MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Working Paper 134A May 10, 1977

Laws for Communicating Parallel Processes

by

Carl Hewitt and Henry Baker

This paper presents some laws that must be satisfied by computations involving communicating parallel processes. The laws are stated in the context of the *actor theory*, a model for distributed parallel computation, and take the form of stating plausible restrictions on the histories of parallel computations to make them physically realizable. 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 proven theorems in mathematics. The laws are used to analyze the mechanisms by which multiple processes can communicate to work effectively together to solve difficult problems.

Since the causal relations among the events in a parallel computation do not specify a total order on events, the actor model generalizes the notion of computation from a *sequence of states* to a *partial order of events*. The interpretation of unordered events in this partial order is that they proceed concurrently. The utility of partial orders is demonstrated by using them to express our laws for distributed computation.

Key Words and Phrases: parallel processes, parallel or asynchronous computations, partial orders of events, Actor theory.

CR Categories: 5.21, 5.24, 5.26.

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

This paper will be presented in August at IFIP '77 in Toronto, Canada
and is intended for internal use only.

I. INTRODUCTION

In recent years, there has been a shift from the centralized serial computing system to the distributed[1] parallel computing network. There are many driving forces behind this shift: the geographical distribution of information sources and sinks, requirements for faster response, cheaper small computers, replacement of paper shuffling with digital message switching, greater system reliability, and so on. In response to this demand, many popular computer languages have added "parallel process" and "message switching" constructs which in many cases are little more than syntactic sugar for very low level operating system tasking facilities. However, little has been done in the way of providing a precise semantics for what these constructs mean.

But a semantics for distributed computing would be required even without geographically distributed systems because tightly coupled systems[2] of large numbers of processors are becoming available which are more powerful than the fastest serial computers in terms of instructions per second. Taking advantage of such massive parallelism has proved more difficult than was expected due to the lack of conceptual and formal models for these systems.

The semantics of serial programs are quite well understood using Floyd-Hoare logic [2,3,4] and the Scott lattice model [5,6,7]. However, most current semantics for parallel programs try to map parallel computations into non-deterministic serial ones by "considering all shuffles" of the elementary steps of the various parallel processes [8,9]. This approach leads to program proofs with a frightful number of cases and no insight [10], and is unnatural because it models independent, local computation steps as sequential incremental changes to the *global state* of a system.

Actor theory is an attempt to remedy these defects. Computations in the actor model are partial orders of events. Hence, specifications for an actor and correctness assertions for a computation can be given very naturally in terms of events and causal relations among events. Since inference rules can use these partial orders directly, the number of cases in proofs is considerably reduced. Thus, actor theory attacks the problem of distributed system design by offering an abstract conceptual and formal model for thinking about and proving theorems about distributed systems.

Many current multiprocessing systems allow *control* messages between processes, but most data is communicated by means of shared memory [11,12], which can easily become the main bottleneck in the system. Dataflow schemata [13] avoid this problem by binding control and data into messages known as *tokens*. This model allows for increased parallelism since any functional unit (e.g. a multiplier) which has received tokens on all of its inputs may proceed to compute with this data and output other tokens quite independently from the rest of the system. Thus dataflow computations can be represented by sets of *local* histories of input-output pairs of values [14]. However, the notion of a computation as a set of local histories of value pairs is not robust; modelling true non-determinism seems to require the greater structure of a single partial order for all the events.

The dataflow and other models [15,16] are also currently biased toward a *static* ensemble of functional units with a fixed topology connecting them; the actor model generalizes by allowing for the creation of new actors and for a dynamically changing topology during the course of a computation. Hence, actor messages must be able to convey the names of other actors as well as data so that new actors may be introduced to old. The power to create new actors and pass their names in message enables actor systems to implement *procedure calls* by sending a procedure actor a

---

1: Lamport [1] defines a distributed system as one in which "the transmission delay is not negligible compared to the time between events in a single process."

2: A *tightly coupled* system is one in which the transmission delay between parts is of the same order of magnitude as the cycle time of those parts.

message consisting of a parameter list and a *continuation*--the name of an actor which will receive the result message returned by the procedure.

