MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

WORKING PAPER 179 FEBRUARY 1979

CONCURRENT SYSTEMS NEED <u>BOTH</u> SEQUENCES <u>AND</u> SERIALIZERS

Carl Hewitt

ABSTRACT

Contemporary concurrent programming languages fall roughly into two classes. Languages in the first class support the notion of a sequence of values and some kind of pipelining operation over the sequence of values. Languages in the second class support the notion of transactions and some way to serialize transactions. In terms of the actor model of computation this distinction corresponds to the difference between <u>serialized</u> and <u>unserialized</u> actors. In this paper the utility of modeling both serialized and unserialized actors in a coherent formalism is demonstrated.

This report describes research done 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.

II -- INTRODUCTION

Contemporary concurrent programming languages fall roughly into two classes. Languages in the first class support the notion of a sequence of values and some kind of pipelining operation over the sequence of values. Examples of such languages are the lambda calculus [Church: 19??], LAMBDA [Scott: 19??], LISP [McCarthy: 1960], data flow [Weng: 1975], APL [Iverson 19??], Networks of Parallel Processes [Kahn and MacQueen: 1977]. Languages in the second class support the notion of transactions and some way to serialize transactions. Examples of language constructs which are designed to serialize transactions are semaphores [Dijkstra: 19??], monitors [Brinch Hansen: 19??; Hoare: 19??], and serializers [Hewitt and Atkinson: 1975], and Communicating Parallel Processes [Hoare: 1978].

The ACT1 language described in this paper is designed to support the implementation of both serialized and unserialized actors. The same communication mechanism is used to support both kinds of actors. A serialized actor can only process one message at a time. It is created in an initial state, and can change its state after each message which it receives. An unserialized actor can process arbitrarily messages in parallel and does not change state.

III -- SEQUENCE ACTORS

Sequences are a standard example of an unserialized actor. They can be constructed using closures in the lambda calculus languages. Alternatively languages like PURE LISP provide primitives for selecting the first element and the rest of the elements of a nonempty list as well as the ability to construct a new list given an object and a list. I will use the unpack notation described in an appendix to this paper to construct sequences and select their subparts.

III.1 -- A Concurrent Case Expression

Clearly some kind of conditional test is needed in implementations. Use will be made of ***select_case_for*** expressions of the following form:

```
(select_case_for expression
   (pattern1 then body1)
   ...
   (patternn then bodyn)
   [none_of_the_above: alternative_body])
```

which when evaluated first evaluates expression to produce a value V.

If the value V matches any of the pattern_i then the corresponding body_i is evaluated and its value is the value of the *select_case_for* expression. If the value V matches more than one of the pattern_i then an arbitrary one of the corresponding body_i is selected to be executed. However, if the value of expression can match two different patterns then the Programming Apprentice will warn the user if it cannot demonstrate that the results of executing the bodies are indistinguishable. This rule has the advantage that it makes body_i depend only on pattern_i making it easy add more selections later.

We shall say that two activities are concurrent if it is possible for them to occur at the same. The concurrent case statement facilitates efficient implementation by allowing concurrent matching of expression against the patterns. This ability is important in applications where attempts to determine whether or not conditions hold take large amounts of time.

If the value V does not match any of the pattern_i then alternative_body is executed. This rule provides the ability to have the patterns represent special cases leaving the alternative_body to deal with the general case if none of the special cases apply.

