Working Paper 134

November 1976

LAWS FOR COMMUNICATING PARALLEL PROCESSES

Carl Hewitt and Henry Baker

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

DRAFT COPY ONLY

Working Papers are informal papers intended for internal use. 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.

Laws for Communicating Parallel Processes

Carl Hewitt and Henry Baker

M.I.T. Artificial Intelligence Laboratory

545 Technology Square
Cambridge, Mass. 02139
(617) 253-5873

SECTION I --- Abstract

This paper presents some "laws" that must be satisfied by computations involving communicating parallel processes. The laws take the form of stating restrictions on the histories of computations that are physically realizable. The laws are intended to characterize aspects of parallel computations that are independent of the number of physical processors that are used in the computation.

SECTION II --- The Concept of an Actor

The theory presented in this paper attempts to characterize the behavior of active objects called actors in parallel processing systems. Actors are the fundamental (and only) objects in the theory. Actors interact with each other only by one actor sending another actor a *messenger*, which is also an actor. The actor to which the messenger is sent is called the *target*. Thus, the only kind of "event" in this model of computation is the *receipt* of a messenger by a target.

Upon the receipt of a messenger, a target actor exhibits *behavior* by sending messengers to other actors. However, an actor can send messengers only to other actors it "knows about", either through inheritance (being born with the knowledge), or acquiring it through messengers sent from other actors. Formally, each actor A is equipped with a set of *immediate acquaintances* acquaintances(A) which are the other actors it "knows". (It may or may not "know" itself; if it does, it can directly send itself messages!) This set of acquaintances can change over time as a result of the messengers which have been received by that actor.

We will also later need the notion of the set of *extended acquaintances* of an actor A which is A itself, plus the acquaintances of A, plus the acquaintances of the acquaintances of A, etc. More formally:

$$\text{acquaintances}^*(A) = \{A\} \cup \text{acquaintances}(A) \cup \text{acquaintances}^2(A) \cup \ldots \text{ (ad infinitum)}$$

Besides its set of acquaintances, each actor also has a *script* that determines what its behavior will be as a result of receiving a messenger.

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.

One example of an actor with non-trivial behavior is that of a *cell*. A cell is an actor which knows directly about only one other actor--its *contents*. When the cell is sent a messenger which consists of a 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 receipt 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.

SECTION III --- Concept of an Event

Events are the discrete steps in the ongoing history of an actor computation; they are the fundamental interactions of actor theory. Every event E consists of the *receipt* of a *messenger* actor, called **messenger(E)**, by a *target* (*recipient*) actor, called **target(E)**.

We will often use the notation:

$$[\![T \leftarrow\!\sim M]\!]$$

to describe an event consisting of the receipt of the messenger M by the target T. The **target(E)** and the **messenger(E)** are known as the (direct) *participants* of the event E. Therefore, we make the definition:

$$\text{participants}(E) = \{\text{target}(E), \text{messenger}(E)\}$$

We concentrate on the *receipt* of messengers rather than the *sending* of messengers because a messenger cannot affect the behavior of another actor until that actor receives it, nor can it affect the actor which sent it, since the recipient cannot know who sent it. The messenger might know the actor to whom any reply should be sent--the "continuation" of the sender--but only if the sender wishes a reply.

SECTION IV --- Orderings on Events

IV.1 --- Activation Ordering

One important 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 receipt of the messenger m_2 by x_2, is said to be ***activated*** by E_1, and E_1 is said to be the ***activator*** of E_2. Thus each event E has at most one activator which if it exists will be denoted as activator(E). We call the transitive closure of the "activates" relation the ***activation ordering*** and if E_1 precedes E_2 in this ordering then we write:

$$E_1 \text{ ++⟩ } E_2$$

In general ++⟩ is a partial ordering because an event E might activate several distinct events E_1, ..., E_n by causing a "fork".

A simple example which illustrates the use of ++⟩ 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 related as follows in the activation ordering:

⟦+ ⟨~~ [*request:* [3 4], *reply-to:* c]⟧ ++⟩ ⟦c ⟨~~ [*reply:* 7]⟧

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

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

$$[\![x \langle\sim [request: m, reply\text{-}to: c]]\!]$$

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

$$[\![c \langle\sim [reply: r]]\!]$$

such that $E_1 \mathrel{++>} E_2$ and there are no events E such that $E_1 \mathrel{++>} E \mathrel{++>} E_2$.

