MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 190 June 1979

A FAIR POWER DOMAIN FOR ACTOR COMPUTATIONS

Will Clinger

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.

A FAIR POWER DOMAIN FOR ACTOR COMPUTATIONS

Will Clinger[1]

1. Abstract

Actor-based languages feature extreme *concurrency*, allow *side effects*, and specify a form of *fairness* which permits *unbounded nondeterminism*. This makes it difficult to provide a satisfactory mathematical foundation for their semantics.

Due to the high degree of parallelism, an oracle semantics would be intractable. A weakest precondition semantics is out of the question because of the possibility of unbounded nondeterminism. The most attractive approach, fixed point semantics using power domains, has not been helpful because the available power domain constructions, although very general, seemed to deal inadequately with fairness.

By taking advantage of the relatively complex structure of the actor computation domain **C**, however, a power domain **P(C)** can be defined which is similar to Smyth's weak power domain but richer. Actor systems, which are collections of mutually recursive primitive actors with side effects, may be assigned meanings as least fixed points of their associated continuous functions acting on this power domain. Given a denotation $A \in \mathbf{P(C)}$, the set of possible complete computations of the actor system it represents is the set of least upper bounds of a certain set of "fair" chains in A, and this set of chains is definable within A itself without recourse to oracles or an auxiliary interpretive semantics.

It should be emphasized that this power domain construction is not nearly as generally applicable as those of Plotkin [Pl] and Smyth [Sm], which can be used with any complete partial order. Fairness seems to require that the domain from which the power domain is to be constructed contain sufficient operational information.

---

1. MIT AI Lab room 910, 545 Tech Square, Cambridge MA 02139. Address until September 1979: c/o Control Data Corporation, 2800 East Old Shakopee Road, P O Box 1249, Minneapolis MN 55440.

2. Introduction

2.1 FIXED POINT SEMANTICS AND NONDETERMINISM

In a simple application of the now classic Scott-Strachey fixed point semantics ([Sc1,2], [MiSt]) a deterministic program may be taken to denote a partial function from inputs to outputs. Given a program, the semantics of the programming language in which it is written defines its associated continuous functional from partial functions to partial functions, and the meaning of the program is defined to be the least fixed point of that functional. This abstract definition is effective because the functional is straightforwardly obtained from the program syntax, and it effectively enumerates a sequence of approximations having the least fixed point as limit. Although the least fixed point may be infinite (considered as a set of input-output pairs), for a given input there is at most one output, and if that pair exists it will eventually be enumerated.

More generally, programs may be given meanings as functions from *input streams* to *output streams*, a stream being a (finite or infinite) sequence. Deterministic programs operating on streams have been given a fixed point semantics by Kahn [Ka].

When general parallelism with side effects is allowed, however, timing effects can introduce *nondeterminism*, so that a single input no longer determines at most a single output. This leads to considering sets of "possible" outputs, and to more complex domains involving the power set. These domains have been investigated by Plotkin [Pl] and Smyth [Sm]. Again the meaning of a program is the least fixed point of its associated continuous functional. In practice this least fixed point may be approximated by enumerating a sequence of subsets of the domain whose limit in the power domain is the least fixed point. Equivalently, a generating tree may be built having the least fixed point as the set of least upper bounds of its maximal paths.

Even for a given input, the set of possible outputs may now be infinite. In fact, when nondeterministic programs take streams as inputs, the set of possible output streams (for a fixed input) may even become uncountable, as in the case of an abstractly specified merge (fair or otherwise). Though the meaning of such a program may still be defined as the least fixed point of a certain continuous function, about the only sense in which that fixed point can be given effectively is by enumerating a countable set of partial output streams, from which the uncountable set of completed outputs may be obtained by taking all limits of sequences satisfying certain properties (for example, the sequences which lie along maximal paths of a generating tree).

The main shortcoming of this theory has been its inadequate treatment of fairness. Fairness is an essential feature of some languages, such as the actor-based languages PLASMA [HeAtk], ACT1 [HeAtt], and ETHER [Ko].

2.2 THE PROBLEM OF FAIRNESS

Roughly speaking, *fairness* is a property of programs that take inputs from two or more "processes" in such a way that each attempt by a process to provide input is bound to succeed sooner or later.

From a programming point of view, the ability to write fair programs is very desirable. For example, a *fair merge* makes possible simple solutions to such time-dependent nondeterministic systems as airline reservation systems. In some cases fairness is needed to prove termination or freedom from deadlock. Nonetheless programming languages which specify fairness or permit a fair merge are rare. The reason is a phenomenon known as *unbounded nondeterminism*, which two of the best semantic methods for nondeterministic languages, those of weakest preconditions and power domains, seem unable to deal with at present.

A program has unbounded nondeterminism if it always halts but for some input the number of states in which it "may" halt is infinite. An example is the following program written in Communicating Sequential Processes (CSP):

```
[X :: Z!stop() ||
 Y :: guard:boolean; guard := true;
      *[guard → Z!go(); Z?guard] ||
 Z :: n:integer; n := 0;
      continue:boolean; continue := true
      *[X?stop() → continue := false
        □ Y?go() → n := n + 1; Y!continue]
]
```

As the author of CSP points out, if the repetitive guarded command in the definition of Z were required to be fair, this program would have unbounded nondeterminism: it would be guaranteed to halt but there would be no bound on the final value of n. It is for this reason that he stops short of requiring fairness, stating only that "an efficient implementation should try to be reasonably fair and should ensure that an output command is not delayed unreasonably often after it first becomes executable." [Ho]

This caution is necessary because CSP's semantics uses power domain techniques with the Egli-Milner ordering, and as a result cannot deal with unbounded nondeterminism. In short, the above CSP program *cannot be proved to halt.*

Concurrent Processes is a more general abstract proposal also using power domain semantics but with a weak power domain ordering defined by Smyth [MiMi]. It cannot specify fairness either.

A proposal which avoids alternative input commands and polling (and thus

nondeterminism) altogether is that of [KaMcQ]; as a result the "fair merge" presented therein must assume for its correctness that both of its input streams are guaranteed to be infinite.

To clear up a possible confusion: even had CSP required fairness, no one believes the above CSP program when implemented on an actual machine would exhibit unbounded nondeterminism. On any given machine, its nondeterminism would be bounded. In the semantics, however, when its meaning is abstracted away from its possible fair implementations, its nondeterminism would have to be considered unbounded.

Therefore the presence of unbounded nondeterminism in the semantics in no way commits a theory in favor of the idea that a real machine could exhibit unbounded nondeterminism. When the formal semantics selects a set of possible computations as a program's denotation, the criterion of correctness for an implementation is that it guarantee that every physical computation correspond to a formal computation in that set; it need not and normally would not guarantee that every formal computation have a corresponding "possible" physical computation in the implementation. This matter is treated at length in [BaSi].

The usual way to achieve fairness in a mathematical semantics has been through the use of *fair oracles* [CaLe], [Ke], [Mi]. The weakest precondition approach has been unable to accomodate unbounded nondeterminism because the correctness condition of (for example) a fair merge would yield a discontinuous predicate transformer [Di]. On the other hand, all approaches using generating trees to model nondeterminism have been plagued by a problem described by Plotkin in [Pl]:

"Now the set of all initial segments of execution sequences of a given nondeterministic program P, starting from a given state, will form a tree. The branching points will correspond to the choice points in the program. Since there are always only finitely many alternatives at each such choice point, the branching factor of the tree is always finite. That is, the tree is finitary. Now Konig's lemma says that if every branch of a finitary tree is finite, then so is the tree itself. In the present case this means that if every execution sequence of P terminates, then there are only finitely many execution sequences. So if an output set of P is infinite it must contain [a nonterminating computation]."

As Smyth points out, the solution should lie in considering limits of paths in the generating tree rather than in defining a limit of cross sections (a set of all nodes at a particular depth) [Sm]. The idea is to exclude certain paths on the grounds that they represent unfair execution sequences.

One way to accomplish that might be to specify mathematically which paths are fair. This is what oracles accomplish; the properties required of the oracles

correspond to properties required of the valid paths of a certain (usually binary) generating tree. Since oracles are specified independent of the computations they control, however, they tend to introduce needless complexity which makes the semantics intractable. This is because they are oblivious to the common situation in which much of the indeterminacy they control makes no difference. This is no small matter for actor-based languages; since the number of actors "wanting to send" is unbounded and the number of messages an individual actor "wants to send" may be infinite, it would seem most natural to take oracles as sequences in $(\omega \times \omega)^{\omega}$ instead of the usual 2^{ω}, which is clearly undesirable.[1] It would be better to build the generating tree first, and then specify the fair paths, than to specify in advance the set of all possible fair paths and then find that most do not appear in the generating tree anyway.

In actor semantics the main actor computation domain is a slightly unconventional history domain which is a complete partial order (cpo) under the initial segment ordering. Fairness is achieved without oracles by using the incomplete subdomain of finite computations to construct the complete power domain, and then taking advantage of the operational information available in the computation domain. Programs are given meanings as least fixed points of a continuous function on the power domain. From such a fixed point the set of possible complete computations may be obtained by taking least upper bounds of all "fair" chains. This contrasts with the finitely generable sets obtained by taking least upper bounds of all "maximal" chains.