In programming languages such as Simula-67 [17], Smalltalk [18], and CLU [19,20], 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, where an actor is the analogue of a class or type *instance*, but considers the added effects of parallelism.

In this paper, we argue that in a distributed computation, the notions of *local state*, *local time*, and *local name space*, should replace those of global state, global time, and global name space. This replacement of local notions for global notions is equivalent to the reformulation of computation in relativistic terms.

We present axioms for actor systems which restrict and define the causal and incidental relations between events in a computation, where each event consists of the receipt of a message by an actor, which results in the sending of other messages to other actors. These axioms make it unnecessary to postulate the existence and "fairness" of some underlying global *scheduler* [21] or *oracle* [22]. The actor 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 electrical busses, crossbar switches [12], ring networks [23], Ethernets [24], or Batcher sorting nets [25]. Programming systems [26] and machines [27,28] especially suited for actor computations have been designed. Simulations of complex control structures by patterns of message-passing have been worked out [26]. Incremental garbage-collection strategies for passive [29,28] and active [30] objects have been devised so that large actor computations can be efficiently mapped onto hardware of reasonable size. Computations which are partial orders of events have been used to develop specifications and proof techniques for modular synchronization primitives [31,32,33,34].

II. EVENTS AND ACTOR COMPUTATIONS

In a serial model of computation, computations are sequences of states, and each state in the sequence determines a next state for the computation by considering the program text in a Von Neumann computer or the specifications for the finite state control of an automaton in computability theory. In the actor model, we generalize the notion of a computation from that of a sequence of global states of a system to that of a partial order of events in the system, where each event is the transition from one local state to another. One interpretation of this partial order is that unordered events proceed concurrently. Other models whose computations are partial orders of events have been studied previously [35,36], but not with the generality considered here.

II.1 The Precedes Ordering

An *actor computation* is a pair $<\mathcal{E},$"-->"$>$, where $\mathcal{E}$ is a set of events and "-->" is a *strict partial order* on $\mathcal{E}$, i.e. "-->" is transitive and for no event E in $\mathcal{E}$ does E-->E. We say that event E_1 in $\mathcal{E}$ *precedes* event E_2 in $\mathcal{E}$ if E_1-->E_2; we say that E_1 is *concurrent with* E_2 if neither E_1-->E_2 nor E_2-->E_1. Intuitively, E_1-->E_2 only if there is some chain of physical actions from E_1 to E_2; i.e. "-->" is the weakest partial order consistent with this principle.

Actor computations are intended to be constructed inductively from some base by adding events in discrete steps. Although this base could be a finite set of *initial*[3] events, we posit a single initial event $E_\perp$ for simplicity. Again, since the computation is to be inductive, we postulate that in any

3: An *initial* event E in $\mathcal{E}$ is one which precedes all others in $\mathcal{E}$.

actor computation, the number of events between any two events is finite; i.e. given events E_1 and E_2 in an actor computation, the set $\{E|E_1\text{-->}E\text{-->}E_2\}$ is finite.[4] This property, which we call *discreteness*, means that the *immediate predecessors* and *immediate successors* of an event are well-defined.[5] Finally, we restrict the number of immediate successors of any single event to be finite.

These finiteness conditions do not rule out non-terminating actor computations; they only eliminate the possibility of "Zeno machines"--machines which compute infinitely fast. For example, consider a computer which can execute its first instruction in 1 second, its second in 1/2 second, its third in 1/4 second, etc. This machine could solve the "halting problem" by simulating a normal computer running on some input, and if the simulation were still running after 2 seconds, it could conclude that the simulated machine does not halt on that input.

These three restrictions (existence of a least element, discreteness, and finite immediate successors) implies that every actor computation has a monotonic[6] injection[7] into the set of non-negative integers $\mathbb{N}$. The existence of these mappings implies that actor computations are no more powerful than serial computations since the existence for every actor computation of a monotonic injection into $\mathbb{N}$ implies that every actor computation can be simulated on a serial computer and that simulation will terminate if and only if the actor computation terminates.

The fact that monotonic injections exist for every actor computation shows that actor computations are consistent with the "consider all shuffles" approach. However, we do not use the injections directly, we only note that they exist; all properties of actor computations are expressed in terms of partial orders, not monotonic injections.