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 has an acquaintance 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)].

[[factorial <~ [request: [3], reply-to: c]]]
+
+
V
[[loop <~ [request: [3 1], reply-to: c]]]
+
+
V
[[loop <~ [request: [2 3], reply-to: c]]]
+
+
V
[[loop <~ [request: [1 6], reply-to: c]]]
+
+
V
[[c <~ [reply: 6]]]

The actor loop is iterative because it only requires an amount of working store 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.2 --- Laws for the Activation Ordering

It is not possible for there to be an infinite number of events between two given events in the activation ordering of the history of a computation. This law implies the existence of primitive actors. Stated more more formally, the Law of Discreteness for the activation ordering states that for any two events E_1 and E_2:

Law: If E_1 ++> E_2, then the set $\{E \mid E_1 \text{ ++> } E \text{ ++> } E_2\}$ is finite.

The discreteness of the activation ordering 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 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.

Discreteness assures the well-definedness of the *immediate successors* and the *immediate predecessors* of an event.

We assume that 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 "++>"). The *immediate successor set* of E (under "++>"), written $\text{immediate-succ}_{++>}(E)$, can be defined formally:

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

Then we have the following law:

Law: For all events E, the set $\text{immediate-succ}_{++>}(E)$ is finite.

We define immediate predecessors in the activation ordering in a manner similar to that used for immediate successors. We would like to 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: For all events E, the set $\text{immediate-pred}_{++>}(E)$ 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. It is inconceivable that a messenger can have two distinct senders.

An event E is an *initial event* for a strict partial ordering "<" if there does not exist another event E' such that E'<E.

Convention: There is a unique initial event $E_\perp$ in the activation order ++> such that for every other event E, $E_\perp$ ++> E.

Intuitively, $E_\perp$ provides the common environment that is necessary for the interprocess communication between two processes proceeding asynchronously. It provides a convenient technical device for expressing properties of computations, such as the intuition that there are only finitely many processes, that are otherwise difficult to express. Another technical reason for the introduction of $E_\perp$ this that it facilitates the comparison of the theory developed here with the Scott-Strachey model of computation. Discreteness of the activation ordering implies that computation is continuous.

The intuition that there are only finitely many objects in the beginning can be expressed as follows:

Convention: acquaintances*($E_\perp$) is finite.

Given the uniqueness of predecessors, we can define for each event $E \neq E_\perp$ the function activator(E) to be the predecessor of E (in "++>"). Successive applications of the function activator traces a finite path of activation from each event back to the initial event $E_\perp$.

IV.3 --- Arrival Orderings

Intuitively, the activation ordering can be identified with "causality"--if an actor x receives a messenger m_1 in an event E_1 and then sends a messenger m_2 to a target t, then the event E_2--which is the receipt of m_2 by t--is "caused" by E. 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* which specifies for each actor the order of arrival of the messengers sent to it. This ordering is required to be total, a policy which is enforced by arbitration.

For each actor x we postulate that there is a *total ordering* $\Rightarrow_x$ on events in which x is the target; i.e. if two distinct events E_1 and E_2 both have x as their destination then one must precede the other relative to $\Rightarrow_x$.

Law: If $E_1 \neq E_2$ and target(E_1)=target(E_2),
then either $E_1 \Rightarrow_x E_2$ or $E_2 \Rightarrow_x E_1$ where x=target(E_1)=target(E_2)

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 share

the same target x. Note that in general the circumstance that $E_1 =>_x E_2$ does not imply that $E_1 ++> E_2$ since E_1 and E_2 events of two processes running asynchronously that both happen to send a message to x. Furthermore the circumstance that activator(E_1) ++> activator(E_2) is no guarantee that $E_1 =>_x E_2$; i.e. the messenger of E_2 could still arrive at x before the messenger of E_1.