2.3 PREVIOUS WORK

Power domain constructions began with the papers of Plotkin [Pl] and Smyth [Sm]. Their constructions have been extended and simplified in [Le] and [SmPl].

Hoare's Communicating Sequential Processes is an outstanding example of a programming language that faces up to the problems of concurrency and side effects [Ho]. Its semantics has been given by Francez et al [Fr].

The mathematical model given by Milne and Milner as a precise notion of "processes" [MiMi] is very much the sort of thing we are here attempting to give for actors.

The style of semantics used in this paper, behavioral semantics, was developed by Irene Greif [Gr1,2]. Akinori Yonezawa suggested behavioral

---

1. By inventing the notion of an *instantaneous schedule*, Henry Baker was able to reduce this to ω^{ω} [Ba].

equations in his thesis [Yo]. The behaviors of individual actors were defined by Hewitt and Attardi while giving the semantics of ACT1 [HeAtt]. The actor computation domain was defined by Greif [Gr1], while the ordering laws were refined through a sequence of papers [HeBa1,2], [Cl]. The fairness problem was attacked by Baker, who incorporated oracles into a behavioral semantics for actor systems [Ba].

3. The Actor Model

3.1 INFORMALLY SPEAKING

The actor model is perhaps best motivated by the prospect of highly parallel computing machines consisting of dozens or even hundreds of independent monoprocessors, each with its own local memory and communications processor, communicating via a packet-switched network in a system similar to some of today's distributed computer networks [Ha], [He3].

In the model, *actors*[1] are *deterministic* computational agents which communicate by sending *messages*. Each message sent is guaranteed to *arrive* at its *target* actor, at which time it is placed in a *queue* associated with the target to await processing. The model assumes nothing else about the mechanism of message transmission, so that the order of arrival is *nondeterministic*.[2] When an actor *accepts*[3] the next message in its queue for processing, it *locks* and accepts no more messages from the queue until (if ever) it finishes with that message.

Messages are processed in the order in which they arrive. The processing of a message may involve (1) sending out a (possibly infinite) set of (immediately) *activated* messages and (2) a change of *local state* within the actor. When (if) an actor finishes processing a message, it *unlocks* and accepts the next message in its queue. If there are none, it waits until there are.

---

1. Actually *primitive serialized* actors. *Unserialized* actors may be constructed using serialized actors and actor *creation* [HeAtk].
2. Unboundedly so.
3. "Receives" in the terminology of [He2]. I find "accepts" less confusing.

3.2 THE BEHAVIOR DOMAIN

Let A be the set of *actors*, and M the set of *messages*.

An abstract actor is described by its *behavior* [HeAtt]. The behavior specifies what the actor does whenever it receives a message: it may change state, and it may send out messages to other actors. This suggests that the behavior of an actor a is a function

$$b^a: \Sigma^a \to (M \to (\Sigma^a \times \textbf{multisets}(A \times M)))$$

where Σ^a is the set of *local states* of a. Since the only purpose of local states is to index the next behavior, though, the behavior domain is defined recursively as

$$B = [M \to (B \times \textbf{multisets}(A \times M))].$$

B may be thought of as the set of trees of height ω, with an unlabelled root node, non-root nodes labelled by multisets in $A \times M$, and such that each node has exactly one outgoing arc labelled by m for each message m.

Let **next** and **events** be the natural projections

$$\textbf{next}: (B \times \textbf{multisets}(A \times M)) \to B$$
$$\textbf{events}: (B \times \textbf{multisets}(A \times M)) \to \textbf{multisets}(A \times M).$$

For example, a *pure actor* never changes state and so $\textbf{next}(b^a(m)) = b^a$ for all messages m.

Behaviors are normally specified using a programming language. Different languages allow different subsets of the behavior domain. In ACT1, for example, $\textbf{events}(b(m))$ is always finite [HeAtt], while in ETHER it is can be infinite [Ko]. The laws of locality [HeBa1,2], which are abstract counterparts of scoping rules, are examples of the sort of restrictions a language may reasonably place on behaviors.

The simplest behavior is the pure passive behavior

$$\textbf{dead}: [?msg] \mapsto (\textbf{dead}, \varnothing)$$

which means the actor is good only for absorbing messages. When an actor system is being modelled, all actors outside the system are given the passive behavior **dead** so that the histories produced show for them only the stream of messages they receive. These should be thought of as outputs.

A programming language, given a program written in the language, provides

a behavior for each actor. That is, a programming language (L, β) consists of a description language L and a map β: $L \rightarrow [A \rightarrow B]$. This means a program specifies behaviors for all actors that could possibly be required for its execution, whether existing initially or created in the course of computation; it thereby sidesteps the actor creation issue by assuming that all actors exist initially with defined behaviors.

3.3 THE COMPUTATION DOMAIN

A (partial) computation is a labelled graph as below. Simple examples may be found in sections 5.3-5.5.

Definition. *The set* C *of* (*partial*) computations *is the set of structures*

$$c = (E, T, M, \text{—}act\text{→}, Arr)$$

where

1. E *is the set of* event nodes *of* c.

2. T: $E \to$ A *gives the target of each event.*

3. M: $E \to$ M *gives the message of each event.*

4. $\text{—}act\text{→}$ *is the* activation ordering: *a strict partial order on* E *such that*
$(e_1 \text{—}act\text{→} e \wedge e_2 \text{—}act\text{→} e$
$\wedge \neg \exists e' (e_1 \text{—}act\text{→} e' \text{—}act\text{→} e \vee e_2 \text{—}act\text{→} e' \text{—}act\text{→} e))$
$\supset e_1 = e_2$.

(*That is, an event* e *has at most one immediate* activator, *which is written* activator(e) *if it exists. If* activator(e) *does not exist, then* e *is said to be* external.)

5. $Arr = \{\text{—}arr_a\text{→} \mid a \in \text{A}\}$ *is the set of* arrival orderings:
$\forall a \in \text{A}$ $\text{—}arr_a\text{→}$ *is a strict total order on* $\{e \in E \mid \text{T}(e) = a\}$.
($\text{—}arr_a\text{→}$ *is the* arrival ordering of a.)

and the following ordering laws *hold, where* $\to$ *is the* combined ordering *defined as the transitive closure of* $\text{—}act\text{→} \cup (\cup Arr)$:

Law of Strict Causality (LSC). $\forall e \in E \ \neg e \to e$.

Law of Countability (LC). E *is countable.*

Law of Finite Predecessors (LFP). $\forall e \in E \ \{e' \in E \mid e' \to e\}$ *is finite.*
□

(Computations $c_1 = (E_1, T_1, M_1, \text{—}act\text{→}_1, Arr_1 = \{\text{—}arr_a\text{→}_1 \mid a \in \text{A}\})$ and $c_2 = (E_2, T_2, M_2, \text{—}act\text{→}_2, Arr_2 = \{\text{—}arr_a\text{→}_2 \mid a \in \text{A}\})$ are *isomorphic* if there exists a one-one correspondence α: $E_1 \to E_2$ such that $T_1(e) = T_2(\alpha e)$,

$M_1(e) = M_2(\alpha e)$, $e \text{ —}act\rightarrow_1 e' \leftrightarrow \alpha e \text{ —}act\rightarrow_2 \alpha e'$, *and* $e \text{ —}arr_a\rightarrow_1 e' \leftrightarrow \alpha e \text{ —}arr_a\rightarrow_2 \alpha e'$ for all $e, e' \in E_1$, $a \in$ A. This isomorphism is unique if it exists. We assume that isomorphic computations are identified in C.)

Each event node e represents a specific arrival instance of the message $M(e)$ at the target $T(e)$. The messages with a given target a are totally ordered by its arrival ordering. If e is not external, then it is a direct result of its activator event so that $(T(e), M(e)) \in$ **events**(b(M(**activator**(e)))), where b is the behavior of T(**activator**(e)) at the time of the event **activator**(e) (or more correctly, when it accepts M(**activator**(e)) for processing). External events model the arrival of messages originating outside the actor system under consideration; they should be thought of as inputs.

Note that there are no "sending events". Since messages are processed in the order of their arrival, all necessary information about senders may be obtained from the activation ordering and the arrival orderings of senders.

The three ordering laws are equivalent to an axiom of realizability in global time [Cl].

For a given computation, let e_i^a ($i \in \omega$) be the ith event in the arrival ordering of a, provided it exists. The *local history* of a is the (finite or infinite) sequence $\{M(e_i^a)\}_{i\in\omega}$.[1] Given computations $x, y \in$ C and events e_x of x and e_y of y such that the two events have the same target a, and $e_x = e_n^a$ in x and $e_y = e_n^a$ in y, a has the same initial local history up through $e_x = e_y$ in both x and y iff

$$\{M(e_i^a) \mid e_i^a = i^{\text{th}} \text{ event in } x \text{ with target } a\}_{i\le n}$$
$$= \{M(e_i^a) \mid e_i^a = i^{\text{th}} \text{ event in } y \text{ with target } a\}_{i\le n}.$$

That is, the sequence of messages arriving at the actor a is the same in both computations at least up to and including the events e_x and e_y.

