



Actor Semantics of PLANNER-73

Irene Greif and Carl Hewitt

Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract

Work on PLANNER-73 and actors has led to the development of a basis for semantics of programming languages. Its value in describing programs with side-effects, parallelism, and synchronization is discussed. Formal definitions are written and explained for sequences, cells, and a simple synchronization primitive. In addition there is discussion of the implications of actor semantics for the controversy over elimination of side-effects.

Introduction

A formalism for semantics is usually developed and presented in some context which provides motivation. It might be designed for the purpose of being the formal definition of a specific language as a guide to programmers and implementors. The formalism might be used as the means for describing properties of a particular program, or as a basis for a set of proof rules for verifying properties of programs. The motivation for this paper is development of a semantic base not for some particular programming language, but rather for a kind of computation.* The systems of interest are those which involve parallelism, side-effects, and synchronization. The concepts which must be available to express properties of these systems will be presented and examples given. However, no particular programming language will be used.

Our approach to the semantics will be centered on the concept of an actor. An actor is simply an object which can send or receive messages. All elements of a system can be viewed as actors. The only activity possible is transmission of one actor to another actor. The energy for computation comes from an activator which can follow the script of an actor. All programs and systems will be referred to as actors. The existence of more than one activator indicates parallelism.

Continuations

Once an actor, *m*, is transmitted to another actor, *t*, the activator proceeds by following the script of *t* using information from *m*. For this to be of any use as a model of computation, it must be possible for *m* to have some structure. In many cases *m* will simply contain a message. It will often also contain a continuation part. Then *m* may

* Actor semantics was not developed to explain or define PLANNER-73. PLANNER-73 is a language dedicated to the direct realization of this semantic base.

be represented as

(apply: message (then-to: continuation)).

If a message is sent without a continuation then all further instructions for computation must be determined by the target. Sending a continuation makes it possible to indicate another actor to which control can eventually be passed (i.e., an actor whose script the activator may follow).

The reason that it is useful to be able to talk about continuations in a semantic formalism is that otherwise much of the control structure has to remain implicit in something like an evaluator or a compiler. Control structure is a pattern of passing messages in a computation. Ideally it should be possible to distinguish between, say, recursive and iterative computations as being two different patterns of message passing behavior which can be precisely characterized.

An ordinary functional call to a function *f* can be implemented by passing a transmission with an explicit continuation to *f*. The continuation represents the actor to which the value of *f* should be sent. In addition, other kinds of control structures can be expressed, such as jumps (in which no continuation is present) and co-routines [see Appendix].

Distribution of Information

Information can be represented in various ways in an actor system. Often it is desirable to represent knowledge redundantly with different uses of the same knowledge appearing in different guises in several places. How one might choose to distribute it depends on one's purposes and the various uses to which the knowledge can be put. Concentrating on use as the key to representation of data structures contrasts with the approach of concentrating on the notion of type [van Wijngaarden, 1969].

For example, in defining lists and the usual operations on lists, there is flexibility in how the information about the structure of lists may be distributed among these operations and the data objects themselves. One possibility is that the function call denoted by (length *L*) can directly produce the length of the list, *L*. That is, the operation, length, could be defined in such a way that it knows enough about how lists are constructed (and about how they can be decomposed) to be able to operate on the list directly and to compute its length. For example, the operator length could repeatedly ask the list *L* for its next element and keep a count of the number of times it succeeds.

Alternatively, the operator `length` might know nothing about lists and when sent one would have to depend on the list itself to understand its own structure (and to understand it in terms of the concept "length" among others). What happens in this case is that (`length L`) turns around and asks `L` for its "length." This makes sense for fixed length lists since for such lists the information never changes and need only be computed once. This message-passing style of definition is particularly useful for "polymorphic" operators in an *interactive* system where new kinds of objects which share some of the properties of lists can be defined at any time. It is desirable for (`length L`) to continue working and to extend correctly to new data types that are continually being defined. When information can be put wherever it seems natural, polymorphic operators can be defined without details of type checking. Instead the operator can be applied to any type of data and it is up to the data to determine whether it is a reasonable operator and if so how to perform its operation. Depending on circumstances, either style can lead to a more efficient implementation, but both should be representable.

Behavior of an Actor

Actors are defined by their behaviors. A behavior is a partially ordered set of events which represent transmissions. The behavior involving only a single activator (process)* is a totally ordered sequence of events. A description of the behavior of an actor has at least two uses. First, it defines the actor. Secondly, it specifies the properties of the actor which can be relied on in a proof about properties of another system in which it is contained.

The event oriented approach to semantics is well suited to the expression of meanings of programs with side-effects and parallelism. Rather than attempting to record the current state of the world in something like a state vector [McCarthy, 1964], changes in state are reflected in changes over time in the behaviors of the actor cells. The event oriented approach is particularly useful when there is parallelism as well as change in state. If one tries to record all possible global states (including control information) for all the parallel processes one can become involved in a combinatorial explosion of states. In order to describe the effect of a change in a single cell using global states, one would have to make the corresponding changes in the state information for each of the processes which could be active.

Level of Detail in Describing Computations

Another degree of flexibility available in actor semantics involves the amount of detail to be included in specifications. That is, an actor can be specified (or its set of possible behaviors can be specified) in as much or as little detail as is desired. In describing an existing actor system this level of detail is fixed by the choice of a set of distinguished actors contained in the actor system. Only events involving these distinguished actors will be recorded as behavior of the system. To specify desired behavior of a system one writes axioms which any behavior of the system must satisfy. At a very high level of detail these axioms will essentially be input-output constraints, i.e. *what* is to be done, not *how*. If more detailed specifications are made, the possible behaviors which will realize those input-output constraints become more restricted. Generally, in

* Note that while the activator as a source of energy for computation may be analogous to some notion of process, the activator is definitely not meant to connote any of the following concepts: program counters, pushdown stacks, or address spaces.

order to prove that a set of specifications is satisfied by a particular system, one examines the behaviors of the system at a level of greater detail than that of the specifications and proves that these behaviors realize the behavior that is required.

These points should give some indication of how the simple actor concept can be used as a building block to define the higher level concepts which are useful descriptive tools. After the formal definitions of behaviors of actors in the following section, we will give semantic definitions of several familiar systems, illustrating the application of these concepts.