IV.3.a --- Finitely Many Predecessors in the Arrival Ordering of an Actor

Given an event of the form [[T <~~ M]], there are only a finite number of events which precede it in the arrival ordering $=>_T$ for T. Stated more formally:

> ***Law*: For all actors x and events E, the set $\{E' | E' =>_x E\}$ is finite.**

One corollary to this law is that for every actor x, the arrival ordering $=>_x$ is *discrete*. Suppose that the arrival ordering for a cell were not discrete. Then the cell could receive the infinite sequence of "store" messages: [*store:* 1], [*store:* 1/2], [*store:* 1/4], [*store:* 1/8], etc. before receiving a "contents?" message. What is it to reply? Zero? But zero was never explicitly stored into the cell!

IV.3.b --- Axiom for Cells

> *Axiom*: There is a simple primitive actor called cons-cell such that whenever it is sent a tuple of the form [v], it creates an actor s which is a new storage cell with initial contents the actor v. More formally for each event E_1 of the form
>
> [[cons-cell <~~ [*request:* [v], *reply-to:* c]]]
>
> there is a unique event E_2 of the form
>
> [[c <~~ [*reply:* s]]]
>
> such that s is a newly created simple primitive actor and $E_1 ++> E_2$.
>
> Intuitively, the contents of the cell x is the value sent with the last "store" message in the arrival ordering, if one exists, or the initial contents v, if no store message has yet been received. More formally for each request $E_{\text{contents?}}$ with target s and messenger with a "contents?" message there is a corresponding reply event which has the initial contents v as its message if $E_{\text{contents?}}$ has no predecessors in $=>_x$ with messages of the form [*store:* n]. If there is such a predecessor, then the reply to $E_{\text{contents?}}$ must have message n where the last store request which precedes $E_{\text{contents?}}$ has message [*store:* n].

The above axiom can be used to prove the Floyd-Hoare axioms for assignment statements. The Floyd axiom assures us that if $P[c_1, ..., c_n]$ is true before an assignment statement of the form "$x_1 \leftarrow E[x_1, ..., x_n]$" is executed where $c_1, ..., c_n$ are contents of cells $x_1, ..., x_n$ respectively, then after execution of the assignment statement there exists a quantity c such that the contents of x_1 is $E[c, x_2, ..., x_n]$ and furthermore that $P[c, c'_2, ..., c'_n]$ is true where $c'_1, ..., c'_n$ are the contents of $x_1, ..., x_n$.

$$P[x_1, ..., x_n] \{x_1 \leftarrow E[x_1, ..., x_n]\} \exists c \; x_1 = E[c, x_2, ..., x_n] \text{ and } P[c, x_2, ..., x_n]$$

The Floyd axiom is easily proved from our axiom for cells. In order to give the proof we must make explicit the assumptions on which the axiom is based. One important assumption is that all the variables $x_1, ..., x_n$ in P refer to the contents of distinct cells. It is important to make this assumption explicit because it does not hold for many commonly used programming languages such as FORTRAN and ALGOL. Suppose that $P[c_1, ..., c_n]$ is true where $c1, ..., c_n$ are the contents of $x_1, ..., x_n$ respectively just before the assignment statement is executed. Thus the cell x_1 will be sent a message of the form [*store:* $E[c_1, ..., c_n]$]. Another important assumption in the Floyd axiom is that there is no parallelism in the system so that the activation ordering ↔ is a total ordering. The assumption that there is no parallelism together with the assumption that all the cells are distinct together imply that contents $c'_2, ..., c'_n$ of $x_2, ..., x_n$ after the assignment statement have not changed since there are no further events in their arrival ordering. Furthermore the contents c'_1 of x_1 after the assignment statement is $E[c_1, ..., c_n]$. Therefore the conclusion of Floyd's axiom has been established where we choose c to be $E[c_1, ..., c_n]$.

Hoare has given another axiom for assignment statements which is also easily derived from our axiom for cells:

$$Q[E[x_1, ..., x_n], x_2, ... x_n] \{x_1 \leftarrow E[x_1, ..., x_n]\} Q[x_1, ..., x_n]$$