III.2 — Squash Double Asterisks

The first problem is to write a procedure [called squash_asterisks] which transforms a string of characters to replace very pair of consecutive asterisks "**" by an uparrow "↑".

```
(define (squash =s)
        ;to squash the double asterisks of a string s which is a sequence of characters
 [is: (a String)]
 [definition:
  (select_case_for s ;the rules for s are
     ((a String [characters: []]) then (a String [characters: []]))
        ;if s is the empty string then the value is the empty string
     ((a String [characters: [(an Asterisk) !=rest_s]]) then
        ;if the first character of s is an asterisk
        (select_case_for rest_s ;then the rules for the rest of s are
           ((a String [characters: []]) then ;if the rest of s is empty
              (a String [characters: [(an  Asterisk)]])) ;then the value is a string with one asterisk
           ((a String [characters: [(an asterisk) !=rest_rest_s]]) then
              (a String [characters: [(an Uparrow) !(squash rest_rest_s)]]))
              ;if the first character of the rest of s is an asterisk
              ;then the value is a sequence beginning with an uparrow followed by
              ;the result of squashing the rest of the rest of s
           ((a String [characters: [((¬ (an Asterisk)) ∧ =second_s) !=rest_rest_s]]) then
              ;if the first character of the rest of s is not an asterisk
              (a String [characters: [(an Asterisk) second_s !(squash rest_rest_s)]]))))
              ;then the value is a sequence beginning with an asterisk followed by
              ;the second character of s followed by
              ;the result of squashing the rest of the rest of s
     [(a String [characters: [((¬ (an Asterisk)) ∧ =first_s) !=rest_s]]) then
        ;if the first character of s is not an asterisk
        (a String [first_s !(squash rest_s)])])])
        ;then the value is the first character of s followed by the result of squashing the rest of s
```

III.3 — Disassemble a Sequence of Cards

The problem is to disassemble a sequence of cards to produce a sequence of the characters in the cards. An extra space should be inserted at the end of each card.

```
(define  (disassemble =s)
         ;to disassemble a card deck s inserting a blank at the end of each card
 [is: (a String)]
 [definition:
  (select_case_for s ;the rules for s are
     ((a Deck [cards: []]) then
        (a String [characters: []])) ;if s is empty then return the empty sequence of characters
     ((a Deck [cards: [(a Card [string: =cs]) !=rest_s]]) then
        (a String [characters: [!cs (a Blank) !(disassemble !=rest_s)]]))))])
```

III.4 --- Assembling a Line Printer Format Listing

The problem is to format a sequence of characters into a listing in line printer format which is a sequence of strings of 125 characters each with the last string completed with spaces if necessary. First we define a procedure that will produced a string with n blanks.

```
(define (blanks =n) ;to produce a string of n blanks
  [is: (a (String [length: n] [characters: (a Sequence [each_element: (a  Blank)])]))]
  [definition:
      (select_case_for n ;the rules for n are
          (0 then (a String [characters: []]))
             ;if it is zero then the value is the empty string
          ((> 0) then ;if n is greater than zero
             (a String [characters: [(a Blank) !(blanks (n - 1))]]))))])
               ;then the value is a string with a blank followed by n minus 1 blanks
```

Using the above procedure we can define the procedure to assemble a string into a listing as follows:

```
(define (assemble =s)
  [is: (a Listing [each_line: (a String [length: 125])])]
  [definition:
    (let ((a Sublisting [lines: =l] [buffer: =b]) be (subassemble s))
            ;the procedure subassemble is defined below
     then (a Listing [lines: [!l (a Line [characters: [!b (blanks (125 - (length b)))]])]]))])

(define (subassemble =s)    ;to subassemble a sublisting of s
  [is: (a Sublisting)]
  [definition:
    (select_case_for s ;the rules for s are
       ((a String [characters: []]) then ;if s is an empty string
              (a Sublisting
                    [lines: []]
                    [buffer: (a String [characters: []])]))
       ((a String [characters: [=first_s !=rest_s]]) then
          ;let first_s be the first character of s and rest_s be the rest of the characters of s
          (let
             ((a Sublisting [lines: =l] [buffer: =b]) be (subassemble (a String [characters: rest_s])))
           then
             (select_case_for (length b) ;the rules for the length of b are
                 ((< 125) then ;if it is less than 125
                     (a Sublisting
                           [lines: l]
                           [buffer: [!b first_s]]))
                 (125 then ;if the length of b is 125
                     (a Sublisting
                           [lines: [!l b]]
                           [buffer: (a String [characters: [first_s]])]))))))])
```

III.5 --- Reformat Card Sequence for Line Printer

The problem is to reformat a sequence of cards into a listing of lines of 125 characters each. The sequence of characters on each card should be followed by an extra space, and the last line of the listing should be completed with spaces if necessary.

```
(define (reformat =s)
   [is: (a Listing)]
   [definition:
      (assemble (disassemble s))])
```

III.6 --- Conway's Problem

The problem [Conway: 1963] is to adapt **reformat** to replace every pair of consecutive asterisks by an uparrow.

```
(define (printer_format_squashed_disassembled =s)
   [is: (a Listing)]
   [definition:
      (assemble (squash (disassemble s)))])
```

It is quite informative to compare the above solution with the one given by Hoare in his published paper on Communicating Sequential Processes. The programming styles are quite different but the amount of concurrency possible in the two implementations is similar. I believe that the programming style represented above constitutes an useful programming methodology which should be supported.