Let $c_1, c_2 \in$ C, $c_1 = (E_1, T_1, M_1, \text{—}act\rightarrow_1, Arr_1)$, $c_2 = (E_2, T_2, M_2, \text{—}act\rightarrow_2, Arr_2)$. The following defines what it means for c_1 to be a *finite initial segment* of c_2.

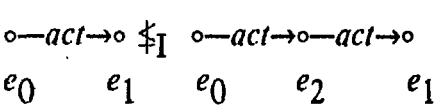
---

1. $\{\cdot\}_{i\le n}$ and $\{\cdot\}_{i\in\omega}$ denote sequences, while $\{\cdot \mid i \le n\}$ and $\{\cdot \mid i \in \omega\}$ denote multisets.

Definition. $c_1 \leq_I c_2$ *iff*

1. E_1 *is finite.*
2. $E_1 \subseteq E_2$ *(More precisely, the events of* c_1 *can be identified with a subset of the events of* c_2 *in such a way that 3-7 hold; this identification is unique if it exists.)*
3. *Targets and messages are the same in* c_1 *and* c_2.
   (That is, $\forall e \in E_1\ T_1(e) = T_2(e)$ *and* $M_1(e) = M_2(e)$.)
4. *The activation ordering of* c_1 *is a full subordering of that of* c_2.
   (Formally $\forall\ e_1, e_2 \in E_1\ \ e_1 \text{ —}act\text{→}_1 e_2 \leftrightarrow e_1 \text{ —}act\text{→}_2 e_2$.)
5. *Immediate activators in* c_1 *are the same as in* c_2.
   $(\forall\ e_1 \in E_1\ \forall\ e_2 \in E_2\ e_2 \text{ —}act\text{→}_2 e_1 \supset e_2 \in E_1$.)
6. *The arrival orderings of* c_1 *are full suborderings of those of* c_2.
   $(\forall\ a \in A, \text{ —}arr_a\text{→}_1 \in Arr_1, \text{ —}arr_a\text{→}_2 \in Arr_2$
   $e_1 \text{ —}arr_a\text{→}_1 e_2 \leftrightarrow\ \ e_1 \text{ —}arr_a\text{→}_2 e_2$.)
7. *Local histories in* c_1 *are initial segments of those of* c_2.
   $(\forall\ e_1 \in E_1\ \forall\ e_2 \in E_2\ e_2 \text{ —}arr_a\text{→}_2 e_1 \supset e_2 \in E_1$.)

$c_1 \leq_I c_2$ means c_1 can be extended to c_2 by adding events, the idea being that c_1 is a possible "snapshot" of c_2 on the way to being complete. In other words, c_2 can be constructed from c_1 by adding external events and events *activated* by events in c_1. Condition 5 states that all activation predecessors of an event e in c_2 are in c_1 if e is, and is illustrated by



Just as conditions 4 and 5 ensure that the activation orderings match, conditions 6 and 7 ensure that all the arrival orderings match.

$(C, \leq_I)$ is a *complete partial order* (cpo), meaning that every directed set (having pairwise least upper bounds) has a least upper bound. The least element of C, $\perp_I$, has no event nodes. Rather than representing a computation which has not yet terminated, $\perp_I$ represents a computation not yet *started*.

4. The Power Domain

Readers may wish to refer to the simplified examples in sections 5.3-5.5 while reading this chapter.

4.1 DEFINITION

An element of C is *finite* iff its set of event nodes is finite. Let $\mathbf{C_{fin}}$ be the set of finite elements of C. C is the *ω-completion* of $\mathbf{C_{fin}}$, and $\mathbf{C_{fin}}$ is a *basis* for C, in that C is the set of least upper bounds of chains in $\mathbf{C_{fin}}$. An element of C may be identified with the set of its finite initial segments.

A finite partial computation c may be interpreted as providing a finite amount of partial information about the complete computation. Specifically, c codes the fact that the computation when complete will have c as an initial segment. In Smyth's weak power domain construction, a finite set F of finite partial computations would code the knowledge that the computation when complete will have (at least) one of the elements of F as an initial segment [Sm]. With this interpetation, only the minimal elements of F make any difference, so that F may as well be supposed to consist solely of elements which are both minimal and maximal in F.

For reasons soon to become clear, we instead require F to be closed under $\leq_I$-predecessor. We lose no information by so doing, since we may interpret F so that the completed computation must extend a maximal element of F.[1]

We now define the power domain **P(C)** as the order completion of $\mathbf{C_{fin}}$ ([HeSt], page 180).

Definition. **P(C)** *is the set of all subsets A of* $\mathbf{C_{fin}}$ *such that A is closed under $\leq_I$-predecessor; that is, if $y \in A$ and $x \leq_I y$ then $x \in A$.*

Definition. *Let $A, B \in$* **P(C)**. *$A \sqsubseteq B$ if and only if $A \subseteq B$.*

---

1. Milne and Milner use the *right closure* of the finite sets such as F as canonical elements from which they obtain their power domain (though in [MiMi] they go directly to the power domain). This is more convenient for their purposes. In effect we use the left closures of finite sets containing only minimal elements to form our power domain. This is more convenient for defining fair chains.

$(\mathbf{P(C)}, \sqsubseteq)$ is a complete lattice.

The interpretation of $A \in \mathbf{P(C)}$ is that it is the set of all possible finite initial histories of any actor system it may represent. This does not completely determine the interpretation, however, since this set is the same for both fair and unfair interpretations. In other words, fairness and unfairness differ only "at infinity", so that a choice between fairness and unfairness can only be made when the set of "completed computations" generated by A is defined.

This set may be defined in two ways. In the first way, which permits unfair implementations in which some messages sent out may never be received, the set of "completed computations" is a *finitely generable* (fg) set, which implies that if it is infinite then it contains nonterminating computations.[1] Fg sets are usually the only sets considered [deB], [Fr], [MiMi], [Pl], [Sm], so that specifying fairness is impossible from the outset. Actor theory requires that all messages sent be received (which is a form of fairness), so we use a second definition.

The unfair definition is presented first, both to relate actor semantics to other approaches and to introduce the definitions adopted by actor theory.

Definition. *Let* $A \in \mathbf{P(C)}$. *A chain* $x_0 \leq_I x_1 \leq_I x_2 \leq_I x_3 \leq_I \cdots$ *in* A *is* maximal *iff for all (partial) computations* y *in* A, *if* y *is related to all of the* x_i *(that is,* $\forall\, i \in \omega \;\; x_i \leq_I y \;\vee\; y \leq_I x_i$*) and for some* i x_i *has all the external events of* y, *then there exists an* n *such that* $y \leq_I x_n$.

The idea is to ensure that the chain does not "settle down" with $x_i = x_{i+1} = x_{i+2} = x_{i+3} = \cdots$ unless x_i is a completed computation. Primitive actors are deterministic, so there cannot exist a $y \in A$ bigger than all elements of the chain unless y has more inputs or the chain "settles down" before the computation is complete.

Definition. *If* $A \in \mathbf{P(C)}$, *then*
fg-complete$(A) = \{\sqcup\{x_i\}_{i\in\omega} \mid \{x_i\}_{i\in\omega}$ *is a maximal chain in* $A\}$.

fg-complete(A) is a finitely generable subset of **C**. It differs from the correct set of completed computations only because it can contain the limits of unfair chains, in which some messages are sent but never received.

---

1. The term "finitely generable" refers to the fact that such a set is defined as the set of limits of chains along infinite paths of some finitary generating tree; see the quotation from Plotkin in section 2.2.

Now to state the definitions giving the fair interpretation of $A \in \mathbf{P(C)}$, which is the interpretation adopted for actor semantics. Recall that $A \in \mathbf{P(C)}$ is interpreted to be the set of all possible finite initial histories of any actor system it may represent. Not all elements of **P(C)** are possible denotations of actor systems, however, because nothing in the definition enforces determinism of actor behavior.

Definition. *$A \in \mathbf{P(C)}$ has the* determinism property *iff, for all computations $x, y \in A$ and events e_x of x and e_y of y such that e_x and e_y have the same target a and a has the same initial local history up through $e_x = e_y$ in both x and y, there exists $z \in A$ such that $x \leq_I z$ and*

$$\{(T(e), M(e)) \mid e \text{ is an event of } y \text{ and } \mathbf{activator}(e) = e_y\}$$
$$\subseteq \{(T(e), M(e)) \mid e \text{ is an event of } z \text{ and } \mathbf{activator}(e) = e_x\}$$

(as multisets).

Definition. *A chain $x_0 \leq_I x_1 \leq_I x_2 \leq_I x_3 \leq_I \cdots$ in A is* fair *iff for all computations y in A and events e_y of y giving the same initial local history as an event e_x of x_i having the same target as e_y there exists an n such that*

$$\{(T(e), M(e)) \mid e \text{ is an event of } y \text{ and } \mathbf{activator}(e) = e_y\}$$
$$\subseteq \{(T(e), M(e)) \mid e \text{ is an event of } x_n \text{ and } \mathbf{activator}(e) = e_x\}$$

(as multisets).