II.2 The Activation Suborder

The action in the actor model consists of objects called *actors* sending *messages* to other actors, called the *targets* of those messages. We model only the receipt of these messages as events because this choice yields a simpler model, while still retaining those aspects of distributed computing systems of interest to us. Hence, an event E is the receipt of the message "message(E)" by the actor "target(E)". Upon receipt of this message in the event E, the target consults its *script* (the actor analogue of program text), and using its current local state and the message as parameters, sends new messages to other actors and computes a new local state for itself. The events in which those messages are received are said to be *activated* by the event E. Every actor computation $\mathcal{E}$ is *complete* in the sense that if an event E in $\mathcal{E}$ causes a message to be sent, there exists an event E' in $\mathcal{E}$ in which that message was received; i.e. every event which was activated has occurred.

Activation is the actor notion of causality and forms a sub-ordering[8] of the precedes relation; i.e. if event E_1 activates E_2, then E_1 precedes E_2. A crude analogy from physics may make activation more clear. A photon (message) is received by an atom (target) which puts it into an excited state. After a while, the atom gives off one or more photons and returns to its ground state.

4: This is a stronger condition than requiring that all chains between E_1 and E_2 be finite.

5: E_p is an *immediate predecessor* of E in the computation $<\mathcal{E},\text{"-->"}>$ if there does not exist an E' in $\mathcal{E}$ such that $E_p\text{-->}E'\text{-->}E$.

6: A function f from a partial order $<A,"\leq">$ to a partial order $<B,"\leq">$ is *monotonic* if $a_1 \leq a_2$ implies $f(a_1) \leq f(a_2)$.

7: A function f is *injective* if it is one-to-one.

8: A *suborder* of a strict partial order $<A,"<">$ is a strict partial order $<A,"<'">$, where "<'" is a transitively closed subset of "<".

These emitted photons may then be received by other atoms, and these secondary events are said to be activated by the first event.

What does this activation suborder look like? Since an event is the receipt of a message by an actor, the message must have been sent by some previous event--the activator event--and since only one message can be received in an event, there is at most one activator for the event. Now since there is a least element $E_\perp$ for the whole computation, it cannot have an activator; however, we will require every other event to have one. Hence, the activation suborder is a *tree* and for each event in the computation there is a unique finite path in this tree back to $E_\perp$.

II.3 The Arrival Suborder

All messages that are sent are eventually received by their targets. If many messages are en route to a single target, their simultaneous arrival may overload the target or cause the messages to interfere with one another. Therefore, to guard against these possibilities and ensure a kind of super-position principle for messages, we postulate an arbiter in front of every actor which allows only one message to be received at a time.[9] In terms of the precedes ordering on the events in a computation, this means that *all events with the same target are totally ordered.* We call the sub-relation defined by the restriction of the precedes relation to the set of events with a particular target actor the *arrival ordering* for that actor. Since there may be no necessary (causal) relation on two events other than that due to the effect of an arbiter, these arrival orders record the arbitrary decisions of the arbiters. Due to the discreteness of the precedes relation, every arrival order is isomorphic to a section[10] of $\mathbb{N}$.

II.4 Creation Events and the Precursor Suborder

Every actor is either one of a finite number of initial actors or is created during the course of a computation. Every created actor is created in a unique event called that actor's *creation event.* Several actors may be created in a single event; the only restriction is that only a finite number may be created in any single event. To satisfy our intuition about creation, we require that an actor's creation event precede the first event in which it receives a message.

From the arrival orderings and creation events, one can construct another subordering of the precedes ordering called the *precursor* suborder. For any event E in a computation, define precursor(E) to be 1) the immediate predecessor of E in the arrival order of the target of E, if one exists; else 2) the creation event of the target of E, if one exists; or else 3) $E_\perp$, if $E \neq E_\perp$. Precursors are thus defined for every $E \neq E_\perp$. The precursor suborder is then the transitive closure of the precursor relation and is a suborder of "-->" due to the restrictions on creation events. The precursor suborder is also a tree with $E_\perp$ as its root, and for every event in the computation there is a unique finite path in this tree back to $E_\perp$.