III.7 --- Multiple Output Sequences

One of the fundamental abilities of coroutines is the capability of combining operations so that multipass algorithms are coalesced into a single pass. A classic example is the **odds_and_evens** procedure which splits a sequence into two subsequences. For example **(odds_and_evens [1 3 2 4 1 5])** is **{[odd: [1 3 1 5]] [even: [2 4]]}**. We begin by defining the notion of an interleaving of two sequences:

```
(define ({=s1 =s2} is (an Interleaving [sequence: =s]))
  [definition:
    (xor ;either
      ({s1 s2 s} each_is []) ;s1, s2 and s are all empty sequences
      (and ;or
        (s is [=first_s !=rest_s]) ;s is not empty
        (or ;and
          (and
            (s1 is [first_s !=rest_s1]) ;the first element of s1 is the same as s
            ({rest_s1 s2} is (an Interleaving [sequence: rest_s])))
              ;and s2 and the rest of s1 is an interleaving of s
          (and ;or vice versa
            (s2 is [first_s !=rest_s2])
            ({s1 rest_s2} is (an Interleaving [sequence: rest_s])))))))])
```

Using the notion of interleaving we can give a partial specification of the problem to be solved and provide an initial implementation.

```
(define (odds_and_evens =s)
   [preconditions: (s is (a Sequence [each_element: (an Integer)]))]
   [is: {[odd: =o (a Sequence [each_element: (an Odd)])]
         [even: =e (a Sequence [each_element: (an Even)])]}]
   [constraints:
       ({|o e|} is (an Interleaving [sequence: s]))]
   [definition: {[odd: (odds s)] [even: (evens s)]}])
     ;where the functions odds and evens are defined below

(define (odds =s)
   [definition:
     (select_case_for s
             ([] then [])
             ([=first_s !=rest_s] then
                    (select_case_for first_s
                           ((an Odd) then [first_s !(odds rest_s)])
                           [none_of_the_above: (odds rest_s)])))])

(define (evens =s)
   [definition:
     (select_case_for s
             ([] then [])
             ([=first_s !=rest_s] then
                    (select_case_for first_s
                           ((an Even) then [first_s !(evens rest_s)])
                           [none_of_the_above: (evens rest_s)])))])
```

Unfortunately the above implementation requires that the input sequence be scanned twice: to look for odd numbers and to look for even numbers. The following implementation solves this problem:

```
(define (odds_and_evens =s)
   [preconditions: (s is (a Sequence [each_element: (an Integer)]))]
   [is: {[odd: =o (a Sequence [each_element: (an Odd)])]
         [even: =e (a Sequence [each_element: (an Even)])]}]
   [constraints:
      ({|o e|} is (an Interleaving [sequence: s]))]
   [definition:
     (select_case_for s
           ([] then {[odd: []] [even: []]})
           ([=first_s !=rest_s] then
              (let ({[odd: =odds_rest_s] [even: =evens_rest_s]} be (odds_and_evens rest_s)) then
              (select_case_for first_s
                 ((an Odd) then
                        {[odd: [!odds_rest_s first_s]]
                          [even: evens_rest_s]})
                 ((an Even) then
                        {[odd: odds_rest_s]
                          [even: [!evens_rest_s first_s]]})))))])
```

It is interesting to compare the above implementation with the style of programming in [Kahn and Macqueen: 1977 and Weng: 1975].

III.8 --- Advantages of Sequence Actors

Sequence actors have a number of advantages over previously published proposals for implementing coroutines.

III.8.a --- No Special Termination Rules

There are no special termination rules that must be imposed to make actor sequences behave properly. The empty sequence [] enjoys the same privileges as all other actors. Note that Communicating Special Processes have a special protocol to deal with terminating processes. Also the networks of parallel processes of [Kahn and Macqueen: 1978] which work with infinite streams. Both finite and infinite sequences are accommodated in a natural way within the programming paradigm based on actor sequences.

