Ordering.

Definition: If G is a set of input-output pairs then

$$D_f(G) \equiv \{\langle x\ y\rangle \mid \langle x\ y\rangle \in f \text{ and } \text{immediate-descendants}_f(\langle x\ y\rangle) \subseteq G\}$$

Intuitively $D_f(G)$ is the set of all input-output pairs of f that can be computed "immediately" from the input-output pairs in G. For example we have the following results for our implementation of the fibonacci function

$D_{fib}(\{\}) = \{\langle 1\ 1\rangle\ \langle 2\ 1\rangle\}$
$D_{fib}(\{\langle 1\ 1\rangle\ \langle 2\ 1\rangle\}) = \{\langle 1\ 1\rangle\ \langle 2\ 1\rangle\ \langle 3\ 2\rangle\}$
$D_{fib}(\{\langle 1\ 1\rangle\ \langle 2\ 1\rangle\ \langle 0\ 4\rangle\}) = \{\langle 1\ 1\rangle\ \langle 2\ 1\rangle\ \langle 3\ 2\rangle\}$
$D_{fib}(\{\langle 3\ 2\rangle\ \langle 4\ 3\rangle\}) = \{\langle 1\ 1\rangle\ \langle 2\ 1\rangle\ \langle 5\ 5\rangle\}$

Lemma: If f behaves like a mathematical function, then D_f is a continuous functional.

Proof: From its definition D_f is clearly monotonic. We will use N to denote the natural numbers [i.e. the non-negative integers]. Suppose that $\{X_i \mid i \in N\}$ is a chain of sets of ordered pairs so that $X_i \subseteq X_{i+1}$. To prove that D_f is continuous we shall prove that

$$\cup_{i\in N} D_f(X_i) = D_f(\cup_{i\in N} X_i)$$

Clearly

$$\cup_{i\in N} D_f(X_i) \subseteq D_f(\cup_{i\in N} X_i)$$

by the monotonicity of D_f. To prove the set inclusion the other way around suppose

$$\langle x,y\rangle \in D_f(\cup_{i\in N} X_i)$$

It follows from the definition of D_f that $\langle x,y\rangle \in f$ and

$$\text{immediate-descendants}_f(\langle x,y\rangle) \subseteq \cup_{i\in N} X_i$$

Therefore there exists a natural number n such that $\text{immediate-descendants}_f(\langle x,y\rangle) \subseteq X_n$ since the immediate f-descendants of $\langle x,y\rangle$ are finite. Thus $\langle x,y\rangle \in D_f(X_n)$ and

$$\langle x,y\rangle \in \bigcup_{i\in N} D_f(X_i)$$

Definition: A sequence $\langle x_i\ y_i\rangle$ such that each $\langle x_i\ y_i\rangle \in f$ will be said to be a descending f-chain if each $\langle x_{i+1}\ y_{i+1}\rangle$ is an immediate f-descendant of $\langle x_i\ y_i\rangle$.

Example: The following are descending fib-chains

[<6 8> <4 3> <3 2> <1 1>]
[<7 13> <5 5> <3 2> <2 1>]

Lemma: If $\langle x\ y\rangle \in f$ then there are only finitely many descending f-chains begining with $\langle x\ y\rangle$.

Proof: Follows from the fact that there are only finitely many events between two events of the form [[f <~~ [*request:* x, *reply-to:* c]]] and [[c <~~ [*reply:* y]]] in the continuation ordering.

Definition: If $\langle x\ y\rangle \in f$ then height(f,$\langle x\ y\rangle$) will be defined as the maximum length of the descending f-chains beginning with $\langle x\ y\rangle$.

Lemma: If $\langle x\ y\rangle \in f$ then $\langle x\ y\rangle \in D_f^{\text{height}(f,\langle x\ y\rangle)}(\{\})$ where D_f^n is the n-fold composition of D_f with itself.

Theorem: If an actor f behaves like a mathematical function then D_f is a continuous functional in the sense of Scott and f is the limit of D_f i.e.

$$\text{graph}(f) = \bigcup_{i\in N} D_f^i(\{\})$$

where graph(f) is the set of input-output pairs of f. It immediately follows that graph(f) is the minimal fixed point of D_f since

$$\text{graph}(f) = D_f(\text{graph}(f))$$

Conversely, if f is the limit of a continuous functional then the method used above can be used to construct a history for each request to f such that the histories are consistent and each history has the property that each event has only finitely many immediate successors and finitely many predecessors in the continuation ordering.