Behaviors

Since actors are defined by their behaviors, the concept of behavior will have to be defined. Behaviors consist of events.

Definition: An event is a four-tuple, $\langle t \ m \ \alpha \ ec \rangle$ where t is the target, m is the transmission, α is the activator, and ec is the event count of the activator α .

As stated earlier, an event is a description of the transmission of m to t by α . Since the events of a single activator are totally ordered in time, an event count is also necessary to completely characterize a particular transmission. This means that if identical transmissions occur involving some particular t , m , and α then the two can in fact be distinguished by the proper time for α at which they occurred.

The event count ec' denotes the time immediately after ec . Therefore $\langle t \ m \ \alpha \ ec \rangle$ immediately precedes $\langle t \ m \ \alpha \ ec' \rangle$ and we can write $\langle t \ m \ \alpha \ ec \rangle \rightarrow \langle t \ m \ \alpha \ ec' \rangle$. For events with different activators, $\langle t_1 \ m_1 \ \alpha_1 \ ec_1 \rangle \rightarrow \langle t_2 \ m_2 \ \alpha_2 \ ec_2 \rangle$ if and only if at least one of t_2 , m_2 , α_2 is created in the event $\langle t_1 \ m_1 \ \alpha_1 \ ec_1 \rangle$. (Some actors have the ability to create new actors -- `cons-cell` creates an actor with the behavior of a cell, `cons-list` creates a list, etc.) The transitive closure of the relation \rightarrow is referred to as causally precedes i.e. for events E_1 and E_2 , $E_1 \rightarrow E_2$ is read as "E₁ causally precedes E₂." This is the ordering which can reasonably be considered to be a property of the actor system in question. The relationship between causally precedes and the actual time at which events occur is that if $E_1 \rightarrow E_2$, then E_1 will necessarily occur before E_2 in any computation. Other orderings may be imposed by axioms for primitive actors. (For example a synchronization primitive introduces constraints in the causally precedes relation.)

Definition: A history is a set of events partially ordered by the transitive closure of the relation \rightarrow .

An external environment for an actor system is a set of actors constituting the "outside" of the system. This includes an assignment of an initial set of activators and initial events for each activator which will determine the "beginning of history" for this system and environment. These initial events will always be about transmissions which cross the boundary and enter the actor system, i.e. events which have a target inside the system and a message outside the system. Histories are written for systems in environments.

The behavior of an arbitrary system of actors can be extracted from the history by choosing only those events which involve certain distinguished actors. Note: we will consider any new actor created by an event involving an actor in the system Σ to be in Σ also.

Definition: For any history of the actor system Σ the behavior of system Σ with respect to the set of distinguished actors $\Delta \subset \Sigma$ is the subset of the events of the history containing events $\langle t \ m \ \alpha \ ec \rangle$ where either

1. t is in Σ and m is not.
2. m is in Σ and t is not.
3. Either t or m is in Δ .

Thus an event will be in the behavior if it describes either a transmission across the boundary of Σ (i.e. with target inside and transmission outside or vice versa) or a transmission with a distinguished actor as either target or message. This is how the level of detail of a behavior is fixed by the choice of Δ . Note that if the set Δ is empty then the minimally detailed behavior results, i.e. the behavior of Σ as seen from the outside.

Two actor systems are equivalent with respect to some set of distinguished actors if and only if they have identical behaviors with respect to that same set of actors. The choice of distinguished actors can make a difference in equivalence of actor systems. For example, two programs for factorial can be seen to be equivalent at the level of input-output transformation (i.e. with no actors distinguished) even if a larger set of distinguished actors would reveal one to be a "counting-up" factorial and the other to be "counting-down."

If multiplication, represented by $*$, is considered to be a distinguished actor then in any behavior of one factorial there might be a series of events $\langle * [1 \ 1] \ \alpha \ ? \rangle \rightarrow \langle * [2 \ 1] \ \alpha \ ? \rangle \rightarrow \langle * [3 \ 2] \ \alpha \ ? \rangle$ and so on, while in the other there might be a series $\langle * [n \ 1] \ \alpha \ ? \rangle \rightarrow \langle * [n-1 \ n] \ \alpha \ ? \rangle \rightarrow \langle * [n-2 \ n*n-1] \ \alpha \ ? \rangle$.

Behavioral Specifications of Systems

Since behaviors of most actors are infinite objects it will not often be possible to write out completely the set of behaviors of any given (or desired) actor. And in fact, many behaviors of a system will contain incidental information about events in the environment. Instead, characteristics common to all behaviors of any actor system containing the actor in question can be specified in the form of axioms about events and the causally precedes relation. These are not Hoare [1969] style axioms about transformations of predicates over the occurrence of an operation or of a line of code, but rather are about the kinds of events that can possibly occur in a computation and how they must be ordered.

When an actor is being specified as a primitive, that is, when it is not defined in terms of other actors, the axioms describe conditions which must be satisfied by any history. Any actor can be described from the outside but if an actor is to be implemented in terms of other actors it should be understood that between any two events there may be an arbitrary number of other events if we examine a history (or even a more detailed behavior). However, all events related by causally precedes will remain so related. When we do not wish to imply that E_2 immediately follows E_3 we will write, "?" in the event count. In general "?" can be used in any place in an event to indicate that we do not care about what actor or activator appears there.

Recall that if two events E_1 and E_2 are known to be in a behavior, we can write $E_1 \rightarrow E_2$ to indicate that E_1 causally precedes E_2 . When E_1 and E_2 have not previously been mentioned, a statement of the form $E_1 \rightarrow E_2$ should be read as "whenever E_1 is in the history (behavior) then so is E_2 and E_1 causally precedes E_2 ," or

alternatively as "if E_1 is in the history (behavior) it will cause E_2 ." This is a way in which we guarantee response in finite time from an actor.

We will continue to use the labelling convention on the transmissions actors. The transmissions are of the following general form:

(apply: message (then-to: continuation) (else-to: complaint-dept))

We will often omit the else-to part when it is not of interest.