We have exhibited two distinct suborders of the precedes order which are independent in the sense that neither by itself encodes all the information in the precedes relation. However, if an event E_1 precedes an event E_2 then there is a finite chain of events through the activation and precursor orderings from E_1 to E_2; i.e. *the precedes order is precisely the transitive closure of the union of the activation and precursor suborders.* Thus, every actor computation is the transitive closure of the union of two finitary trees, and encodes the interplay of the two.

9: More parallelism requires additional copies of an actor; these copies will not lead to problems so long as the actor is "pure", a concept defined later.

10: A *section* of $\mathbb{N}$ is either $\{n \in \mathbb{N} | n \leq s\}$, for some $s \in \mathbb{N}$, or $\mathbb{N}$ itself.

II.5 Actor Induction

One reason for defining all of these orders and suborders on actor computations is to allow a form of computational induction for actor systems called *actor induction*. Actor induction is defined for all of the orders defined above, and when faced with proving some property of an actor computation, one chooses which one based on the complexity of the interactions on which the property depends. Properties of programs without "side-effects" can usually be proved using only "activation" induction, while properties of isolated databases can usually be proved using only "arrival" induction. Complex programs with side-effects will require full-blown "precedes" induction.

"Precedes" induction states that if $P(E_\perp)$ is true, and if P(activator(E)) and P(precursor(E)) imply P(E), then P is true of all events in the computation. Precedes induction can be trivially proved correct by considering any monotonic injection from a computation into $\mathbb{N}$ and using weak[11] induction. The other inductions and their proofs are similar.

II.6 Fork-Join Synchronization

Let us consider how a system of actors, when requested to calculate some value, can split up the load among themselves and combine the results to form the final answer. Actor F receives a request message to calculate the value $f(x)=h(g_1(x),g_2(x))$ and send the reply to the actor R. Actor F then creates an actor H and sends a message to G_1 requesting it to calculate $g_1(x)$ and send its reply to H. Simultaneously, F sends a message to G_2 requesting it to compute $g_2(x)$ and send its reply to H. Assuming that G_1 and G_2 are both total, H will receive messages from both G_1 and G_2 containing the values $g_1(x)$ and $g_2(x)$, respectively. Now because of the arbiter on H, these two messages are guaranteed to arrive at H in some order, but the order cannot be predicted. Therefore, the reply messages to H will also contain an indicator 1 or 2 to tell H that they were sent by G_1 or G_2, respectively. The script for H specifies that H is to wait until it has received replies from both G_1 and G_2, and when it has, it is to compute $h(g_1(x),g_2(x))$ and send that value to R; i.e. H uses its arrival order to accumulate information over time before replying. Thus, fork-join behavior can be implemented in an actor system.

11: *Weak induction* states that if P(0), and if P(i) for all i<k implies that P(k), then P(n) for all n in $\mathbb{N}$.

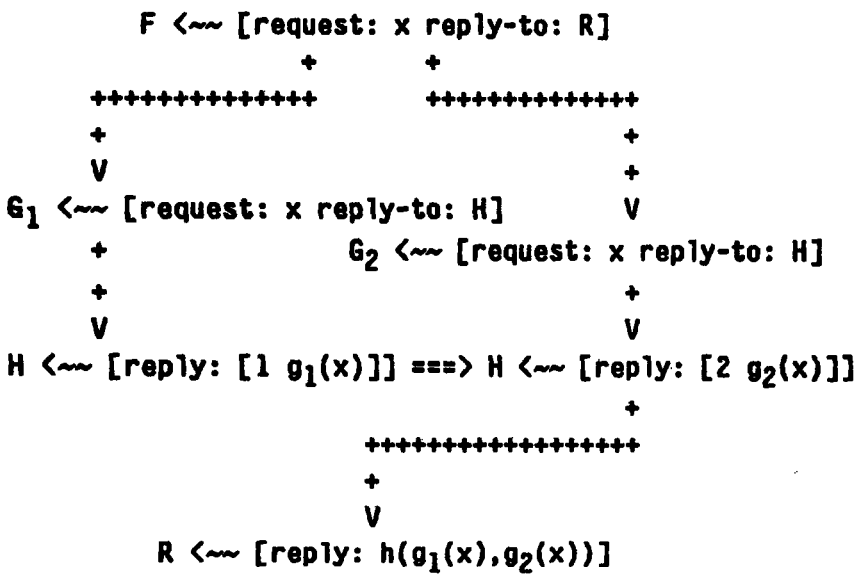

Figure 1. Diagram of Fork-Join Synchronization
"A <~~ M" means the event where actor A receives message M;
"E_1 ++> E_2" means E_1 precedes E_2 in the activation sub-ordering;
"E_1 ==> E_2" means E_1 precedes E_2 in H's arrival sub-ordering.