IV.3.c --- Busy Waiting

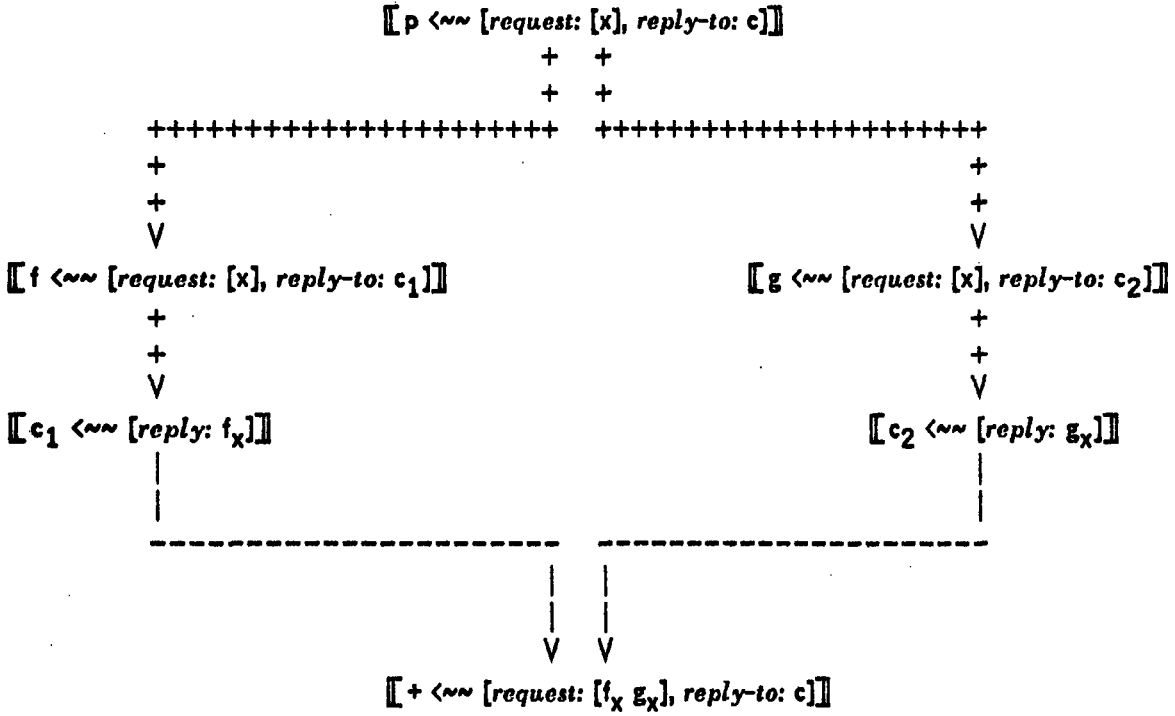
Busy waiting is the kind of waiting used in most multi-processing systems. In this kind of waiting, the contents of a cell is continually checked and, if it is unchanged, the processor branches back to check it again. This kind of waiting is used when one processor cannot depend upon another to "wake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

IV.4 --- General Precedes Relation and the Law of Causality

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 ++> and the arrival orderings $=>_x$ for every actor x.

We would like to present a simple example to illustrate the combined ordering. Consider the behavior of a program p which whenever it receives an input x computes the value of f of x and the value of g of x in parallel and returns the sum of the values. Such a program would produce a history with the following structure:



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 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

Law: For no event E does E-->E.

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

E_1 is of the form $[\![x \leftsquigarrow m_1]\!]$
E_2 is of the form $[\![y \leftsquigarrow m_2]\!]$
E_3 is of the form $[\![y \leftsquigarrow m_3]\!]$
E_4 is of the form $[\![x \leftsquigarrow m_4]\!]$

The Law of Causality states that the history of the computation given below is physically impossible to realize even though it is locally reasonable in the sense that any subset of the orderings can be realized.

$E_1 \mathrel{+\!\!+\!\!>} E_2$;receipt of m_1 by x causes the receipt of m_2 by y
$E_2 \Rightarrow_x E_3$;x receives m_2 before m_3
$E_3 \mathrel{+\!\!+\!\!>} E_4$;receipt of m_3 by y causes the receipt of m_4 by x
$E_4 \Rightarrow_y E_1$;y receives m_4 before m_1

The above example of an impossible computation is due to Guy Steele.

Unfortunately, discreteness of the activation ordering $\mathrel{+\!\!+\!\!>}$ and the arrival orderings $\Rightarrow_x$ is not sufficient to imply that "-->" must be discrete. Therefore, we must introduce it as a separate law:

Law: The general precedes relation "-->" is a discrete relation.