The above theorem makes precise the physical basis for believing that the graph of every physically realizable mathematical function is the limit of a continuous functional: the Law of Finitely Many Immediate Successors and the Law of Finite Chains between two Events in the Continuation Ordering. As currently developed the Scott-Strachey theory does not account for the the properties of the arrival orderings of actors such as synchronization primitives and shared data bases. An interesting topic that is left open for future research is how the Scott-Strachey theory can be extended in a natural way to encompass the physical constraints imposed by the arrival orderings of actors.

SECTION XI --- FUTURE WORK

When we first began our investigation into message-passing system we developed the intuitively appealing idea of "actors" as agents which communicate by passing messages. This intuitive notion proved to be too naive a basis for precise technical work in the same way that the intuitive notion of a "set" as a collection of objects proved to be too naive a basis in mathematics. The solution has been the development of the axioms in this paper which are intended to serve as the first step in developing axioms which capture the intuitive notion of actors as agents which communicate by sending and receiving messages.

There remains a great deal of work to be done in the development of the theory presented in this paper. The "completeness" of the axioms presented here needs to be intensively studied to determine if they can be significantly strengthened.

A mathematical characterization of the models which satisfy the axioms needs to be developed. The characterization should include a description of a standard model obtained by a constructive method for enumerating all the computation histories of a system that satisfy the axioms in this paper. The development of such a constructive model will prove the consistency of the axioms in this paper as well as providing a standard model in which the axioms can be interpreted.

We would like to apply the semantic theory developed in this paper in several directions. The semantics of programming languages for multi-processing problem solving languages such as KRL, OWL, PLASMA, SIMULA, SMALLTALK, AMORD, and the quantificational calculus need to be rigorously developed. In this way we hope to be able to make precise technical contributions to the "declarative-procedural" controversy.

There are a number of questions concerned with how efficiently actor systems can be implemented on networks of machines. In terms of the physical transport of information there are several ways in which an event can be implemented. The information in the messenger can be physically transported to the target; the target can be transported to the messenger, or the two can rendezvous at some other location. Under differing circumstances any one of the above possibilities might be more efficient. For example if the target is a small function which makes use of a large number of the extended acquaintances of the messenger then it is probably more efficient to transport the target to the messenger. On the other hand if the target is a large data base which is searched according to the directions of a small query in the messenger, then it is probably more efficient to transport the messenger to the target. Research is needed to develop dynamic mechanisms for deciding what information to transport for computations that are physically distributed on a network of machines. Hopefully some general mechanisms can be developed which, in practice, yield acceptable efficiency.

SECTION XII --- CONCLUSION

In this paper we have presented some laws that must be obeyed by the computations of communicating parallel processes. The theory is based on axiomatizing the causal and incidental relations between computational events where each event consists of sending a message. An important advantage of the actor message-passing model is that specifications for actors can be expressed directly in terms of the events involving those actors. Our approach is different from the more usual one which is to postulate the existence and "fairness" of some underlying global "scheduler" [21] or "oracle" [22]. Partial orders provide a means for concentrating on the causal relations among event as opposed to time relationships that result from some arbitrary interleaving.

The development of histories in the actor model of computation as partial orders of events as a generalization of the previous development as sequences of events has proven to be very fruitful. The partial orders $-act\rightarrow$, $-arr\rightarrow_x$ for each actor x, $-cont\rightarrow$, and $\rightarrow$, are all physically well grounded in the sense that if two events are observed to be related in a certain way in some observation frame then they will be observed to be related in the same way in all observation frames. Each of these different orderings serves its own purpose in the model. The following table summarizes the partial orders which we have introduced to describe the histories of computations:

$-act\rightarrow$	activation	causality between events
$-arr\rightarrow_x$	arrival	local time of arrival of messages sent to x
$\rightarrow$	combined	general notion of one event preceding another
$-cont\rightarrow$	continuation	nested activities

Partial orders of histories have been used to develop specification and proof techniques for modular synchronization primitives [32,34]. The machinery of partial orders of events provides the semantic glue needed to relate the specifications and implementations of communicating parallel processes.