III.8.b --- Greater Concurrency

Actor sequences offer the possibility of greater concurrency than has been available in previous proposals for coroutines. We describe a Split of a sequence in the following way:

```
(describe (a Split [initial: =i] [final: =f])
  [is: [!i !f]]
  [constraints:
    (or
       (not (i is []))
       (not (f is []))))])
```

In other words a Split is a division of a sequence into an initial segment i and a final segment f such that at least one of them is not empty. Using the above construct we can define an actor which forms the product of all the elements in a sequence as follows:[1]

```
(define (product_of_elements =s)
  [definition:
    (select_case_for s
      ([] then 1)
      ((a Split [initial: =i] [final: =f]) then
            ((product_of_elements i) * (product_of_elements f))))])
```

The above implementation makes good use of one of the fundamental characteristics of actors: they are defined by their behavior, not their physical representation [Hewitt, Bishop, and Steiger: 1973]. A sequence can be split at any point that is convenient. This capability can be used to vastly increase the parallelism of some systems. Consider the product_of_elements of sequences formed by processes such as the one given below which generates a stream of the values of the leaves of a binary tree.

```
(define (fringe =t)
  [preconditions: (t is (a Binary_tree))]
  [definition:
    (select_case_for t
            ((a Leaf [value: =x]) then [x])
            ((a Nonterminal [left_subtree: =t1] [right_subtree: =t2]) then
                  [!(fringe t1) !(fringe t2)]))])
```

For example

1: This implementation is an adaptation of a suggestion of Henry Lieberman.

```
(product_of_elements
   (fringe
      (a Nonterminal
          [left_subtree:
             (a Nonterminal
                   [left_subtree: (a Leaf [value: 987])]
                   [right_subtree: (a Leaf [value: 89])])]
          [right_subtree:
             (a Nonterminal
                   [left_subtree: (a Leaf [value: 789])]
                   [right_subtree: (a Leaf [value: 98])])])))
```

will perform the multiplications (987 * 89) and (789 * 98) concurrently. This degree of parallelism is not possible using previously published proposals for coroutines. If enough processors are available, the above implementation using sequence actors can execute in the logarithm of the time of previously published coroutine mechanisms.

IV -- OTHER UNSERIALIZED ACTORS

IV.1 --- Recursive Subroutines

Recursive functions are often quite awkward to implement using serialized actors. Consider the problem of implementing (factorial n) as fast as possible. I.e. the problem is to compute

n * n-1 * ... * 2 * 1

Consider the following grouping of the above multiplications:

(n * n-1 * ... * n/2) * ((n/2 - 1) * (n/2 - 2) * ... * 2 * 1)

The multiplications in two outermost expressions can be performed in parallel. This idea can be applied recursively to obtain the following implementation:

```
(define (factorial =n)
  [preconditions: (n is (a Positive_integer))]
  [is: (an Integer)]
  [definition:
    (subfactorial n 1)])

(define (subfactorial =k =r)
  [preconditions:
      ({k r} each_is (a Positive_integer))
      (k ≥ r)]
  [is: ((factorial k) / (factorial r))]
  [definition:
    (select_case_for k
            (r then 1)
            ((r + 1) then k)
            ((> (r + 1)) then
                  (select_case_for (k + r)
                      ((an Even) then
                                (*
                                  (subfactorial k ((k + r) / 2))
                                  (subfactorial ((k + r) / 2) r)))
                      ((a Odd) then
                                (k * (subfactorial (k - 1) r)))))))])
```

The above implementation executes in the logarithm of the time of the iterative implementation on an actor machine which consists of a large number of processors connected by a high bandwidth network. The reader is urged to attempt to write the above implementation using serialized actors.

V -- SERIALIZERS

The guardians in this paper are implemented using primitive serializers which are a further development of serializers [Hewitt and Atkinson: 1977]. Primitive serializers have the advantage that they have been given a mathematical denotation [Hewitt and Attardi: 1978]. Primitive serializers are also more flexible in that they have less machinery built into them than previous serializers which gives them the ability to efficiently implement the facilities (such as queues) that were provided by previous serializers as well as to implement new facilities that were not provided before. Unlike previous serializers, primitive serializers can explicitly deal with actors which act as customers to whom replies should be sent.