Definition. *If $A \in \mathbf{P(C)}$ has the determinism property, then*
$\mathbf{complete}(A) = \{\sqcup\{x_i\}_{i\in\omega} \mid \{x_i\}_{i\in\omega}$ *is a fair chain in A}.

The idea is that primitive actors are deterministic, so if on one occasion an actor sends out a certain set of messages to targets upon receiving a given message in a certain state, then on all other occasions it must send out that same set when in the same state it receives the same message. All messages sent out arrive at their targets, by actor fairness.

When the question of fairness does not arise, $\mathbf{complete}(A) = \mathbf{fg\text{-}complete}(A)$.

The following theorem shows that the denotation of an actor system may be recovered from the set of completed computations constructed from it, so that A and $\mathbf{complete}(A)$ contain the same information.

Theorem. *Suppose A has the determinism property, and $A' = \{x \in \mathbf{C_{fin}} \mid \exists\, c \in \mathbf{complete}(A)\ x \leq_I c\}$. Then $A' = A$.*

Sketch of proof: $A' \subseteq A$ is trivial. Let $x \in A$. Must construct a fair chain in A beginning with x. Order the events of x, and, for each event e_i^a, order the elements of the multiset

$$\cup_{y \in A}\{(T(e'), M(e')) \mid \textbf{activator}(e') = e, T(e) = a, \textit{ and } a \textit{ has the same initial local history up through } e = e_i^a \textit{ in both } x \textit{ and } y\}$$
$$- \{(T(e'), M(e')) \mid \textbf{activator}(e') = e_i^a\}$$

(where e_i^a is the i^{th} event in the arrival ordering of a). Call this set the set of *pending events* for the event e. Fix its order for all time. Build new elements of the chain by adding, in order, the first pending event for each event in x. When that has been done, order the set of pending events for each new event (defined using the above definition but with x replaced by the last computation yet produced). Add the new events to the old order on "already happened" events. Do the round robin again, adding the next pending event for each event. Continuing in this manner produces a fair chain. □

4.2 INTERPRETING ELEMENTS OF P(C) AS FUNCTIONS

This section and the next are far more tentative than the rest of this paper. Most methods given have been worked out only informally.

Suppose an actor system is given, with meaning $A \in \mathbf{P(C)}$. An *input event* is an external event whose target is an actor in the actor system. An *output event* is an event whose target is not in the actor system. An *input stream* is the sequence of input events in the arrival ordering of some actor in the system. The actor system defines a function from the input streams of its actors to sets of possible multisets of output events. This function is definable from A.

If the actor system takes a single input, modelled by a single external event e, then the set of all possible finite initial segments of computations on that input may be obtained by restricting A to those partial computations which have e as the sole external event or have no external events at all (ie $\perp_I$). The set of possible completed computations may be had by applying **complete** to this restriction. Examples are found in sections 5.3-5.5.

For the actor system to have streams as output rather than multisets, there must be synchronization between the actor system and its environment. This leads to rules for combining actor systems.

4.3 COMBINING ELEMENTS OF P(C)

Brock and Ackerman have described a technique for combining modules described by sets of partial orders representing histories ([BrAc], appendix). A variant of their technique may be used to combine actor systems with meanings $A, B \in \mathbf{P(C)}$ either by applying the variant directly to A and B to form $C \in \mathbf{P(C)}$ or to **complete**(A) and **complete**(B) to form **complete**(C).

The main reason their technique is not usable directly is that they assume that each communication port of a module can be connected to at most one other module. It is possible for two actor systems to send messages to the same actor, so this assumption fails for the actor model. As a result, the actor version of their technique must match subsequences of input events with subsequences of output events in all possible ways, while their version simply matches port histories. The simplification gained by their restriction on ports suggests that it may be useful for building structured programs.

Briefly, the technique consists of forming all ordered pairs $(x, y) \in A \times B$ and rejecting immediately those pairs for which arrival orderings in x are incompatible with those in y. For those ordered pairs that remain, output events of x must be identified with input events of y in all possible ways, and vice versa. The two partial computations are then merged into one.

Several special cases simplify this procedure.

If an actor system contains only one actor, then the denotation of the actor system in **P(C)** contains exactly the same information as the behavior of the actor. In theory we could do without the behavior domain and use **P(C)** instead. That would be a step in the wrong direction, of course, since we want simplicity rather than complexity.

Barber and Simi suggest that actor systems should be constructed modularly with a *receptionist* for each actor system [BaSi]. The receptionist is a designated actor which is the target of all inputs to the module, and which sends all outputs from the module. The other actors in the system are protected from external events, so the receptionist serves as the sole link between the module and other actor systems. The protection would be expressed by scoping rules in a programming language. In the formal semantics scoping would be reflected by additional axioms such as the laws of locality stated by Hewitt and Baker [HeBa1,2].

When an actor system with a receptionist has a deterministic input-output behavior, the actor system may be considered to be a single actor. The precise transformation of its meaning in **P(C)** to its behavioral specification in B has not yet been worked out.

This cannot be done in the present semantics when the actor system is nondeterministic, so that the meanings of such systems must remain in **P(C)**.

Note that the receptionist is available to all modules, contrary to the suggestion that ports be used only for communication between two fixed modules. We do not know very much yet about writing parallel programs, and it is far too early to define the best ways of structuring such programs.

5. Meanings as fixed points

5.1 TERMINOLOGY

Let P be a program. $\beta(P)$ is then an assignment of behaviors to actors. For $a \in A$, let $b^a = (\beta(P))(a)$ be the behavior of a. Let $c = (E, T, M, \text{—}act\text{→}, Arr = \{\text{—}arr_a\text{→} \mid a \in A\})$ be a computation in C. Let

$$e^a_0 \text{—}arr_a\text{→} e^a_1 \text{—}arr_a\text{→} e^a_2 \text{—}arr_a\text{→} \cdots$$

be the successive events in the arrival ordering of a. The successive behaviors of a are then given by

$$b^a_0 = b^a$$
$$b^a_{n+1} = \mathbf{next}(b^a_n(M(e^a_n))).$$

The event e^a_n is *unexpanded* iff it has no activation successors and $\mathbf{events}(b^a_n(M(e^a_n)))$ is nonempty. An event is *expanded* iff it is not unexpanded.

The event e^a_n is *expanded consistently* iff it is expanded and $\{e \mid \mathbf{activator}(e) = e^a_n\} \subseteq \mathbf{events}(b^a_n(M(e^a_n)))$. Iff equality holds, it is *expanded completely*. Events which are expanded but not consistently are *expanded inconsistently*.

A computation is *consistent* if none of its events are expanded inconsistently. It is *complete* if all its events are expanded completely. Note that these definitions are functions of the behavior assignment specified by the actor system.

Already the meaning of the actor system could be defined simply as the set of complete computations of C. The next section shows how to go about constructing this set.

5.2 DEFINITION OF τ_*: P(C) → P(C)

This section defines a continuous function τ_*: **P(C)** → **P(C)** whose least fixed point is a suitable denotation for the actor system. τ_* is defined pointwise from a function τ: $\mathbf{C_{fin}}$ → **P(C)**.

First to define τ: $\mathbf{C_{fin}}$ → **P(C)**.

Let $c \in \mathbf{C_{fin}}$ with notation as in the last section. τ will produce a set of (finite initial segments of) computations which are like c except one new event has been added. If c is complete, the new events will all be external.

Let H be the set of events of c which are not expanded completely. For $e_n^a \in H$ define the multiset of pending events with activator e_n^a as

$$P_n^a = \mathbf{events}(b_n^a(M(e_n^a))) - \{(T(e), M(e)) \mid \mathbf{activator}(e) = e_n^a \textit{ in } c\}$$

and the multiset of pending events with activator tags as

$$P = \cup_{e_n^a \in H}\{((a', m'), e_n^a) \mid (a', m') \in P_n^a\}$$
$$\cup \{((a', m'), \perp) \mid (a', m') \in \mathbf{A} \times \mathbf{M}\}$$

where the second term represents the external events that could be added.

Let $c((a', m'), e)$ be the computation obtained from c by adding one new event node with target a', message m', and activator e (or no activator if $e = \perp$). Formally

$$c((a', m'), e) = (E \cup \{e'\}, T', M', \text{—}act\text{→}', Arr')$$

where e' is a new event node and

$$T'(e_1) = T(e_1) \text{ if } e_1 \in E$$
$$a' \text{ if } e_1 = e'$$

$$M'(e_1) = M(e_1) \text{ if } e_1 \in E$$
$$m' \text{ if } e_1 = e'$$

$$e_1 \text{—}act\text{→}' e_2$$
$$\leftrightarrow (e_1, e_2 \in E \wedge e_1 \text{—}act\text{→} e_2)$$
$$\vee (e_2 = e' \wedge (e_1 = e \vee e_1 \text{—}act\text{→} e))$$

$$e_1 \text{—}arr_a\text{→}' e_2$$
$$\leftrightarrow (e_1, e_2 \in E \wedge e_1 \text{—}arr_a\text{→} e_2)$$
$$\vee (e_2 = e' \wedge e_1 \in E \wedge T(e_1) = a = a').$$

Define τ by

$$\tau(c) = \{x \in \mathbf{C_{fin}} \mid \exists\, ((a', m'), e) \in P \textit{ such that } x \leq_I c((a', m'), e)\}.$$

If c is consistent, every element of $\tau(c)$ is also consistent.