Now we can define immediate predecessors and successors of an event E under -->. Note that an event $[\![t \leftsquigarrow m]\!]$ has at most two immediate predecessors in the relation --> one of which is the activator of the event and the other is the immediate predecessor of the event in $\Rightarrow_t$.

SECTION V --- Law of Creation

Intuitively the creation of an actor x must precede the 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, we shall require that there is a unique event birth(x) in which x first makes its appearance. More precisely birth(x) has the property if x is a participant in another event E then

birth(x) --> E .

Each actor is either an extended acquaintance of a participant of $E_\perp$, or is created during the course of the computation. It follows that, if x is created in the course of a computation then we can the conception event of an actor x as follows

conception(x) = activator(birth(x))

so that the conception causes the birth event in which x first makes its appearance.

We have the immediate corollary that an actor cannot be used before it has been conceived:

Corollary: $\neg \exists E \quad E \rightarrow conception(x)$ and $x \in acquaintances^*(participants(E))$

SECTION VI --- Law of Locality

The Law of Locality is intended to formalize the notion that the acquaintances of an actor could have been acquired only through meeting them, or some other actor that knew them, and so on. The statement of the law for general actor systems is quite subtle. Therefore, we first present the law for actor systems consisting entirely of *immutable* actors. In this context, a *immutable* actor is one whose set of extended acquaintances does not change with time. This restriction eliminates the possibility of ***cells***, because cells by definition have one acquaintance--their contents--which is supposed to change when the cell is updated.

Let **target(E)** be the *target* of the event E, let **messenger(E)** be the *messenger* of the event E, and let **conceived(E)** be the set (possibly empty) of actors "conceived" in the event E--i.e. the set of actors which claim E as their conception event. Then we define the set of *extended-participants* of an event E as follows:

$$participants^*(E) = acquaintances^*(target(E)) \cup acquaintances^*(messenger(E))$$

The Law of Locality (for systems which only contain immutable actors) then has two parts:

(1) For all actors A,
$$acquaintances(A) \subseteq participants^*(conception(A)) \cup conceived(conception(A))$$
(2) For all events $E \neq E_\perp$,
$$participants(E) \subseteq participants^*(activator(E)) \cup conceived(activator(E))$$

Part (1) of the Law of Locality states that the immediate acquaintances of an actor A must all be extended-participants of its conception event. In other words, an actor cannot know about another actor which was not known about, either directly or indirectly, when it was conceived.

Part (2) of the Law of Locality states that both the target and messenger of an event must have been known, either directly or indirectly, to the participants of its activator event. By part (1) then, if either the target or the messenger were newly conceived, then their acquaintances are a subset of the extended participants of the activator event.

SECTION VII --- An Example: Church's Lambda Calculus

In this section, we would like to describe an actor system for evaluating expressions in Church's λ-calculus.

We first must describe the *syntax* of λ-expressions. λ-expressions are either *atomic symbols*, *λ-abstractions*, or *combinations*. Given a λ-expression <e>, we can form a *λ-abstraction* whose *bound-variable* is a particular atomic symbol--say x--thereby creating another λ-expression "λx.<e>" whose *body* is <e>. Given two λ-expressions $<e_1>$ and $<e_2>$, we can create the λ-expression $(<e_1> <e_2>)$, called a *combination*, in which $<e_1>$ is the *operator* and $<e_2>$ is the *operand*.

In order to *evaluate* λ-expressions, we create a system of actors for the λ-expression to be evaluated. Each atomic symbol, each λ-abstraction, and each combination becomes a distinct actor.

In order to cause these actors to compute, we must send the actor which we have created from the given λ-expression a messenger which consists of a *command* "eval", an environment which is an actor that provides the value associated with a given symbol, and a continuation actor which will receive the final value computed, if any.

Instead of specifying a programming language for writing down each kind of actor, we will give axioms which specify the required behavior for each actor in a manner similar to our axioms for cells.

Intuitively, an *environment* acts like a function from atomic symbols to actors which are the "values" of those atomic symbols in that environment. If an environment e receives a message of the form [*bind:* x, *to:* y], then it replies with a new environment e' whose behavior is defined as follows: Whenever e' is sent a messenger of the following form

[*request:* [*lookup:* z], *reply-to:* c]

then if z=x, c is sent the reply y, otherwise e is sent the messenger

[*request:* [*lookup:* z], *reply-to:* c]

We will use the notation (*extend* e *by-binding* x *to* y) for the actor e'. There is an empty environment called empty whose behavior, when sent a message of the form [*lookup:* x] is undefined.