Sequences

The following axioms about the behaviors of sequences summarize some of the familiar properties of sequences while illustrating the use of axioms to constrain the kinds of behavior an actor can have. Note that since sequences are indeed functional objects they could have been described in more conventional mathematical notation [Scott and Strachey, 1971].

An actor which is a sequence of n elements has the property (among others) that it can be asked for its i th element. If a sequence is represented by $\langle x_1 \dots x_n \rangle$ this property can be specified by the following axiom:

$\langle x_1 \dots x_n \rangle$ (apply: [i] (then-to: c) (else-to: cd)) $\alpha \ ec \rangle$
 $\rightarrow \langle c \text{ (apply: } x_i \ \alpha \ ec \rangle \quad \text{for } 1 \leq i \leq n$

This means that at the level of detail at which sequences are "black boxes" the event in which the sequence $\langle x_1 \dots x_n \rangle$ is sent the message [i] and the continuation, c, always causes as the next event the transmission of the i th element of the sequence to that continuation. This transmission had three components: a message, a continuation, and a complaint department. This last actor is the one to which 'not-applicable' messages are sent if the target actor does not like its message. For instance, for $i > n$, it is the case that the sequence satisfies

$\langle x_1 \dots x_n \rangle$ (apply: [i] (then-to: c) (else-to: cd)) $\alpha \ ec \rangle$
 $\rightarrow \langle cd \text{ (apply: 'not-applicable') } \alpha \ ec \rangle$.

When using axioms as specifications of all possible behaviors one assumes that for any messages not explicitly dealt with the actor being described causes 'not-applicable' to be sent to the complaint department. Thus at this stage in the definition it can be assumed that sending any transmission (apply: x (then-to: c) (else-to: cd)) to a sequence where x is not a sequence containing an integer, causes 'not-applicable' to be sent to cd.

Cells

Cells do not fit as neatly into a functional notation. A cell is created with some initial contents. It is possible to ask for the contents of a cell. Update messages to that cell can change its contents. Thus it is possible that several events in a single behavior can have identical targets (namely the cell) and identical transmissions and yet cause different events to occur. This occurs when the events have transmissions

(apply: 'contents (then-to: continuation))

but the contents have been changed between the two events. In each case the continuation is sent different actors.

This means that cells are side-effect primitives. Loosely speaking, an actor system has undergone a side-effect if its behavior has changed over time. In terms of functions, a side-effect has occurred when the same procedure p has been called on two different occasions with the same argument and returned different values. This can be generalized to behaviors by looking at future events rather than just at values returned.

Cells correspond to side-effect primitives in various languages. Therefore, it is likely that most readers have relied on them in programming. We will now try to formulate exactly which properties we have been relying on. First would be that creation of a cell (or allocation of space for a cell) with its initial value should be an operation guaranteed to take only finite amount of time. This is also a desirable property of updates and contents queries. (Anyone who has convinced himself that a given program which contains assignment statements always terminates has relied on these properties).

Axiom 0:

Cons-cell creates cells.

$\langle \text{cons-cell (apply: [x] (then-to: c_1) (else-to: cd_1)) } \alpha \text{ ec}_1 \rangle$
 $\rightarrow \langle c_1 \text{ (apply: cell}^* \text{)} \alpha \text{ ec}_1 \rangle$

where cell is a newly created actor.

Axiom 1:

Cells can respond to contents queries and updates.

$\langle \text{cell (apply: 'contents (then-to: continuation) (then-to: cd)) } \alpha \text{ ec} \rangle$
 $\rightarrow \langle \text{continuation ? } \alpha \text{ ec} \rangle$
 $\langle \text{cell (apply: [x] (then-to: continuation) (else-to: cd)) } \alpha \text{ ec} \rangle$
 $\rightarrow \langle \text{continuation (apply: cell) } \alpha \text{ ec} \rangle$

[Notice that a cell returns itself, not its value after update]

Newly created actors are marked in events by an asterisk[*]. It is often useful to be able to tell explicitly when an actor, a , appears for the first time in event E since for any other event E_1 in which a occurs it must be the case that $E \rightarrow E_1$.*

The next two properties that we rely on are related. Initially we expect contents queries to find the original contents in the cell. This should be true until an update transmission is sent. Similarly, once an update transmission is sent, say $[x]$, we would expect to find y in the cell until it is updated. Thus two more axioms are required, one about retrieving the initial contents and the other about retrieving the last contents stored.

Axiom 2 says that if cell first appears in event E_1 and if no update occurs between E_1 and contents query E_2 , then E_3 will be an event which sends the initial contents to the continuation.

* This has implications for the relation "knows about" which can be formalized and which has implications for control structure recognition and implementation details. [See Appendix]

Axiom 2

Initial Contents -- A cell created with x as its contents will respond to the message 'contents with x if no new value has been explicitly placed there yet.

Suppose that $E_0 \rightarrow E_1$ is in the behavior where

E_0 is $\langle \text{cons-cell (apply: [x] (then-to: c) (else-to: cd)) } \alpha_1 \text{ ec}_1 \rangle$
 E_1 is $\langle c \text{ (apply: cell}^* \text{)} \alpha_1 \text{ ec}_1 \rangle$

If there are events E_2, E_3 where

$E_1 \rightarrow E_2 \rightarrow E_3$
 E_2 is $\langle \text{cell (apply: 'contents (then-to: c_1) (else-to: cd_1)) } \alpha \text{ ?} \rangle$
 E_3 is $\langle c_1 \text{ (apply: y) } \alpha \text{ ?} \rangle$

and $\neg \exists E (E_1 \rightarrow E \rightarrow E_2)$

where E is of the form

$\langle \text{cell (apply: [x] (then-to: ?) (else-to: ?)) } \alpha \text{ ?} \rangle$

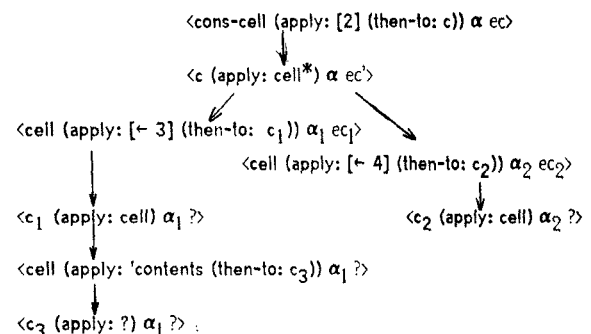
then $y = x$.