τ: $\mathbf{C_{fin}} \to \mathbf{P(C)}$ is monotonic and continuous; note, however, that continuity is vacuous because $\mathbf{C_{fin}}$ has the discrete topology.

Define τ_*: $\mathbf{P(C)} \to \mathbf{P(C)}$ by

$$\tau_*(A) = \cup_{x \in A} \tau(x).$$

The following facts are immediate.

Fact. *τ_* is monotonic and continuous.*

It therefore has a least fixed point, since $\mathbf{P(C)}$ is a complete lattice.

Fact. *The least fixed point of τ_* has only consistent elements.*

Fact. *The least fixed point of τ_* has the determinism property.*

Theorem. *If A is the least fixed point of τ_*, then* **complete**(A) *is the set of all complete computations in* C *relative to the given behaviors.*

Sketch of proof. Every element of **complete**(A) is consistent since every element of A is consistent. Every element of **complete**(A) is complete by the definitions of fair sequence, τ, and τ_*.

If $c \in \mathbf{C}$ is complete, let E be the events of c and let g: $E \to \omega$ be a one-to-one function onto an initial portion of the integers (in other words, onto ω itself if E is infinite, or onto **n** if c has n events) which preserves the combined ordering $\to$ of c. (The existence of such a function is proved in [Cl].) For $n \in \omega$, let c_n be the initial segment of c having events $\{g^{-1}(i) \mid i < n\}$. Then $c_0 = \perp_{\mathbf{I}}$ and $c_{i+1} \in \tau(c_i)$ so, for all i, $c_i \in A$; furthermore $\{c_i\}_{i \in \omega}$ is a fair sequence in A, and so $\sqcup\{c_i\}_{i \in \omega} \in$ **complete**(A). □

5.3 EXAMPLE 1: INFINITE LOOP

This trivial example is included for comparison with the next example, and to introduce notation used in all three examples.

As in the next two examples, the actor system being considered contains only one actor a. All other actors, such as *user*, have the passive behavior **dead** defined in section 3.2. That is, they simply ignore all incoming messages:

$$\textbf{dead}\colon\ [?msg] \mapsto (\textbf{dead}, \varnothing)$$

To review the notation for behaviors, this means that upon receiving any message, the new behavior is **dead** and the set of messages sent out to targets is empty.

For clarity, if m is a message sent to a target actor t, we usually write $[\![t \leftarrow m]\!]$ to indicate the ordered pair (t, m).

In this first example, a initializes itself to a state 0 when it receives a go instruction, and sends itself an increment instruction. When it receives an increment instruction in state n, it enters state $n+1$ and sends itself another increment instruction. Were it ever to receive a halt instruction, it would tell *user* its current state. Its initial behavior is b given by

b: [go] $\mapsto$ (b_0, {[[$a \leftarrow$ [add1]]]})

b_i ($i \in \omega$): [add1] $\mapsto$ (b_{i+1}, {[[$a \leftarrow$ [add1]]]})
[halt] $\mapsto$ (**dead**, {[[*user* $\leftarrow$ [i]]]})

When behaviors are given in this manner, it is intended that all messages not provided for are simply ignored.

It is now an easy matter to compute the function τ_* associated with this very simple actor system. We will only discuss computations having exactly one external event, of the form [[$a \leftarrow$ [go]]], for which we need only $\perp_I$ and the partial computations having this one external event. We therefore pretend that the least fixed point of τ_* contains only such computations. We will similarly simplify the examples in the next two sections.

When computations are represented graphically, activation links will be indicated by solid arrows. The activation order is the transitive closure of such links. Arrival orderings will be indicated by dashed vertical lines, labelled by the name of the actor which is target of all events in the arrival ordering. Higher events precede lower events in these arrival orderings, which is to say time flows downward on the page. Events may then be labelled simply by their messages, since the targets are clear from the labels of the arrival ordering in which they appear.

The least fixed point of τ_* (ignoring the computations having more external events than we care about) is then A as given below. In none of the computations we are considering does a halt instruction ever arrive, so the infinite loop is the only completed computation. The unfair definition of the set of completed computations gives the same result for this example.

In the next example, the unfair definition gives a different result from the fair definition. In fact, if $B \in \mathbf{P(C)}$ has A as a subset, then **fg-complete**(B) must contain this infinite loop. This is not true of **complete**(B), as shown by the following section.

Fig. 1. A

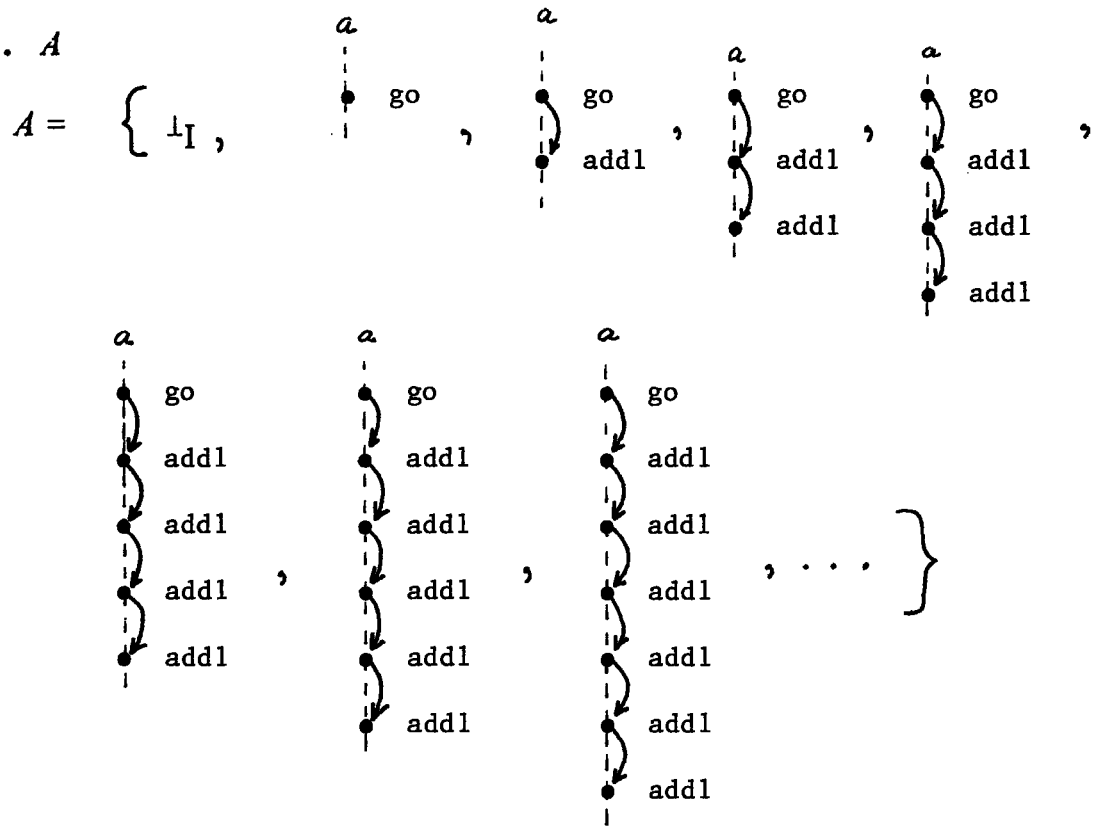
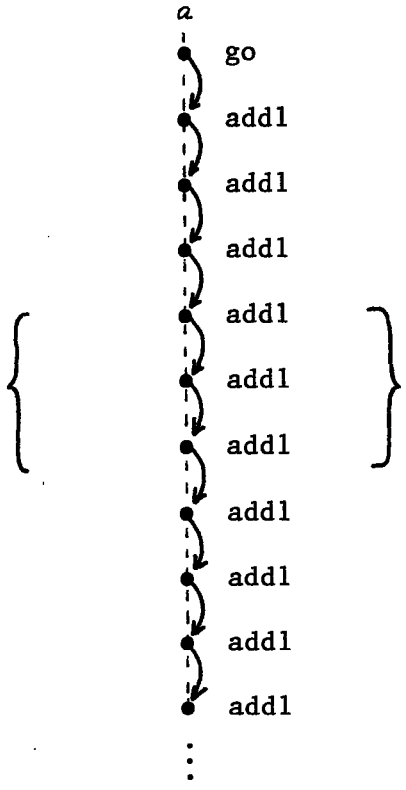


Fig. 2. complete(A)

complete(A) = fg-complete(A)



5.4 EXAMPLE 2: TERMINATING UNBOUNDED CHOICE

The notation used for behaviors and computations was explained in the previous section.