At the same time primitive serializers maintain the advantages of serializers over other published proposals for synchronization primitives such as monitors [Hoare: 1974; Brinch-Hansen: 1973] and Communicating Sequential Processes [Hoare: 1978]. The examples considered in this paper are used to illustrate the advantages of using primitive serializers for specifying and proving properties of guardians.

VI -- Primitive Serializers

The design goals for monitors is that they were intended to be a structuring construct for implementing operating systems. There have been some attempts to develop useful proof rules for monitors [Howard: 1976; Gjessing: 1977; Hoare: 1974; Owicki: 1978] Serializers [Hewitt and Atkinson: 1976] are a further step toward these goals. However the language construct developed by Hewitt and Atkinson may be too complicated to be useful both as a formal foundation and as a basis for the proof methodology. In the study we present here the approach has been reversed. Instead of designing a desirable set of primitives and then trying to describe their semantics in a formal way, we started with a basic primitive with a simple semantics.

The syntax of a simple primitive serializer is the following:

```
(create_serialized_actor
    [state:
        [component_1: description_1]
        ...
        [component_n: description_n]]
    [initialize:
        [component_1 ← expression_1]
        ...
        [component_n ← expression_n]]
    [constraints: ...universally true properties declared here...]
    [receivers:
        (pattern_for_communication_1 received body_1)
        ...
        (pattern_for_communication_n received body_n)]
    [transitions:
        (define transition_1: transition_body_1)
        ...
        (define transition_k: transition_body_k)])
```

Serializers are introduced to create actors whose state may change after the receipt of a communication. A convenient way to express this is by means of the notion of state. The behavior of an actor depends on its (local) state, and its state may change as communications are received. The actor created by ***create_serialized_actor*** behaves in the following way. It can be either ***locked*** or ***unlocked***. When it is created it is ***unlocked***. When the first communication arrives, the serializer becomes ***locked*** and the body of a receiver in the **receivers** section whose pattern matches the communication received is selected for execution. Communications arriving while the serializer is ***locked*** are queued outside the serializer in order of arrival. When the serializer becomes unlocked the next communication to be processed is be taken from this queue. An important consideration in the design of efficient serializers is that they should remain locked for as short a time as possible.

Commands of the form (*transmit* t ← c) are used in receivers to transmit the communication actor **c** to the target actor **t**.

When a result of the form

```
(transition_i
        [component_1 ← expression_1]
        ...
        [component_n ← expression_n])
```

is computed then the serializer transfers control to **transition**$_i$ in a state with the state components component$_1$, ..., and component$_n$ having the values expression$_1$, ..., and expression$_n$ respectively.

When a result of the form

```
(unlock
        [component_1 ← expression_1]
        ...
        [component_n ← expression_n])
```

is computed then the serializer unlocks in a state with the state components component$_1$, ..., and **component**$_n$ having the values **expression**$_1$, ..., and **expression**$_n$ respectively.

Note that there are three separate events which must occur before a communication C can be received by a serialized actor T. First it must be transmitted in a transmission event of the form

(*a* Transmission [target: T] [communication: C])

Next it must arrive in a delivery event (synonymous with arrival event) of the form

(*a* Delivery [target: T] [communication: C])

Hardware modules called arbiters are used to establish an arrival ordering for all communications delivered to T. Finally it must be received in a receipt event of the form (*a* Receipt [recipient: T] [communication: C])

Communications are received in the order in which they are delivered. The receipt event marks a transition in which the target changes from unlocked to locked. Thus if a serialized actor becomes locked then no more messages can be received until it unlocks.

VI.1 --- A Concurrent Conditional Expression

The implementation of the hardcopy server given below makes use of a conditional construct of the following form:

```
(select_one_of
   (if condition_1 then body_1)
   ...
   (if condition_n then body_n)
   [none_of_the_above: alternative_body])
```

If any condition_i holds then the corresponding body_i is evaluated. If more than one of the condition_i hold then an arbitrary one of the corresponding body_i is selected to be evaluated. However, if it seems to the Programming Apprentice that more than one of the condition_i might hold simultaneously then it will warn the user if it cannot demonstrate that the execution of the corresponding body_i have equivalent effects. The rule of concurrent consideration of conditions encourages programs which are more robust, modular, easily modifiable, and efficient than is possible with the conditional expression in LISP for the reasons which are enumerated in the discussion of the *select_case_for* expression. If none of the condition_i hold then alternative_body is executed.