Axiom 3 makes a similar statement about the last contents stored by an update, rather than about the initial contents. The only significant change from the statement of Axiom 2 is that E_0 is:

$\langle \text{cell (apply: [x] (then-to: c) (else-to: cd)) } \alpha \text{ ec}_1 \rangle$.

Unfortunately these axioms are reasonable only in the absence of parallelism since we are ignoring the fact that \rightarrow only refers to time constraints imposed by the program. In programs without parallelism, it is the case that time constraints imposed by the program form a total order on events making these axioms sufficient. (Also, due to the predictable nature of the cell in the absence of parallelism its use does not really increase power [See Appendix].) But in general there may also be update events occurring independently of this ordering.

In a behavior such as:



The message place marked by the question mark can represent either the message 3 or 4. Which actor it will be is not a property of the system. Axiom 3 would force it to be 3.

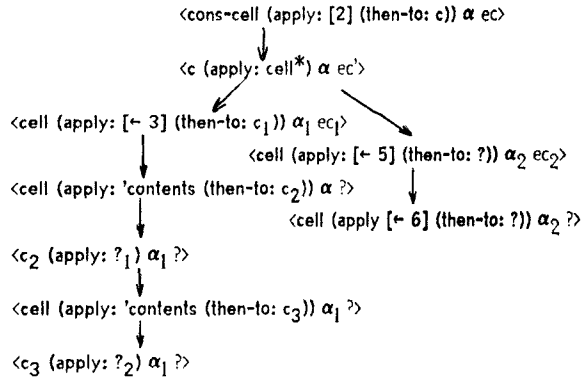
The obvious patch is to weaken the axiom from saying exactly what has to be returned to the continuation, to just restricting the set of actors it could be. This requires adjusting the last line of axiom 2 to say that y is in the set

$$\{ z \mid z = x \vee \exists E_4 \text{ such that } (E_4 \text{ is } \langle \text{cell (apply: [x] (then-to: c) (else-to: cd)) } \alpha \text{ ?} \rangle \wedge \neg (E_4 \rightarrow E_2 \text{ or } E_2 \rightarrow E_4)) \}$$

That is, y can be the initial contents or any value stored in parallel with this activator.

Axiom 3 should be adjusted similarly, such that the contents could be either the contents stored in the last update event to causally precede the contents query, or in any update which is parallel to the activator.

Now observe the following behavior:



In fact, our intuition about the contents of cells is that it is even more restricted than stated, because if ?₁ has value 5, then ?₂ can no longer have any value in {3, 5, 6}, but rather can only have values in {5, 6}. There are many examples in which we can make deductions that severely restrict the possible contents of a cell. These are actually deductions about constraints in addition to those imposed by → as to the order in time in which events could have occurred.

A first reaction to these examples might be that although the current definition jars with intuition, it may be reasonable to accept bizarre behavior once non-determinism is a possibility. Perhaps we simply should not rely on any meaningful contents in this situation. However, there are times when we do rely on the behavior of cells even in the presence of parallelism. For instance, several solutions to the mutual exclusion of critical sections which do not make use of semaphores [Dijkstra, 1965; Knuth, 1966], implicitly make use of the cell's retaining its last contents stored even in the presence of parallelism.

We found that the fact from which all our deductions about reasonable behaviors were made was that the contents retrieved had to be consistent with some possible total ordering of all references to that cell. In that ordering each time the contents is checked it must agree with the last contents stored. Thus cells allow one to make deductions about the actual order in which events occur even though the events may not be necessarily ordered by →. This is the property of cells which makes it possible to use busy waiting to synchronize independent processes.

Axiom 4:

In any behavior containing an actor, cell, created by cons-cell, it is possible to define a total ordering, ⇒, on all events which have cell as target. This ordering has the properties that if E₁ and E₂ have cell as target then

1. E → E₂ implies E₁ ⇒ E₂
2. If E₁ → E

where E₁ = <cell (apply: 'contents (then-to: c)) α ec>

and E = <c (apply: x) α ec'>

then x is the last contents stored with respect to ⇒.

The last contents stored with respect to an ordering can be defined for an arbitrary total ordering, →, and used in axiom 4 with respect to ⇒.

Definition: For the event E, the last contents stored in cell with respect to the ordering → is defined to be either

the initial contents, if cell is created in event E₀ and there is no update event E₃ such that E₀ → E₃ → E

or

the actor stored in update E₀, if E₀ → E and there is no other update E₃ such that E₀ → E₃ → E.

There can still be non-determinism since although in any computation the events referring to a particular cell can be totally ordered, in fact, only parts of that ordering will be forced by the actor system.

Synchronization

Synchronization fills in the gaps in the total order. It imposes total ordering on all events involving it. This means that if some synchronization actor always encased a cell it would never be necessary to worry about unordered updates and contents queries. The synchronization would impose an ordering and the cell would always contain the last (in the causally precedes relation) contents stored.

This is part of a general form for specifying behaviors of any system using synchronization. There must be an axiom to the effect that an ordering is imposed, and the axioms about other properties of the system can be dependent on this ordering.

One way to protect a cell would be to define a primitive cons-protected-cell which can create cells and protect them. Externally cons-protected-cell should have behavior satisfying:

```

<cons-protected-cell (apply: x (then-to: c)) α ec>
  --> <c (apply: protected-cell*) α ec'>.
  
```

In order to describe the behavior of protected-cell, it is necessary to describe cons-protected-cell in some more detail. Although its means of synchronization can be left unspecified, the fact that it contains a locally known cell and that this cell is encased in a protective actor which is known to the world should be axiomatized.

The behaviors of cons-protected-cell also satisfy:

```

<cons-protected-cell (apply: x (then-to: c)) α ec>
  --> <cons-cell (apply: x (then-to: c1)) α ?>
  where c1 has behaviors which satisfy:
  <c1 (apply: cell*) α ?>
  --> <c protected-cell* α ?>.
  
```

Now we can describe the behavior of protected-cell.

Axiom 1:

From the outside a protected cell should look like a cell.