Again the actor system contains only one actor a, but with a slightly different initial behavior b given by

$$b:\ [go] \mapsto (b_0, \{[\![a \leftarrow [add1]]\!], [\![a \leftarrow [halt]]\!]\})$$

$$b_i\ (i \in \omega):\ [add1] \mapsto (b_{i+1}, \{[\![a \leftarrow [add1]]\!]\})$$
$$[halt] \mapsto (\textbf{dead}, \{[\![user \leftarrow [i]]\!]\})$$

That is, upon initialization a sends itself a halt instruction as well as the increment instruction. Since all messages sent eventually arrive at their targets, a will eventually receive this halt instruction and terminate. Unlike example 1, then, there will be no infinite computations in the set of completed computations (except for those which have infinitely many external events).

Again it is a simple matter to compute the associated function τ_*: **P(C)** → **P(C)**. As in the previous example we only care about what happens when the initialization event is the only external event, so we again pretend the least fixed point of τ_* contains only such computations. The least fixed point is B as given below.

The set of completed computations is also given below. This actor system has unbounded nondeterminism, yet it always halts. It is clear that the longer computations are not very likely to happen in any reasonable implementation. Unreasonable implementations which favor the longer computations are still correct, provided they can guarantee fairness and thus termination; they are merely inefficient. It is difficult to see how an implementation could guarantee termination without putting a bound on the nondeterminism, but implementations are not required to preserve all the nondeterminacy present in the semantics.

The set of completed computations is not a finitely generable set. The corresponding finitely generable set **fg-complete**(B) contains a computation in which the halt message never arrives. This shows that approaches which only consider finitely generable sets cannot provide a reasonable semantics for actors, and explains why actor semantics uses **complete** to define the set of completed computations instead of **fg-complete**.

Fig. 3. B

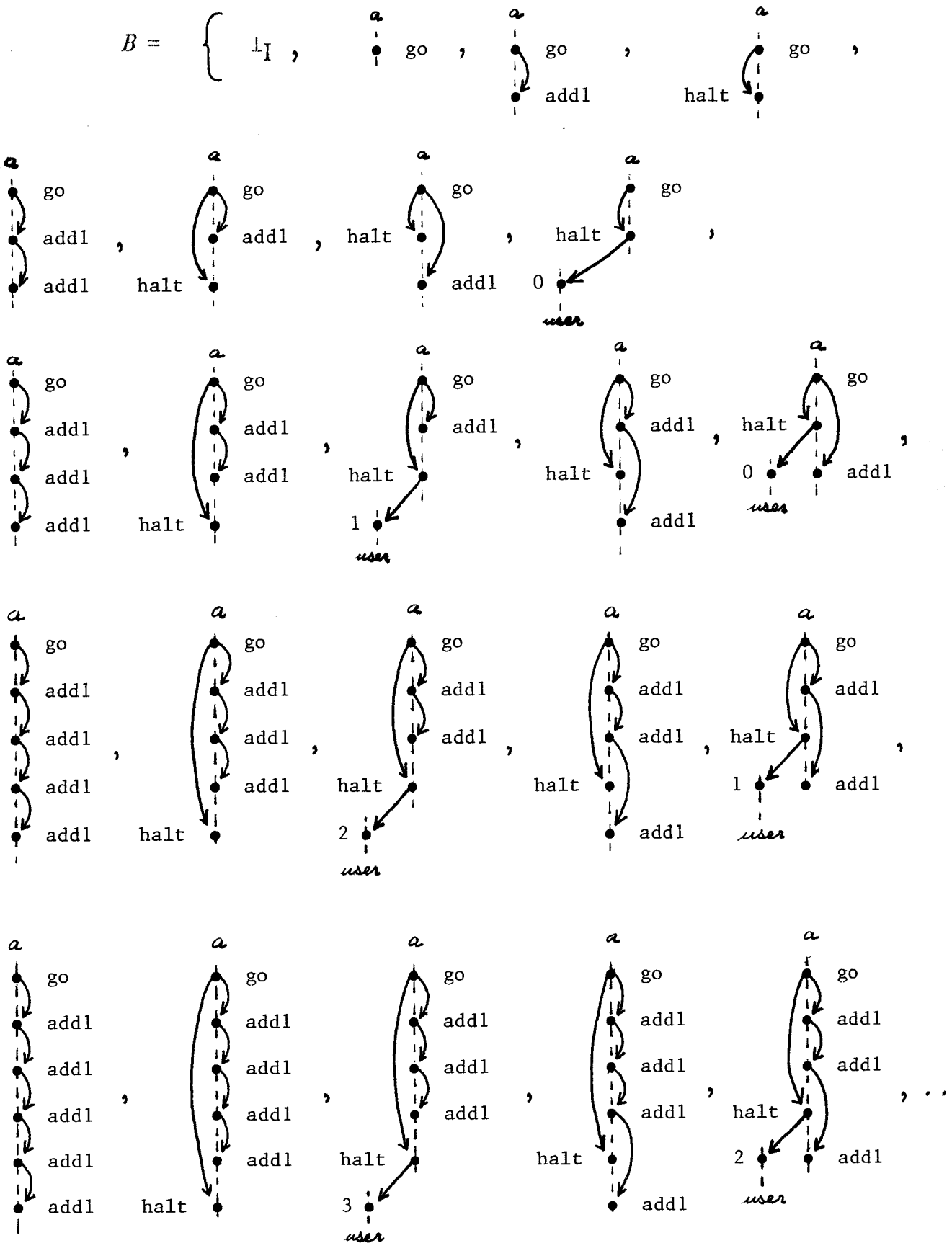


Fig. 4. complete(B)

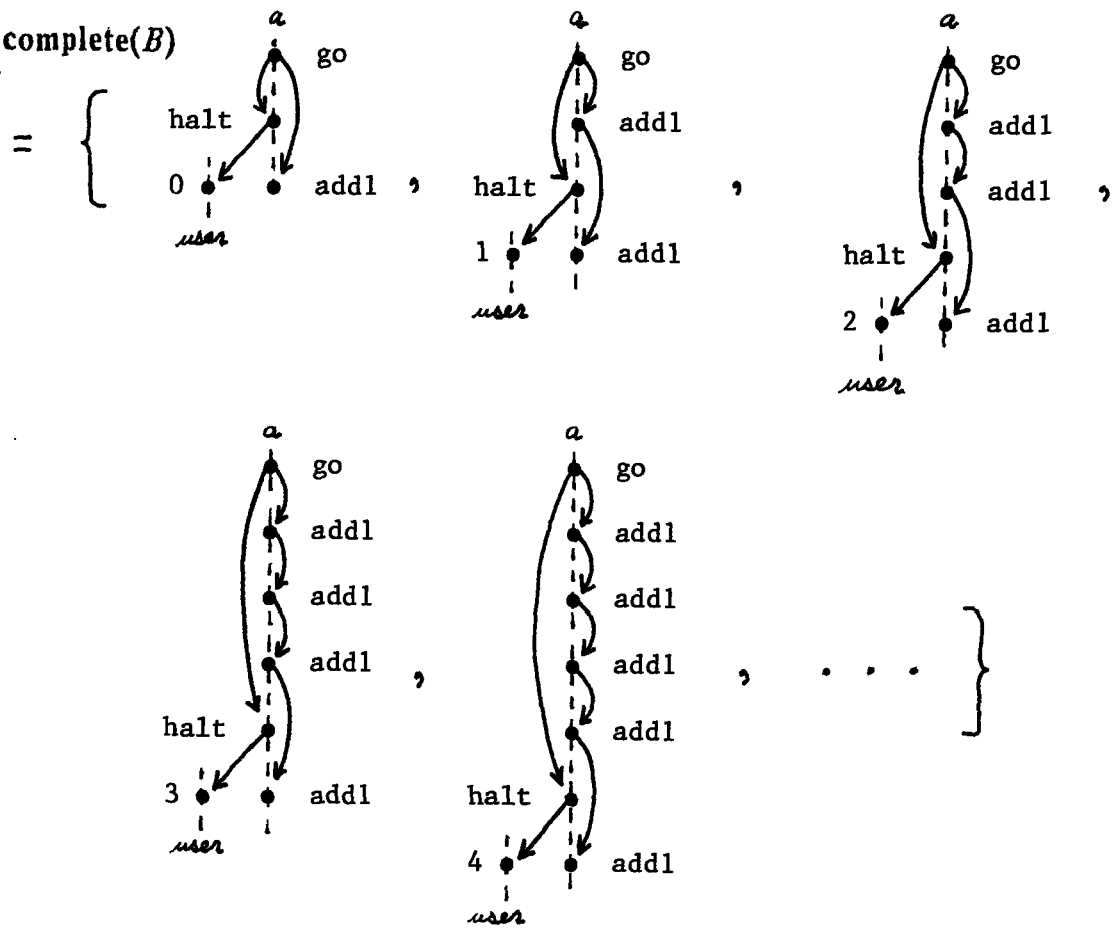
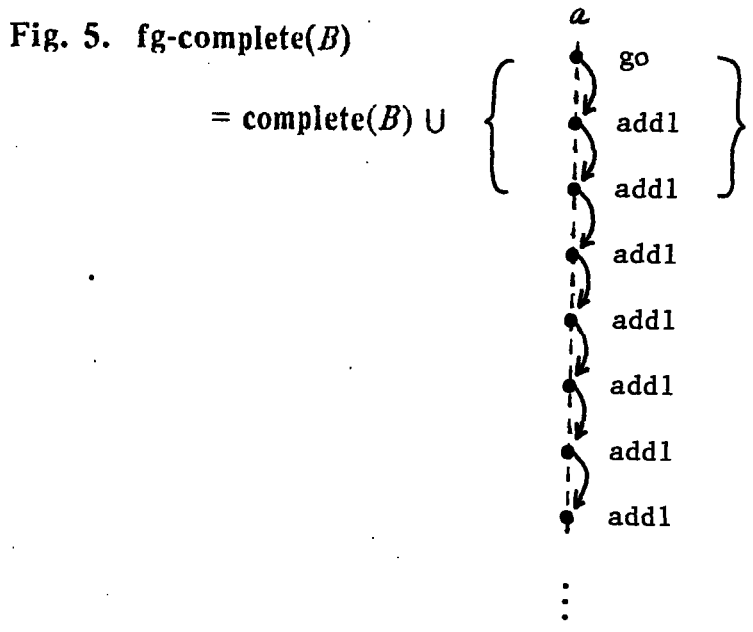