We have been asked "why use the arrival order to effect joins? Why not simply say that the calculations of $g_1(x)$ and $g_2(x)$ together activate the calculation of $h(g_1(x),g_2(x))$?" The reason is that activation is more than the sending of a "control token", more than a "goto"; it is the sending of a message which contains data. In this sense it is more analogous to a data flow token or a procedure call. Since we have declared that an event is the receipt of a message by an actor, and since a message can have but one event which activated its sending, we are forced to the conclusion that an event can have but one activator. Given this, the activation sub-ordering of the precedes relation is a tree, hence incapable of performing joins by itself.[12] Another reason for the reliance of joins on the arrival ordering is that for two concurrent and separated events to both activate a join event would require the conjunction of those two concurrent and separated events to be noted by some agent. This would allow information to flow by means other than the exchange of messages, a situation which is inconsistent with our philosophy of computation.

III. LOCALITY

Information in an actor computation is intended to be transmitted by, and only by, messages. The most fundamental form of knowledge which is conveyed by a message in an actor computation is knowledge about the existence of another actor. This is because an actor A may send a message to another actor B only if it "knows about" B, i.e. knows B's *name*. However, an actor cannot know an actor's name unless it was either created with that knowledge or acquired it as a result of receiving a message. In addition, an actor cannot send a message to another actor conveying names he does not know. In the next section we give restrictions which enable actor computations to satisfy these intentions.

12: *Goal-directed activities*, discussed later, can be used to hide the asymmetry of the actor join.

III.1 Actor Acquaintances

The order of arrival of messages at an actor defines a "local time axis" for the actor which is punctuated by its receipt of messages. Therefore, for any event along this local time axis we can define a *vector of acquaintances* for the actor, which encodes the names of other actors this actor knows at that instant. A *name* in this vector is just enough information to allow this actor to send a message to the actor denoted by the name.

In order to be precise about how an actor's vector of acquaintances may evolve over the course of its local time, we must define the notion of a *participant*. A participant in an event E is either the target of E, an acquaintance of the target[13] , an actor mentioned in the message of E, or an actor created in E.

An actor is given a finite initial vector of acquaintances when it is created. Every element of this initial vector must be a participant of the actor's creation event. Intuitively, an actor can initially know about no more than its parents, acquaintances of its parents, and its siblings. This acquaintance vector may change as a result of the messages the actor receives; when an actor receives a message, it may add to its acquaintance vector any name mentioned in the message (a message may mention only finitely many names). Of course, it is also allowed to forget acquaintances at any time. In the worst case (for storage), an actor could remember the names of every actor it knew about when created, as well as every name it was ever told in a message.[14] However, most actors remember very little of what they have been told. For example, a *cell*[15] remembers only one acquaintance, and *pure* actors like the number "3" or the function "+", by definition remember only their initial acquaintances.

As we have noted above, an actor at a given point in its local time is restricted in what other actors it can send messages to. In particular, an event E may activate an event E' only if 1) the target of E' is a participant of E; and 2) any actor mentioned in the message of E' must also be a participant of E. Through these *locality* restrictions, we have ruled out "broadcasting" protocols in which messages are sent to every actor in the system [1], because in a model with no global states which allows the creation of new actors, the phrase "every actor" is not well-defined.

An analogy can be made between actor computations and the computations of programs in block-structured languages like Algol and Simula [26]. The "free variables" of an Algol block are analogues of actor acquaintances; calling a procedure and returning a value both correspond to sending messages. The actor model is considerably more general, though, because actors may be sent messages asynchronously, may be created during a computation, and may gain new acquaintances and forget old ones.