$\langle \text{protected-cell (apply: 'contents (then-to: c)) } \alpha \text{ ec} \rangle$
 $\rightarrow \langle \text{c (apply: ? (then-to: c)) } \alpha \text{ ec} \rangle$
 and
 $\langle \text{protected-cell (apply: [← ?] (then-to: c)) } \alpha \text{ ec} \rangle$
 $\rightarrow \langle \text{c (apply: protected-cell) } \alpha \text{ ec} \rangle$.

The rest of the properties of `protected-cell` can be inferred from further details about what it does with its messages:

Axiom 2:

`Protected-cell` sends messages to `cell`.

$\langle \text{protected-cell (apply: x (then-to: c)) } \alpha \text{ ?} \rangle$
 $\rightarrow \langle \text{cell (apply: x (then-to: c}_x \text{)) } \alpha \text{ ?} \rangle$
 where
 $\langle \text{c}_x \text{ t } \alpha \text{ ec} \rangle$
 $\rightarrow \langle \text{c t } \alpha \text{ ec} \rangle$.

Thus `protected-cell` has side-effects because its behavior reflects the behavior of `cell`.

Axiom 3:

`Protected-cell` adds ordering to a behavior.

If $E_1 \rightarrow E_2$ and $E_3 \rightarrow E_4$ where

$E_1 = \langle \text{protected-cell (apply: x (then-to: c}_1 \text{)) } \alpha \text{ ?} \rangle$
 $E_2 = \langle \text{cell (apply: x (then-to: c}_{1x} \text{)) } \alpha \text{ ?} \rangle$
 $E_3 = \langle \text{protected-cell (apply: x}_2 \text{ (then-to: c}_2 \text{)) } \alpha \text{ ?} \rangle$
 $E_4 = \langle \text{cell (apply: x}_2 \text{ (then-to: c}_{2x} \text{)) } \alpha \text{ ?} \rangle$

then either $E_2 \rightarrow E_4$

or $E_4 \rightarrow E_2$.

This says that somewhere in the interface between `protected-cell` and `cell` there is locking and unlocking going on which causes all events with `cell` as target to be ordered by causally precedes. The exact mechanism cannot be seen at this level of detail but it exists.

Axiom 4:

Given E_1, E_2, E_3 , and E_4 as above and $E_2 \rightarrow E_4$, then in fact

$E_2 \rightarrow \langle \text{c}_{1x} \text{ (apply: ?) } \alpha \text{ ?} \rangle \rightarrow E_4$.

That is, the event in which a response is made from the cell triggers `protected-cell` and allows another transmission to the cell. Thus c_{1x} must know how to free the cell. Again, since the synchronization mechanism is primitive we cannot see the chain of events *causing* the unlocking. It is simply a property of this mechanism that it causes the protected cell to let someone else get to the cell.

Now, from the point of view of the cell, all events are totally ordered by causally precedes. If cells were always used inside protectors only the simpler single activator axioms for cells would be interesting. The more complex global ordering constraints can be deduced from a combined understanding of synchronization which imposes ordering and of cells which remember their contents.

Conclusions

We can express characteristics of a side-effect primitive in the presence of parallelism. Side-effects are described by delineation of possible changes in behavior rather than in a state vector. Synchronization can be used to impose additional ordering constraints.

Further research is in progress on applying this semantic formalism to structured synchronization actors, similar to monitors [Hoare, 1974]. Once a sufficiently expressive set of descriptions is developed we will implement some systems, such as an airline reservation system or a solution to the readers-writers problem, and prove "correctness" relative to these descriptions.

The formalism can also be used in comparative studies of the behaviors characteristic of a variety of control structures. An example of this kind of work accompanies a result on elimination of cells presented in the Appendix.

Appendix

Concerning Side-Effects

There is some controversy at present over the desirability and necessity of use of side-effect primitives. A result to the effect that cells (or other side-effect primitives) can be eliminated from programs and that this can be done within certain bounds on loss in efficiency is given before starting our discussion of this controversy. Although this result may be of little help in deciding which way to program, it does show that it is possible to avoid side-effect primitives if that is your preference. This is the least which must be known before discussion is meaningful. Also, when considered in light of the actor semantics of this paper this fact can be seen to be true even in the presence of iteration, generators, and co-routines. We will present the result with informal argument rather than proof and then proceed with discussion of side-effects in various control and data structures.

Theorem: There is a uniform procedure for eliminating cells from programs with side-effects. It causes at most an $(n * (\ln n))$ vs n loss of efficiency in time, and at most a $(c * m)$ vs m increase in storage where c is a constant.

The general idea is that instead of having cells with contents which can be changed, the system has to keep track of these contents in some format which can be referenced by the cell's name. We call this object a *map*. Then whenever the contents of a cell is needed, it can be found by asking the *map*. Whenever the value of a cell named n_1 is to be updated to v , this change must be simulated without an actual side-effect. That is, the existing *map*, map_1 , can not be changed by the operation of simulating updating the contents of a cell. This can be accomplished by creating a *new map*, which we shall call map_2 , whose behavior is identical to that of map_1 except that when asked for the contents of n_1 it must respond with v . The new *map*, map_2 , must be the one examined in the future on all update and content operations on cells.

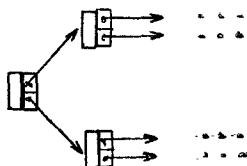
All statements involving cells must have access to the current *map*. The way to give them this access is to always pass around the current *map*. This can be done in applicative programming languages by passing around an additional argument, namely the *map*, to all functions. The *map* would then be available as a local variable in each procedure body.

In imperative languages this would be done by rewriting sequential code to pass the map from one statement to the next so that the simulated side-effects will propagate through the sequence. In both cases all creation of new cells, and all updating or accessing of the contents of cells will be simulated by operations on the current map.

In an actor based language in which the basic operation is the transmitting of a message actor to a target actor, the current map must be included in all messages.