Fig. 5. fg-complete(B)



5.5 EXAMPLE 3: POSSIBLY NONTERMINATING UNBOUNDED CHOICE

Sometimes parallelism is modeled by sequential programs with nondeterministic choice, usually called **or** [deB], [Ke], [Mi], [Pl], [Sm]. Actor computations cannot be modeled in this way. For example, unbounded choice cannot be programmed using **or** without allowing the possibility of nontermination.

This is not to imply that terminating unbounded choice can be implemented in any way, but we distinguish between implementations and programs. If unbounded choice is programmed, each implementation is free to place a bound on the program's choice but that bound is in no way fixed by the program's semantics. In contrast, to actually implement unbounded choice would be to guarantee that infinitely many choices are physically possible.

The example in this section shows how **or** may be modeled in an actor system using an "arrives-first" choice. There is one actor a which initializes itself to 0, and then decides whether to return an answer or to increment according to which of two messages that it sends itself arrives first. More precisely, its initial behavior is given by

b: $[\text{go}] \mapsto (b_0, \{[\![a \leftarrow [\text{add1}]]\!], [\![a \leftarrow [\text{stop}]]\!]\})$

b_i $(i \in \omega)$: $[\text{add1}] \mapsto (\text{wait}_i, \emptyset)$

$[\text{stop}] \mapsto (\textbf{dead}, \{[\![\mathit{user} \leftarrow [i]]\!]\})$

wait_i $(i \in \omega)$: $[\text{stop}] \mapsto (b_{i+1}, \{[\![a \leftarrow [\text{add1}]]\!], [\![a \leftarrow [\text{stop}]]\!]\})$

The notation was explained in section 5.3.

Considering as before only computations which have the single external event $[\![a \leftarrow [\text{go}]]\!]$, and pretending that the least fixed point C of the τ_* associated with this simple actor system contains only such computations, the least fixed point is as given on the next page.

The set of completed computations is infinite, but contains a nonterminating computation. It is a finitely generable set, and for this example the unfair interpretation of the fixed point gives the same set as the fair interpretation adopted by actor semantics. That is, **complete**(C) = **fg-complete**(C). **complete**(C) is given below.

Fig. 6. C

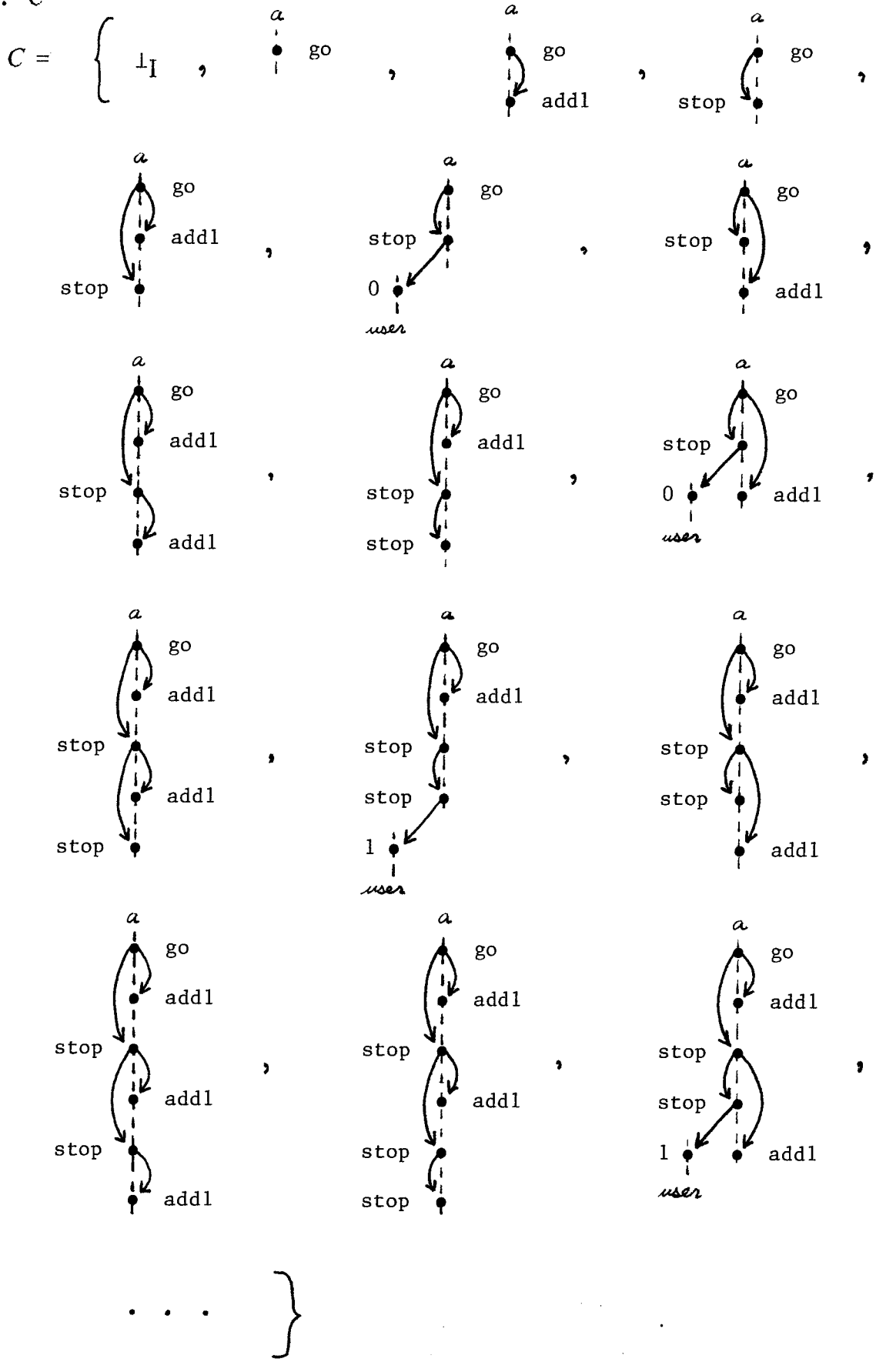
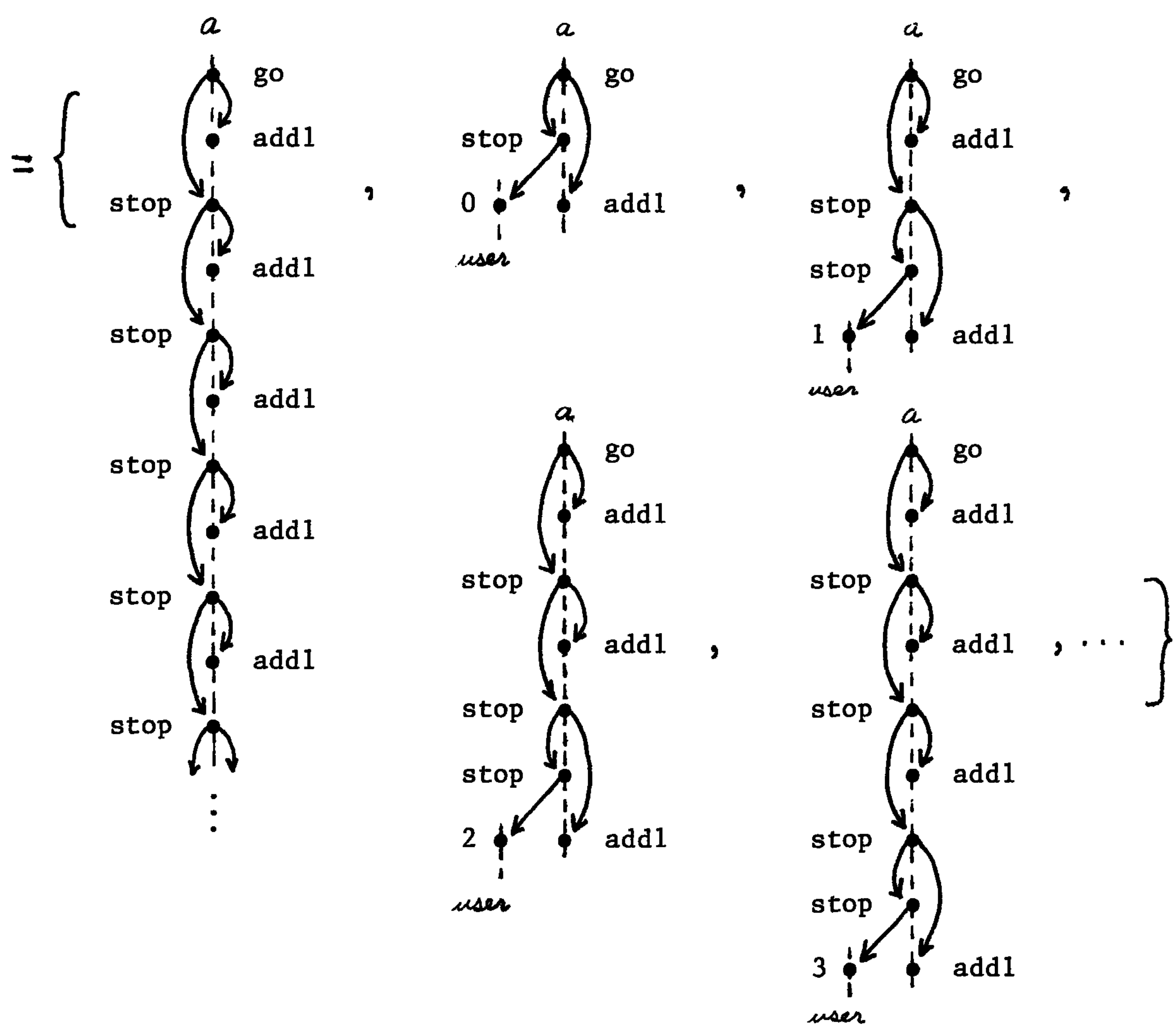