This paper has traced some of the important relationships between the actor message-passing model of computation and classical denotational, semantics. It has been proved that every actor which behaves like a mathematical function is the limit of a continuous functional. This result provides a physical basis for the treatment of continuity in the Scott-Strachey theory of computation. The actor message-passing model has important applications for the semantics of communicating parallel processes which will be explored in subsequent papers.

SECTION XIII --- ACKNOWLEDGEMENTS

The research reported in this paper was sponsored by the MIT Artificial Intelligence and Laboratory and the MIT Laboratory for Computer Science under the sponsorship of the Office of Naval Research. A preliminary version of some of the laws in this paper were presented in an invited address delivered at the Conference on Petri Nets and Related Systems at M.I.T. in July 1975.

Our research on actors is an attempt to provide a semantic under standing of constructs for supporting modular programs that have been developed in programming languages and operating systems. The original impetus for the research came from a conversation with Alan Kay about the SMALLTALK language which he was designing. His idea was to base all computation on communicating objects each of which can have the power of a digital computer. The design of SMALLTALK built on the class instance distinction of SIMULA, the separation of goal language from method language in PLANNER, and the control ideas in David Fisher's thesis. We have worked to construct a theoretical model that encompasses these ideas in addition to similar abstractions which have been developed for operating systems such as domains of protection and capabilities.

This paper builds directly on the thesis research of Irene Greif. Many of the results in this paper are straightforward applications or slight generalizations of results in her dissertation. We are further indebted to Irene for the suggestion that the arrival ordering of an actor may be one of the fundamental differences between the actor model of computation and the Scott-Strachey model. Some of the notation for representing partial orders of events was developed at the Workshop on Language Features for Non-deterministic Programs which was held in Cambridge in August 1976. Many of the ideas presented in this paper have emerged in the last three years in the course of conversations with Irene Greif, Robin Milner, Jack Dennis, Jerry Schwarz, Joe Stoy, Richard Weyhrauch, Steve Ward, and Bert Halstead. Valdis Berzins, Henry Lieberman, Ernst Mayr, John Moussouris, Bruce Schatz, and Guy Steele made valuable comments and criticisms which materially improved the presentation and content of this paper. The arrow notation used for the different partial orders is due to Gary Fostel.

SECTION XIV --- BIBLIOGRAPHY

[1] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. Memo CA-7603-2911, Mass. Computer Assoc., Inc. March 1976.

[2] R. W. Floyd. Assigning Meanings to Programs in Mathematical Aspects of Computer Science (ed. J.T. Schwartz), Amer. Math. Soc., 1967, 19-32.

[3] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. CACM 12,10 (Oct. 1969), 576-580.

[4] V. Pratt. Semantical Considerations on Floyd-Hoare Logic. 17th IEEE Symp. on Founds. of Comp. Sci., Oct. 1976, 109-121.

[5] D. Scott. Outline of a Mathematical Theory of Computation. 4th Princeton Conf. on Inf. Sci. and Sys., 1970, 169-176.

[6] D. Scott. The Lattice of Flow Diagrams. Symp. on Semantics of Algorithmic Langs., Springer-Verlag Lecture Notes in Mat. 188, 1971.

[7] J. Vuillemin. Correct and Optimal Implementations of Recursion in a Simple Programming Language. J. of Comp. and Sys. Sci.. 9, 3, Dec. 1974.

[8] R. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. CACM 18,12 (Dec. 1975), 717-721.

[9] S. Owicki. A Consistent and Complete Deductive System for the Verification of Parallel Programs. 8th ACM Symp. Th. Comp., Hershey, Pa., May 1976, 73-86.

[10] R. Rivest and V. Pratt. The Mutual Exclusion Problem for Unreliable Processes. 17th IEEE Symp. on the Founds. of Comp. Sci., Oct. 1976, 1-8.

[11] E. Organick. The MULTICS System: An Examination of its Structure. MIT Press, 1972.

[12] W. Wulf, et al. HYDRA: The kernel of a multiprocessor operating system. CACM 17,6 (June 1974), 337-345.

[13] J. Dennis and D. P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. 2nd IEEE Symp. on Comp. Arch., N.Y., Jan. 1975, 126-132.

[14] G. Kahn. The Semantics of a Simple Language for Parallel Programming. IFIP-74, Stockholm, Sweden, North-Holland, 1974.