We will continue to use the message passing metaphor to explain the bound on loss of efficiency. In accordance with this decision, time is measured by the number of transmissions which must occur. Transmissions can be thought of as taking one unit of time, although an arbitrary amount of real time may pass between transmissions, depending on how the program is being run. The addition of the map to each transmission only linearly increases the time taken. The important inefficiency comes in the number of transmissions needed to look up information in the map. The bound is based on a tree structured implementation of the map. It is well known that a tree with n nodes can be searched or copied in time proportional to $(\ln n)$. Thus at any time if there are m nodes in the map a simulation of a cell statement (creation, updating, accessing) causes extra transmissions in a number proportional to $(\ln m)$. At the n th transmission of the original computation there can be at most n transmissions in which a cell was created and the size of the map increased. Therefore a computation of n transmissions can be simulated in at worst $(n \ln n)$ transmissions.

The bound on space is based on one possible physical storage arrangement for the tree. If it is stored in the form



then if the original computation created m cells (realized physically in m location) the simulation requires $3m$.

Implications of the Theorem

Cells [also called *references*, *variables*, and *locations* in the literature] are objects which are very commonly used in current generation programming. The very common use of cells needs to be re-evaluated in the light of the discovery of styles of programming which do not require use of cells. The following are some points from both sides of the argument.

The use of cells can introduce a time variant behavior in systems which makes them harder to understand and debug. The time variant behavior can make it much harder to prove and verify properties of programs because *some* of the assumptions which have been established in the middle of a module as a result of conditional tests can be invalidated by invoking a procedure with side-effects. At the level of hardware the use of cells causes difficulties for multi-processor computer systems which use buffer [cache] memories. Unless expensive precautions are taken, two processors might load the contents of a cell into their respective caches and update the contents

inconsistently. In the actor model the intuitive notion of one object pointing to another object has been formalized into one actor *knowing about* another so that the former can send transmissions to the latter. The use of cells can introduce cycles into the graph of the *knowing about* relation for actors making the use of a reference count scheme for garbage collection impossible.

On the other hand the reference count scheme for garbage collection itself puts a substantial efficiency drain on systems and may not be the most economical way to do garbage collection. An alternative scheme is to run a conventional copying garbage collection in parallel with regular computation. Also a style of programming without cells creates more garbage than a style using cells. Many programmers have become accustomed to using cells in their programs and are reluctant to change without a very persuasive reason just as many of the programmers who have become used to using the *goto* are reluctant to give it up.

On a more philosophical level, some theoreticians note that a side-effect is a kind of behavior that is very important in the world in which we live and feel that this kind of behavior should be directly realized in the computational primitives available for use by programmers. Also there are cases, such as communication between two processes running in parallel, where cells are absolutely required.

Programming Styles which do not use Cells

Note that the Elimination of Cells Theorem applies to actor systems with explicit access to transmissions as opposed to the λ -calculus where there is explicit access only to the message part of a transmission. Thus it applies to systems with iteration, generators, and co-routines [e.g. systems as powerful as SIMULA-67 [Birstwhistle et al, 1974] which use quasi-parallelism but not real parallelism]. Each of the above control structures has its own programming style which does not involve the use of cells. We illustrate these styles and the trade-offs involved using simple examples below. This is similar to the use of continuations as in [Mazurkiewicz, 1971; Fischer, 1972].

Iteration and Side-effects

Iterative control structures can be realized in the λ -calculus and thus without cells. We will give a simple example to illustrate how this is done:

```
(label iterative-factorial
  (lambda [n a]
    (if (n = 1)
        then a
        else (iterative-factorial (n - 1) (n * a))))
```

Of course the above definition is also a *recursive* definition of *iterative-factorial* in the sense that it is defined *in terms of itself*. However, in terms of the actor semantics it is not a *recursive* definition in the sense of ALGOL-like languages i.e. it does not use up arbitrarily large amounts of intermediate working storage in order to do the computation. Using the actor semantics the following version of factorial is *recursive* in both senses of *recursive*:

```
(label recursive-factorial
  (lambda [n]
    (if (n = 1)
        then 1
        else (n * (factorial (n - 1))))))
```

The reason that storage increases is that the number of actors

currently pointed to ("known about") continues to increase. Actors which are pointed to are not eligible for garbage collection. After formalizing the knows about relation we can demonstrate that iterative-factorial only requires a bounded amount of working storage

Conceptually every actor a consists of two components which are *inextricably* bound together: a data part and an action part [called a script]. The data of an actor consists of a finite set of actors *known* to the actor and the script of the actor consists of a recipe of what to do if the actor is sent a transmission. We shall indicate that an actor a *knows* a set of actors A by the notation a^A . A fundamental restriction of the actor model of computation is that an actor t can only be sent a transmission if it is *known* to the transmitter. More formally the restriction can be expressed as follows: if an event

$$\langle t1^T m1^M \alpha \text{ ec} \rangle \rightarrow \langle t2^M \alpha \text{ ec} \rangle$$

occur in a behavior then, it must be the case that $t2$ and $m2$ are either the same as $m1$ or are *known* either to $t1$ or to $m1$. I.e. $t2$ and $m2$ are restricted to being in the set

$$\{m1\} \cup T1 \cup M1$$

An actor a^A created by an event $\langle t^T m^M \alpha \text{ ec} \rangle$ is restricted to knowing about a subset of the union of the actors *known* to t^T and the actors *known* to m^M . I.e. A is restricted to being a subset of $T \cup M$. Any other event E which refers to a^A must be causally preceded by the event which created t . We postulate that each newly created actor is not identical to any pre-existing actor. More formally, the requirement is that for any pre-existing actor x it must be the case that $x \neq a$ where $=$ is the usual equality relation. The above restrictions express a kind of locality or "compactness" restriction for actor systems.

We can demonstrate that iterative-factorial does not use arbitrarily large amounts of working storage by analyzing the computations of iterative-factorial and observing the structure of the *knows about* relation. In the course of the computation some temporary actors will be created but only one such temporary will be needed at any given instant. This establishes an upper bound on the amount of temporary storage needed.

Generators With and Without Cells

We shall demonstrate how to define generators with and without using cells by defining generators that produce the squares 1, 4, 9, 16, etc. in sequence on demand using a device called streams by Peter Landin [1965]. We will develop two implementations that use the same interface with the polymorphic operator *next* which requests the next element. The interface is defined in such a way that it will work with streams with side-effects [Balzer, 1971; Krutar, 1971; and Mitchell, 1971] as well as with functional streams [Landin, 1965]. The polymorphic operator *next* which makes this happen works by immediately turning around and passing the buck to *the-supply* using the message

(next: (else-complain-to: the-complaint-dept))

to politely ask *the-supply* for its next and to complain to the-complaint-dept if it doesn't have a next.