Each *atomic symbol* x, when sent the messenger [*request:* [*eval:* e], *reply-to:* c] causes the event ⟦e <~~ [*request:* [*lookup:* x], *reply-to:* c]⟧. In other words, an atomic symbol x returns the value with which it is associated in the environment e when it is sent a message of the form [*eval:* e].

Each *λ-abstraction* λz.body, when sent a request message [*eval:* e] immediately replies with a newly created actor (called closure_e) such that whenever there is an event of the form

$$[\![\text{closure}_{\mathbf{e}} \leftarrow\!\sim [\textit{request:}\ \mathbf{m},\ \textit{reply-to:}\ \mathbf{c}]]\!]$$

then there is an event of the form

$$[\![\mathbf{body} \leftarrow\!\sim [\textit{request:}\ [\textit{eval:}\ \mathbf{e}'],\ \textit{reply-to:}\ \mathbf{c}]]\!]$$

where **e'** = (*extend* **e** *by-binding* **z** *to* **m**).

Combinations can be evaluated in several different ways, each having different termination properties. We can model this flexibility by the way in which a combination behaves when sent an "eval" message. We will give axioms for two of the most interesting possibilities; other possibilities can readily be axiomatized using similar techniques.

First we will show how to axiomatize "normal order" evaluation. Each combination (**operator operand**) is an actor which, when sent the messenger [*request:* [*eval:* **e**], *reply-to:* **c**], causes the operator to be sent the message [*eval:* **e**]. If a reply v_{operator} is received to this latter request then v_{operator} is in turn sent the messenger [*request:* (*delay* **operand e**), *reply-to:* **c**] where (*delay* **operand e**) is a "delayed" form of the expression **operand**. In general the actor (*delay* **z e**) is defined to have the following behavior: The first time it receives a messenger m_0, it sends the request message [*eval:* **e**] to **z**. If a reply v_z is received to this request then v_z is sent the messenger m_0. Subsequent messengers **m** received by (*delay* **operand e**) are directly passed to v_z without re-evaluating the expression **z**.

In a similar way we can axiomatize the behavior of combinations that maximize the amount of parallelism in the computation. When a combination (**operator operand**) receives a request message of the form [*eval:* **e**] then it immediately replies with message **f** (whose behavior is defined below) which represents the "future" value of the combination and in parallel activates the evaluation of the expressions **operator** and **operand** by sending them [*eval:* **e**] messages. If replies v_{operator} and v_{operand} [which themselves might be futures!] are received for these requests then v_{operator} is sent the message v_{operand}.

Now we are in a position to define the behavior of the actor **f** which represents the "future" value of the combination. If a reply $v_{\text{combination}}$ is received to the message sent to v_{operator} and if a messenger **m** is received by **f**, then **m** is sent to $v_{\text{combination}}$. Note that although this scheme for the evaluation of combinations is highly parallel it may not be very efficient because it may involve the computation of many values that are never used in the computation of the final value of the combination.

SECTION VIII --- Actor Induction

VIII.1 --- Specifications

The general precedes ordering among events is fundamental to many important computational concepts. The concept that F always converges (terminates) can easily be expressed in terms of events: If an event E of the form

⟦F <~~ [*request:* argument-tuple, *reply-to:* continuation]⟧

[supplying F with an argument-tuple and a continuation actor to which the reply should be sent] occurs in the history of a computation then there must be another event E' (E-->E') such that

⟦continuation <~~ [*reply:* answer]⟧

in which the continuation actor is actually sent the reply "answer".

The precedes ordering can also be used to state a powerful inductive principle for proving properties of computational systems. The message-passing induction principle unifies the separate rules that are currently used to prove properties about for data structures and procedures. The idea is to associate a ***contract monitor*** (input specification) with each actor in a computational system which checks that every messenger which is received by that actor satisfies the specification. At first this idea may seem to lack generality because it neglects to consider output specifications as well as input specifications. However, the contract monitor M for an actor F can check the ***output*** specifications for F by placing another contract monitor M' on the ***continuations*** of messages sent to F.