[15] Hoare, C.A.R. Communicating Sequential Processes. Dept. of Comp. Sci. The Queens of Belfast. Aug. 1976.

[16] J. Feldman. A Programming Methodology for Distributed Computing (among other things). TR9, Dept. of Comp. Sci., U. of Rochester, Feb. 1977.

[17] G. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula Begin. Auerbach, Phil., Pa., 1973.

[18] Learning Research Group. Personal Dynamic Media. SSL76-1, Xerox PARC, Palo Alto, Cal., April, 1976.

[19] B. Liskov and S. Zilles. Programming with Abstract Data Types. SIGPLAN Notices (April 1974), 50-59.

[20] B. Liskov. An Introduction to CLU. CSG Memo 136, MIT LCS, Feb. 1976.

[21] E. Cohen. A Semantic Model for Parallel Systems with Scheduling 2nd SIGPLAN-SIGACT Symp. on Princ. of Prog. Langs., Palo Alto, Cal., Jan. 1975.

[22] R. Milner. Processes: A Mathematical Model of Computing Agents. Colloquium in Math. Logic., Bristol, England, North-Holland, 1973.

[23] D. J. Farber, et al. The Distributed Computing System. 7th IEEE Comp. Soc. Conf. (COMPCON 73), Feb. 1973,31-34.

[24] R. Metcalfe and D. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. CSL 75-7, Xerox PARC, Palo Alto, Cal., Nov. 1975.

[25] K. E. Batcher. Sorting Networks and their Applications. 1968 S JCC, April 1968, 307-314.

[26] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. WP 92, MIT AI Lab., Dec. 1975. Accepted for publication in the A.I. Journal.

[27] R. Steiger. Actor Machine Architecture M.S. thesis, MIT Dept. EECS, June 1974.

[28] P. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. PhD Thesis, MIT Dept. of Elect. Eng. and Comp. Sci., June, 1977.

[29] H. G. Baker, Jr. List Processing in Real Time on a Serial Computer. WP 139, MIT AI Lab., Feb. 1977, also to appear in CACM.

[30] H. Baker and C. Hewitt. The Incremental Garbage Collection of Processes. ACM SIGART-SIGPLAN Symp., Rochester, N.Y., Aug. 1977.

[31] I. Greif. Semantics of Communicating Parallel Processes. MAC TR-154, MIT LCS, Sept. 1975.

[32] N. Goodman. Coordination of Parallel Processes in the Actor Model of Computation. MIT LCS TR-173, June, 1976.

[33] V. Berzins and D. Kapur. Path Expressions in Terms of Events. MIT Specification Group Working Paper, Dec. 1976.

[34] C. Hewitt and R. Atkinson. Synchronization in Actor Systems 4th SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang., Jan. 1977, 267-280.

[35] A. Holt, et al. Final Report of the Information System Theory Project, RADC-TR-68-305, RADC, Griffis AFB, N.Y., Sept. 1968.

[36] F. Furtek. The Logic of Systems. TR-170, MIT Lab. for Comp. Sci., Camb., Mass., Dec. 1976.

[37] G. Plotkin. A Powerdomain Construction. SIAM J. Comput. 5,3 (Sept. 1976), 452-487.

[38] D. J. Lehmann. Categories for Fixpoint Semantics. Theory of Computation TR 15, Dept. of Comp. Sci., Univ. of Warwick, 1976.

[39] C. Hewitt and H. Baker. Laws for Communicating Parallel Processes. IFIP-77 , Montreal, Aug. 1977.

[40] I. Greif and C. Hewitt. Actor Semantics of PLANNER-73 ACM SIGPLAN-SIGACT Conf., Palo Alto, Cal., Jan. 1975.

[41] Hoare, C. A. R. "Monitors: An Operating System Structuring Concept" CACM. October, 1975.

[42] Hansen, P.B. "Operating System Principles" Prentice-Hall. 1973.

[43] Bustard, D. W. "Parallel Programming Pascal (PPP)" Version 1. Dept. of Computer Science. Queen's University of Belfast. November 1975.

[44] Sullivan, H. and Bashkow T. R. "A Large Scale, Homogeneous, Fully Distributed Paralle Machine" Proceedings of Fourth Annual Symposium on Computer Architecture. March 23-25, 1977. pp 105-117.