A stream consists of two components: a first which is the next element in the stream, and a rest which is another stream that holds the rest of the elements after the first. Below we define a stream that produces integers which are squares up to some limit. It produces the

next larger square each time it is asked for it, and returns the square along with a stream for generating more squares.

To construct a stream of squares,
 receive a sequence of
 a limit of how many squares to produce
 and a count from which to begin producing squares.
 Then return an actor which,
 when asked either to produce the next element
 or to complain to a complaint-dept if it doesn't have one,
 checks the count to see
 if it is greater than the limit
 and if so complains to the complaint-dept.
 Otherwise it
 returns a stream which has
 the square of the current count as its next element
 and rest which is a stream of squares
 with the same limit
 but whose count is one greater than the input count.

Below we define a polymorphic operator *print-elements* which prints the elements of any stream which it is given. For example

(print-elements (cons-stream-for-squares (limit: 3) (count: 1)))

will print 1, 4, 9, and then print "done."

To print-elements of a stream
 receive a stream.
 Ask the stream for its next.
 If it has a next,
 receive the stream and bind its first and rest.
 Print the first of the stream.
 Then print-elements of the rest of the stream.
 If the stream has no next,
 print "done."

Below we define a port [a stream with side effects] that produces square integers up to some limit. It produces the next larger square each time it is asked for its next returning itself [with its behavior modified by a side effect] as the "rest."

To construct a port for squares,
 receive a sequence of
 a limit of how many squares to produce
 and a count from which to begin producing squares.
 Let current-count be a cell that initially contains the initial count.
 Label the actor whose behavior is defined below as the-port.
 Return the-port which is an actor which,
 when asked either to produce the next
 or to complain to a complaint-dept if it doesn't have one,
 checks the contents of current-count to see
 if it is greater than the limit,
 and if so complains to the complaint-dept.
 Otherwise it
 increments the contents of the current count.
 Then returns a stream which has
 as its next element
 the square of
 the contents of current-count
 and rest which is the-port.

Note that we have defined the port in such a way that the polymorphic operator *next* can also be used to obtain squares from it. Thus in many cases the decision of which of the two implementations

of the square generator to use can often be deferred or even later reconsidered. For example

```
(print-elements (cons-port-for-squares (limit: 3) (count: 1)))
```

will also print 1, 4, 9, and then print "done." However, the behavior of `cons-stream-for-squares` is not exactly the same as that of `cons-port-for-squares`. Once a port for squares has been printed then it is "totally used up" and has no more useful behavior. There is no "backup" capability built into ports. The burden of providing such a capability if it is desired is borne by the user of the port. Debugging a system where ports are used as the basic communication primitive can be painful because useful debugging activities such as printing the contents of a port sometimes have unfortunate side-effects. In contrast a stream for squares does not change its behavior as a result of being printed and can be used for other purposes at the same time such as searching for successive squares whose sum is an odd prime. Generators can always be implemented almost as efficiently without the use of cells as by using cells except that avoiding the use of cells imposes more of a garbage collection problem.

Data Structures and Side-Effects

The *dialogue style* of programming used for generators is applicable to defining actors which behave like a variety of data structures such as stacks, queues, sets, dequeues, and bags, making it possible to implement them either with or without side-effects. For example (`pop the-stack`) can return a value of the form

```
(next: the-top-of-the-stack (rest: a-stack-without-the-top-element))
```

The technique is also applicable to implementing control structures [such as iteration and co-routines] which can be implemented either with or without cells.

Co-routines and Side-effects

Co-routines are a control structure which are often used by programmers in artificial intelligence and systems programming. They have recently found increasing popularity [Hewitt, 1971; Bobrow-Wegbreit, 1972; McDermott and Sussman, 1972; Davies, 1972; Hewitt, Bishop, and Steiger, 1973] in knowledge-based applications as the foundation for a *dialogue-based* programming style. Early Artificial Intelligence programs were mainly organized as multi-pass heuristic programs consisting of a pass of information gathering, a pass of constraint analysis, and a pass of hypothesis formation. It is now generally recognized that multi-pass organizations of this kind are inflexible because it is often necessary for information to flow across these boundaries at all stages of the processing.

Co-routines were introduced by Conway [1963] to convert the separate lexical and syntactic passes of two-pass a compiler into a single pass compiler. Using cells and gotos a co-routine control structure [in a language like PAL, GEDANKEN, or CONNIVER] can be built to do this in the following way. We assume that there are four cells known globally to both the lexical analyzer and the syntactic analyzer of the compiler:

```
messageto-lexical: cell for messages to lexical analyzer
messageto-syntactic: cell for messages to syntactic analyzer
resumelexical: cell for resume-points of lexical analyzer
resumesyntactic: cell for resume-points of syntactic analyzer
```

The system is initialized by setting `resumelexical` and `resumesyntactic`

to their respective starting locations. To start the system execute the following:

```
Update the contents of the cell messageto-lexical to be 'start
then goto the contents of the cell resumelexical!
```

Whenever the lexical analyzer has parsed the next lexical item L then it can resume the analyzer by executing the following:

```
Update the contents of resumelexical to be a-lexical-resume-point
which is defined below.
Then update the contents of messageto-syntactic to be L
and then goto the contents of resumesyntactic:
a-lexical-resume-point:
...
Read the contents of messageto-lexical:
...
Return valuelexical as the value of this lexical routine.
```

Whenever the syntactic analyzer needs more input it can resume the lexical analyzer by executing the following:

```
Update the contents of resumesyntactic to be a-syntactic-resume-point
which is defined below.
Update the contents of messageto-lexical to be 'next
and then goto the contents of resumelexical:
a-syntactic-resume-point:
...
Read the contents of messageto-syntactic:
...
Return valuesyntactic as the value of this syntax routine.
```