Suppose we want express the specification that whenever an actor X is sent a message which satisfies the input predicate I then its reply must satisfy the output predicate O. We can simply make M_I a contract monitor for the actor X such that whenever X is the destination of an event of the form

⟦X <~~ [*request:* message, *reply-to:* C]⟧

the contract monitor M_I constructs a new actor C' which which has O as its contract monitor. Whenever C' is sent a messenger m it simply immediately turns around and sends m to C, after checking that O is true of m. In a similar way we can express "invariant properties". An invariant property I of of a actor X is a property that, if true of its message, will also be true of its reply. We simply make I the contract monitor of X when it is created and furthermore place a contract monitor I' on the continuation of every messenger which is sent to X. The purpose of I' is simply to check that the invariant I continues to hold when the operation of X is completed.

Like most induction rules actor induction has is a two step process:

Basis Step: Prove that an actor X satisfies its contract monitor at its *birth*; i.e. it is initialized to the proper initial state when it is created.

Induction Step: Given that an actor X satisfies the contract monitor for X and then receives the messenger M which satisfies the contract monitor for messengers to X, then prove that the contract monitors for all events directly activated by the receipt of M by X are satisfied and that the contract monitor for X is still satisfied after these activations.

SECTION IX --- FUTURE WORK

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. A constructive theory needs to be developed for enumerating all the the computation histories of a system that satisfy the axioms in this paper.

SECTION X --- 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.

Conversations with Alan Kay, John McCarthy, Alan Newell, and Seymour Papert were useful in getting us started on this line of research. The development of the model of communicating parallel processes presented in this paper has been greatly influenced by Seymour Papert's "little people" model of computation, a seminar given by Alan Kay at M.I.T. on an early version of SMALLTALK, and the work of Church, Fischer, Landin, McCarthy, Milner, Morris, Plotkin, Reynolds, Scott, Stoy, Strachey, Tennent, Wadsworth, etc. on formalisms based on the lambda calculus.

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.

Henry Lieberman made valuable comments and criticisms which materially improved the presentation and content of this paper.

SECTION XI --- CONCLUSION

In this paper we have presented the axioms for a theory of the computations involving communicating parallel processes that are physically realizable. The theory is based on axiomatizing the causal and incidental relations between computational events where each computation event consists of sending a message.

The theory developed in this paper has been used to study synchronization problems involved in communicating parallel processes. It provides the semantic glue needed to relate the specifications and implementations of communicating parallel processes. The semantics of programming languages for communicating parallel processes can be rigorously developed by treating the constructs in the language as actors whose behavior is axiomatized when they are sent "eval" and "match" messages.

SECTION XII --- BIBLIOGRAPHY

Atkinson, R. and Hewitt, C. "Synchronization in Actor Systems" SIGPLAN-SIGACT Symposium on Programming Languages. Jan. 17-19, 1977. Los Angeles, Calif.

Church, A. "The Calculi of Lambda Conversion" Annals of Mathematical Studies 6, Princeton University Press, 1941, 2nd edition 1951.

Floyd, R.W. "Assigning Meanings to Programs" in *Mathematical Aspects of Computer Science* (ed. J.T. Schwartz), 19-32, 1967.

Greif, I. "Semantics of Communicating Parallel Processes" MAC Technical Report TR-154. September 1975.

Greif, I. and Hewitt, C. "Actor Semantics of PLANNER-73" Proceedings of ACM SIGPLAN-SIGACT Conference. Palo Alto, California. January, 1975.

Hoare, C.A.R. "An Axiomatic Basis for Computer Programming" *CACM* 12, 576-580, 1969.

Learning Research Group "Personal Dynamic Media" Technical report. Xerox Palo Alto Research Center. 1976.

Palme, J. "Protected Program Modules in SIMULA-67" Technical Report PB-224 776. Research Institute of National Defense. Stockholm. July 1973.

Reynolds, John. "User-defined Types and Procedural Data Structure as complementary Approaches to Data Abstraction" University of Syracuse. 1975.

Vuillemin, J. "Correct and Optimal Implementations of Recursion in a Simple Programming Language. Journal of Computer and System Sciences. vol 9. no 3. December 1974.

Wadsworth, Christopher. "Semantics and Pragmatics of the Lambda-calculus" Ph. D. Oxford. 1971.