III.2 Names, Paths, Ids

One may ask "in what form is this knowledge about other actors encoded?" In current distributed systems, each component is given a unique name in some global name space. For example, the PDP-11 Unibus[16] assigns a bus address to each component which other components on the bus use to access it. However, as computer networks grow and meet, each with its own naming conventions, the hope for such a global name space is doomed.

13: An acquaintance of an actor A is simply an actor B whose name is encoded in A's acquaintance vector.

14: One can prove by actor induction that the number of acquaintances remains finite.

15: More will be said about cells later.

16: *Unibus* is a trademark of the Digital Equipment Corporation.

Names for actors have at least two uses: sending the actor a message and determining whether two names denote the same actor. To send an actor a message, one must only be able to compute a path to it from the sending actor. Therefore, different actors may have different names for the same target. To determine identity, however, one must be able to take two names, compare them in some fashion, and determine whether the actors they denote are indeed the same actor. A naming system which assigns to every actor a unique *id*, independent of context, we call a *global id space*. A global id space has the properties that the id can be independent of any communication path and identity checking is trivial because the id uniquely determines the actor.

We contend that using a global id space to implement message passing is awkward to manage in a distributed system in which new actors are created and connectivity varies. This is because global directories have to be continually updated to permit the conversion of ids to paths.

Therefore, we prefer local naming for denoting actors in distributed systems. A *local name* is simply the encoding of a path from an actor to a target. Such a path need not be the only path to the actor denoted; system reliability could be increased by allowing an actor to convert one path to another, using its knowledge of system connectivity. Indeed, if paths are encoded as sequences of elementary connections in the network, then subsequences which converge on the same component are interchangeable. For some networks, one might even be able to define for each actor a canonical path to each other component, thus allowing that actor to determine trivially whether two paths converge on the same target. In general networks, however, one would like both unique ids and paths; identity of the targets of two paths could then be determined by using the paths to request and compare the unique ids of the actors they denote.

IV. CELLS

One of the simplest examples of an actor which depends on its arrival ordering for correct behavior is the *cell*. The cell in actor theory is analogous to the program variable in modern high-level programming languages in that it has a value which can be changed through assignment. This value is encoded as the cell's single, changeable acquaintance which is initialized to the name of some actor when the cell is created. A cell responds to two types of messages, "contents?" messages and "store!" messages. When a cell receives a request [contents? reply-to: c], the cell sends the name of its acquaintance to the actor c. When a cell receives a request [store! y reply-to: c], it memorizes new contents by making y its new acquaintance, forgetting its previous acquaintance, and then sending an acknowledge message to c.

IV.1 Busy Waiting and Fairness

Busy waiting is a synchronization mechanism used in some multiprocessing systems where the only communication channel between processors is through shared memory. To synchronize two processors with this method, one processor continually checks the contents of a shared cell for a change due to a store message from the other processor. When a change is detected by the first processor, it is synchronized with the second. This kind of synchronization is used when the first processor cannot depend on the second to "wake it up" when the second changes the cell's contents. One problem with this method is that unless the arbiter in front of the cell is "fair", the messages from one processor could be ignored and never be processed because the cell is *flooded* with messages from the other processor. Much effort has gone into the problem of specifications for fairness of the scheduling mechanism and elaborate algorithms for fair synchronization have been developed (see biblio. in [31,32]).

The actor model requires no such notion of scheduling or fairness to prove that such flooding is impossible. Why? A complete actor computation has no undelivered (unreceived) messages outstanding. Furthermore, the existence of a monotonic injection from the computation into $\mathbb{N}$ implies that between any two messages received by a cell, there are at most a finite number of other messages received. Hence, between the first "contents?" messages received by the cell from the first processor and the "store!" message from the second processor, there can be only a finite number of other messages received. Therefore, if time is measured by the arrival of messages at the cell, the first processor will eventually find that the contents of the cell have changed. However, the "length of time" required to synchronize is not bounded by any computable function, so although busy waiting works, it may not be a satisfactory synchronization method.