In order to explain the implications of passing transmissions instead of messages we shall attempt to translate the above scheme into the λ -calculus and show where it runs into problems. The idea is very similar to the one used in the Elimination of Cells Theorem in that we will increase the size of all the messages passed around inside the lexical analyzer to include `resumesyntactic` and increase the size of the messages passed in the syntax analyzer to include `resumelexical`. The system is started by sending

```
['start the-syntax-analyzer]
```

to the-lexical-analyzer. Whenever the lexical analyzer has parsed the next lexical item L then it can resume the syntax analyzer by executing the following:

```
(resumesyntactic
  L
  ( $\lambda$  [message resumesyntactic]
    ...
    [valuelexical resumesyntactic]))
```

Unfortunately, the above scheme has a bug in that by the rules of the λ -calculus the value

```
[valuelexical resumesyntactic]
```

will be sent to the syntax analyzer instead of to the lexical analyzer! The bug can be fixed to produce an efficient solution by using Landin's J operator as follows:

```
(let
  {[back-to-lexical = J( $\lambda$  [x] x)]}
```

```
(resumesyntactic
  L
  (λ [message resumesyntactic]
    ...
    (back-to-lexical [valuelexical resumesyntactic])))
```

Using transmissions we shall indicate how to eliminate the side effects involved in co-routining between the syntactic and lexical analyzers without severe loss of efficiency. To start the system send the message

[start the-syntax-analyzer]

to the-lexical-analyzer. Whenever the lexical analyzer has parsed the next lexical item L, then it can resume the syntax analyzer by sending the transmission

```
(apply: L
  (then-to: continuation))
```

to the-syntax-analyzer where continuation is an actor which receives transmissions from the syntax analyzer and resumes the work of the lexical analyzer. Whenever the syntax analyzer needs more input from the lexical analyzer it sends a transmission of the following form to the-lexical-analyzer

```
(apply: 'next
  (then-to: continuation))
```

When should Cells be Used?

We would like to re-emphasize that we do not advocate rewriting programs using the algorithm described in the proof. Such activity is just as pointless as systematically rewriting programs that use the goto construct by substituting while loops and boolean variables. Instead a style of programming that has certain definite advantages must be developed. The theorem does not imply that the dialogue style of programming which we discussed above can always be used to eliminate cells. Further research is needed on this question. There has been a shift in formalisms for artificial intelligence away from heavy reliance on side-effects as illustrated by the development of the context mechanism in QA-4 [Rulifson, 1972], defining PLANNER worlds in terms of actors [Hewitt, Bishop, and Steiger, 1973], and the communication between PLANNER-73 plans in terms of message passing [Smith and Hewitt, 1974].

A programming style that avoids the use of cells seems very attractive since the programs are just as efficient and seem more readable to many people. In the case of generators and co-routines, not using cells circumvents the pitfalls of using side-effects but imposes some additional garbage collection overhead. The proposal to completely avoid the use of cells in actors that behave like data bases seems more controversial because a great many data-base applications require parallel access to the data-base on the part of several activators. In all such cases the cells used should be encased within some kind of modular synchronization primitive such as *monitors* [Hoare, 1974] or *serializers* [Hewitt et al, 1974].

Acknowledgements

This research was sponsored by the MIT Artificial Intelligence Laboratory and Project MAC under a contract from the Office of Naval Research. We would like to thank Marilyn McLennan, David Marr, Allen Brown, Ben Kuipers and Drew McDermott for helpful comments which led to substantial improvements in the form and content of this paper.

Bibliography

- Balzer, R. M. "PORTS- A Method for Dynamic Interprogram Communication and Job Control" 1971 SJCC.
- Birstwhistle et al. "SIMULA Begin" 1974.
- Bobrow, D. and Wegbreit, Ben. "A Model for Control Structures for Artificial Intelligence Programming Languages" IJCAI-73. August, 1973.
- Conway, M. E. "Design of a Separable Transition-Diagram Compiler" CACM. July, 1963.
- Davies, D. J. M. "POPLER 1.5 Reference Manual" TPU Report No. 1. Theoretical Psychology Unit, School of Artificial Intelligence, University of Edinburgh. May, 1973.
- Dijkstra, E. W. "Solution of a Problem in Concurrent Programming Control" CACM 8, 9. Sept, 1965.
- Fischer, M. J. "Lambda Calculus Schemata" ACM Conference on Proving Assertions about Programs" Jan. 1972.
- Hewitt, C. "Description and Theoretical Analysis [Using Schemata] of PLANNER: A Language for Proving Theorems and Manipulating Models for a Robot" Phd. MIT. February, 1971.
- Hewitt, C., Bishop P., and Steiger, R. "A Universal Modular Actor Formalism for Artificial Intelligence" IJCAI-73. Stanford, Calif. Aug. 1973. pp. 235-245.
- Hoare, C. A. R. "An Axiomatic Basis for Computer Programming" CACM 12,10. Oct. 1969.
- Hoare, C. A. R. "Monitors: An Operating System Structuring Concept" CACM 17,10. Oct. 1974
- Knuth, D. E. "Additional Comments on a Problem in Concurrent Programming Control" CACM 9, 5. May 1966.
- Krutar, R. A., "Conversational Systems Programming" SIGPLAN Notices. Dec. 1971.
- Landin, P. J. "A Correspondence Between ALGOL 60 and Church's Lambda-Notation" CACM. February, 1965.
- Mazurkiewicz, A. "Proving Algorithms by Tail Functions" Information and Control. 1971.
- McCarthy, J. "A Formal Description of a Subset of Algol" in Formal Language Description Languages for Computer Programming. North Holland Publishing Company. 1964.
- McDermott, D. V., and Sussman G. J. "The Conniver Reference Manual" A.I. Memo No. 259. 1972.
- Mitchell, J. G. "The Modular Programming System: Processes and Ports" NIC 7359. June, 1971.

- Rulifson, Johns F, Derksen J. A., and Waldinger R. J. "QA4: A Procedural Calculus for Intuitive Reasoning" Phd. Stanford. November 1972.
- Scott, D. and Strachey, C. "Towards a Mathematical Semantics for Computer Languages" Oxford University Computing Laboratory. August, 1971.
- Smith, Brian and Hewitt, Carl. "Towards a Programming Apprentice" AISB Conference. July 1974.
- vanWijngaarden, A. , "Report on the Algorithmic Language ALGOL-68" Mathematische Centrum. 1969.