The reader will probably have noticed that the *select_one_of* construct is very similar to the *select_case_for* construct which we introduced earlier in this paper. The reason for introducing both constructs is that whereas the *select_case_for* construct is often quite succinct and readable there are cases such as the implementation below in which it is desirable to concurrently test properties of more than one actor in a single conditional expression making the use of *select_one_of* preferable.

The *select_one_of* expression is similar to the conditional expression of McCarthy modified in two important respects. The first modification is that conditions have been generalized to allow pattern matching as in the pattern directed programming languages PLANNER, QA-4, POPLER, CONNIVER, etc. The second modification is to consider the conditions concurrently.

VII -- Air Line Flights

The purpose of the flight reservation actor is to maintain information on the number of seats still available on a plane, the names of the passengers who have reservations, and the names of those waiting for a reservation. We assume that the flight reservation actor is sent messages by other computers connected by a network.

Two kinds of transactions will be handled by a flight actor. The first kind is initiated when the flight actor receives a communication of the form

(*a* Request [message: (*a* Booking [*person:* p])] [customer: c])

which requests that a booking be made for a person p and a reply as to whether or not the booking was made to be sent to the actor c. The possible replies that will be sent to c are that a new booking was made, that the person p was already booked, or that the person p has been waitlisted in case a seat becomes available later. The second kind of transaction is initiated by the receipt of a communication of the form

(*a* Request [message: (*a* Cancellation [*person:* p])] [customer: c])

which requests that a booking for a person p be canceled. The possible replies that will be sent to c are that the booking was canceled or that the person p was not booked.

An important specification which must be met is that a response is guaranteed to be sent to the customer c for each request received. Being part of a public utility, an airline flight is not allowed to ignore a request. We believe that the requirement that processes must reply requests which they receive is an important one in the specification of the behavior of concurrent systems. It is analogous to the requirement in serial programming that a procedure must return a value if it is provided with valid arguments. For example if a subroutine **factorial** is sent the following communication:

(*a* Request [message: 3] [customer: c])

then it is natural to require that c will be sent the following communication:

(*a* Response [reply: 6])

In exactly the same way we require that if the reservation system is sent a communication of the form

(*a* Request [message: (*a* Cancellation [*person:* p])] [customer: c])

then c must be sent a communication which satisfies the following description:

(*a* Response [reply: (booked ∨ waitlisted)])

Below we give an implementation of (create_flight capacity) which creates a new flight actor with the specified capacity. The implementation makes use of variables seats_left, reserved, and waiting which are respectively the number of seats left on the flight and the passengers who are currently booked, and the requests for bookings which are waiting for available seats.

```
(define (create_flight =capacity)     ;to create a flight with a given capacity
  [is:
     (a Serialized_actor [receives_messages: (either (a Booking) (a Cancellation))])
     (a Guaranteed_responder)]
  [definition:
   (create_serialized_actor
      [state:
         [seats_left: (an Integer)]
         [reserved: (a Set [each_element: (a Passenger)])]
         [waiting: (a Queue [each_element: (a Booking_request)])]]
      [initialize:
         [seats_left ← capacity]
         [reserved ← {}]
         [waiting ← empty_queue]]
      [receivers:
         ((a Request [message: (a Booking [person: =p])] [customer: =c]) received
                  ;if the request is to book a person p
            (select_case_for p
                  ((∈ passengers) then
                     ;if p is an element of the set of passengers
                     (transmit c ← [reply: already_booked]) ;reply that the person is already booked
                     (unlock))
                  (¬(∈ passengers) then   ;if p is not an element of passengers
                     (select_case_for seats_left
                        (0 then  ;if it is 0 then
                           (transmit c ← [reply: wait_listed])
                           (unlock [waiting ← (waiting enqueue the_request)]))
                        ((> 0) then  ;if seats_left is greater than 0
                           (transmit c ← [reply: booked])
                           (unlock
                              [passengers ← (passengers U {p})]
                                 ;add p to the set of passengers
                              [seats_left ← (seats_left - 1)])))))
                                 ;decrement the count of seats left
          ((a Request [message: (a Cancellation [person: =p])] [customer: =c]) received
            (select_case_for p
                  ((∈ passengers) then
                        (transmit c ← [reply: canceled])
                        (ponder
                           [seats_left ← (seats_left + 1)]
                           [passengers ← (passengers - {p})]))
                                 ;reply that the booking has been canceled
                  (¬(∈ passengers) then
                        (transmit c ← [reply: not_booked])
                        (unlock))))]
[transitions:
  (define ponder
   (select_one_of
     (if  (seats_left > 0)
      and
            (waiting is (a Queue
                              [front: (a Request
                                         [message: (a Booking [person: =p])]
                                         [customer: =c])]
                              [all_but_front: =rest_waiting]))
      then
            (transmit c ← [reply: booked])
            (unlock
                 [passengers ← (passengers ∪ {p})]
                 [seats_left ← (seats_left - 1)]
                 [waiting ← rest_waiting]))
   [none_of_the_above: (unlock)]))])])
```