Fig. 7. complete(C)

complete(C) = fg-complete(C)



---

6. Final Remarks

6.1 CONCLUSIONS

The semantics presented in this paper is not yet practical for proving programs correct; it will have to be simplified to eliminate irrelevant operational details. The challenge lies in finding simplifications that do not restrict

generality. The semantics should also be generalized to apply to nondeterministic actors.

To be useful, a general theory such as actors must be able not only to deal with general parallelism but also to recognize and exploit the useful special cases that can be dealt with by simpler methods (such as applicative functions, determinate operators on streams, actor systems with receptionists). Programming languages must encourage these more tractable ways of structuring parallel programs.

In its present state, the semantics is most useful as a precise account of the actor model of parallel processing. It will be used to justify proof rules currently under development.

Its solution to the fairness problem is of theoretical interest. Instead of quantifying over sequences of integers as is done with oracles, completed computations are defined by quantifying over a set of partial computations. This set can be large and complex, but on the other hand it can be simple. Its complexity mirrors the complexity of the computer programs it denotes in an intuitive way, due to the operational definition of meanings as sets of possible initial histories. This kind of semantics naturally supports proofs by symbolic evaluation [BaSi], [HeAtt], [Yo].

The exact relation between this semantics and the more conventional power domain semantics remains to be investigated.

6.2 ACKNOWLEDGEMENTS

Carl Hewitt gave me much friendly encouragement, helful criticism and suggestions, and his provocative faith in the actor model. Giuseppe Attardi pointed out and explained to me the problem of fairness in actor semantics, sharing his ideas and listening to mine. Irene Greif encouraged me to write it all down, and suggested directions for further research.

REFERENCES

[Ba] Henry Baker. Actor systems for real-time computation. MIT PhD Thesis, MIT LCS Technical Report 197, March 1978.

[BaSi] Gerald Barber and Maria Simi. A concurrent program: its symbolic evaluation and the property of unbounded nondeterminism. Forthcoming.

[BrAc] Dean Brock and William B Ackerman. An anomaly in the specification of nondeterministic systems. MIT Computation Structures Group Note 33-1, January 1978.

[CaLe] J M Cadiou and J J Levy. Mechanizable proofs about parallel processes. Proc 14th Annual Symposium on Switching and Automata Theory, October 1973, page 34.

[Cl] Will Clinger. Global time in actor computations. Working paper, June 1979.

[deB] J W de Bakker. Recursive programs as predicate transformers. IFIP Working Conference on the Formal Description of Programming Concepts, St Andrews, New Brunswick, Canada, August 1977, pages 7.1-7.15.

[Di] Edsger Dijkstra. A Discipline of Programming. Prentice Hall, 1976.

[Fr] Nissim Francez, C A R Hoare, Daniel Lehmann, Willem de Roever. Semantics of nondeterminism, concurrency and communication. Draft, August 1978.

[Gr1] Irene Greif. Semantics of communicating parallel processes. MIT PhD Thesis, MIT Project MAC Technical Report 154, September 1975.

[Gr2] Irene Greif. On proofs of programs for synchronization. Proc 3rd International Colloquium on Automata, Languages, and Programming. Edinburgh University Press, 1976, pages 494-507.

[Ha] Robert Halstead. Multiprocessor implementations of message-passing systems. MIT SM thesis, MIT LCS Technical Report 198, February 1978.

[HeSt] Horst Herrlich and George Strecker. Category Theory. Allyn and Bacon, 1974.

[He1] Carl Hewitt. Viewing control structure as patterns of passing messages. Artificial Intelligence 8, 1977, pages 323-363. Also in Winston and Brown [ed], Artificial Intelligence: an MIT Perspective, MIT Press, 1979.

[He2] Carl Hewitt. Concurrent systems need both sequences and serializers. Working paper, February 1979.

[He3] Carl Hewitt. Preliminary design of the APIARY for VLSI support of knowledge-based systems. Working paper, April 1979.

[HeAtk] Carl Hewitt and Russell Atkinson. Synchronization in actor systems. Record of 1977 Conference on Principles of Programming Languages, Los Angeles CA, January 1977, pages 267-280.

[HeAtt] Carl Hewitt and Giuseppe Attardi. Proving properties of concurrent programs expressed as behavioral specifications. Working paper, June 1978.

[HeBa1] Carl Hewitt and Henry Baker. Laws for communicating parallel processes, IFIP-77, Toronto, August 1977, pages 987-992.

[HeBa2] Carl Hewitt and Henry Baker. Actors and continuous functionals. IFIP Working Conference on Formal Description of Programming Concepts, St Andrews, New Brunswick, Canada, August 1977. Also available as MIT AI Memo 436A.

[Ho] C A R Hoare. Communicating sequential processes. CACM **21**, 8, August 1978, pages 666-677.

[Ka] Gilles Kahn. The semantics of a simple language for parallel programming. IFIP-74, Stockholm, 1974.

[KaMcQ] Gilles Kahn and David McQueen. Coroutines and networks of parallel processes. IFIP-77, Montreal, August 1977, pages 993-998.

[Ke] Robert Keller. Denotational models for parallel programs with indeterminate operators. IFIP Working Conference on the Formal Description of Programming Concepts, St Andrews, New Brunswick, Canada, August 1977.

[Ko] Bill Kornfield. ETHER -- a parallel problem solving system. To appear in IJCAI-79.

[Le] Daniel Lehman. Categories for fixpoint semantics. Seventeenth Annual Symposium on Foundations of Computer Science, October 1970, pages 122-126.

[MiMi] George Milne and Robin Milner. Concurrent processes and their syntax. JACM **26**, 2, April 1979, pages 302-321.

[MiSt] Robert Milne and Christopher Strachey. A theory of programming language semantics. Chapman and Hall, London, 1976.

[Mi] Robin Milner. An approach to the semantics of parallel programs. Proceedings Convegno di Informatica Teorica, Pisa, 1973.

[Pl] G D Plotkin. A powerdomain construction. SIAM J Computing **5**, 3, September 1976, pages 452-487.

[Sc1] Dana Scott. Outline of a mathematical theory of computation. Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems, 1970, pages 169-176.

[Sc2] Dana Scott. Data types as lattices. SIAM J Computing **5**, 3, September 1976, pages 522-587.

[Sm] Michael Smyth. Power domains. Journal of Computer and System Sciences **16**, 1978, pages 23-36.

[SmPl] M B Smyth and G D Plotkin. The category-theoretic solution of recursive domain equation. Proceedings 18th Annual IEEE Symposium on Foundations of Computer Science, pages 13-17.

[Yo] Akinori Yonezawa. Specifications and verification techniques for parallel programs based on message passing semantics. MIT PhD thesis, MIT LCS Technical Report 191, December 1977.