V. GOAL-DIRECTED ACTIVITIES

Since one of the goals of actor theory is to study patterns of passing messages, we must identify several common message types. The two most common types of messages are *requests* and *replies* to requests. A request message consists of two parts: the request itself, and the name of an actor which is to receive the reply. A reply to a request consists of a message sent to the actor named in the request for this purpose; 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 *goal-directed activity* corresponding to a request event E_Q in a computation to be the set of events which follow E_Q in the partial order but precede any reply E_R 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 activity(E_Q) = (E_Q--≥) ∩ ∪{--≥E_R|E_R is reply to E_Q}. Activities embody the notion of "goal direction", since we only include those events in an activity which contribute to a reply. Note that if no reply is ever made to the request E_Q in the computation, then the activity corresponding to E_Q is vacuous, since no event can contribute to a reply which does not occur.

If we let *concurrent activities* be those whose request events are concurrent, 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 and pure actors are freely copied to avoid arbitration bottlenecks, then goal-directed activities are *properly nested*, meaning that two activities are either disjoint, or one is a subset of the other.

We say that an activity corresponding to a request is *determinate* if at most one reply occurs regardless of the request, and *non-determinate* otherwise. Thus, determinate activities correspond to our usual notion of the subcomputations needed for subroutines. In fact, many interesting actor computations are activities; they start with a request and end with a reply. However, if no reply is forthcoming, the activity is by definition empty even though the computation may be infinite.

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 event* as a request which activates exactly one immediate reply, with no events intervening. Thus, the activity corresponding to a primitive event 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. Perhaps a suitable theory of *homomorphisms* of computations which map activities into single events can be worked out.

VI. CONCLUSIONS and FUTURE WORK

In creating a model for some real phenomenon, one must choose which features of the phenomenon to emphasize and which to ignore. The actor model ignores the sending of messages and concentrates instead on their arrival. The handling of messages upon their arrival is considerably more interesting because of the non-determinacy involved. Our model also ignores unreliabilities in the network--every message is eventually received--because we believe that this issue is orthogonal and separable from our current concerns. The model ignores the issue of "real time", i.e. time which can be measured, and concentrates only on the orderings of events. Thus, though messages are eventually received, there is nothing in the theory which specifies an upper bound on how long that might take. Finally, like all models, issues of representation are ignored in favor of issues of behavior; i.e. the representation of actors and messages has been left somewhat fuzzy.

We are currently working on a Scott-type fixed-point semantics [5,6,7] for actor systems in which completed computations are fixed points of some continuous functional over a conditionally complete semilattice. However, the elements of the lattice cannot be simply partial computations because of the arbitrariness in the arrival sub-ordering due to arbitration. They must encode all the different ways that this arbitrariness could have been resolved. Perhaps the work of Plotkin [37] or Lehmann [38] on non-deterministic computations can be extended to solve this problem of actor computations.

In a follow-on paper [39], we investigate very simple kinds of actors which use fork-join parallelism to compute mathematical functions, and show how the laws presented here can be used to prove that the objects computed by them are fixed points of continuous functionals in a complete semilattice.

ACKNOWLEDGMENTS

The research reported in this paper was sponsored by the MIT Artificial Intelligence Laboratory and the MIT Laboratory for Computer Science (formerly Project MAC) 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 MIT in July 1975.

This paper builds directly on the thesis research of Irene Greif [40,31]; the section on locality was influenced by Richard Steiger [27]. 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, Richard Stallman, Joe Stoy, Richard Weyhrauch, Steve Ward, and Bert Halstead. Valdis Berzins, Ben Kuipers, Henry Lieberman, Ernst Mayr, John Moussouris, Guy Steele, and Steve Zilles made valuable comments and criticisms which materially improved the presentation and content of this paper.

REFERENCES

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. C.A.R. Hoare. Communicating Sequential Processes. Dept. of Comp. Sci., The Queen's Univ., 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. R. Atkinson and C. Hewitt. 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. Applications of the Laws for Communicating Parallel Processes. IFIP Working Conf. on the Formal Desc. of Prog. Concepts, New Brunswick, Aug. 1977.
40. I. Greif and C. Hewitt. Actor Semantics of PLANNER-73 ACM SIGPLAN-SIGACT Conf., Palo Alto, Cal., Jan. 1975.