We would now like to consider some of the interesting aspects of the above implementation.

VII.1 --- **Receipt of Reservation before Waitlist Notification**

Within the actor model of message passing, it is possible that a customer will receive a reply that they have a reservation before they receive a reply that they have been waitlisted. This models observed behavior in real networks with gateways as well as the behavior of ordinary mail systems.

VII.2 --- **Mutual Acquaintances**

Because of the ability of an airline flight to keep customers on a waiting list in case a seat becomes free, the customers become an acquaintance of the airline flight while at the same time the flight is an acquaintance of the customer. This situation of mutual acquaintanceship implies that reference counts cannot be relied on to reclaim storage effectively. Tom Knight and I have improved the methods for doing garbage collection in systems with large numbers of processors building on the work of [Bishop: 1977; Baker: 1977 and 1978; and Halstead: 1978].

VII.3 --- **Limitations of Procedure Calls as Communication Mechanism**

The above implementation points up one of the limitations of procedure calls as a communication mechanism. Normal procedure calls allow just one response for each request sent. However in the case of booking requests we sometimes want to send two responses.

VII.4 --- **Potentially Unbounded Nondeterminism**

Because of the guarantee of service, the above actor exhibits potentially unbounded nondeterminism [Hoare: 1978]. This is permissible within the CHRONON model for actor systems although it seems to cause grave difficulties for other mathematical models of concurrent systems. For example the possibility of unbounded nondeterminism is apparently inconsistent with the use of Egli-Milner Ordering as the basis for a mathematical model of concurrent systems.

VIII -- INTERRUPTS

In order to study the problem of interrupts we will consider the simple problem of implementing division using successive subtraction. The behavior of a serialized actor named **the_divider** created by **create_uninterruptible_divider** defined below is as follows. If it receives a communication asking it solve a division problem given a numerator **n**, divisor **d**, and partial quotient **q**, then if the divisor **d** is less than the numerator **n** then the customer is told the answer. Otherwise **the_divider** is sent a simpler version of the problem to solve.

```
(define uninterruptible_divider
 [is:
     (a Serialized_actor [receives: (a Request [message: (a Problem)])])
     (a Guaranteed_responder)]
 [definition:
   (create_serialized_actor
     [state:]

     [initialize:]

     [receivers:
          ((a Request
                    [message: (a Problem
                                        [numerator: =n]
                                        [divisor: =d]
                                        [partial_quotient: =q])]
                    [customer: =c])
          received
                   (select_case_for d
                     ((< n) then
                            (transmit c ← [reply: (an Answer [quotient: q] [remainder: r])])
                            (unlock))
                     ((≥ n) then
                            (transmit uninterruptible_divider ←
                                   (a Request
                                          [message: (a Problem
                                                         [numerator: (n -d)]
                                                         [divisor: d]
                                                         [partial_quotient: (q + 1)])]
                                          [customer: c]))
                            (unlock))))])])
```

Notice an important difference between the unsynchronized communication in the actor and the synchronized communication in communicating sequential processes: a serialized actor can send a message to itself, but if a communicating sequential process attempts to communicate with itself then deadlock immediately results.

Within the actor model of computation, it is easy to prove that the implementation given above has the guaranteed response property. That is if an event of the form

```
(a Delivery
      [target: uninterruptible_divider]
      [communication:
         (a Request [message: (a Problem)] [customer: c])])
```

occurs then an event of the form

```
(a Transmission
       [target: c]
       [reply: (a Answer)])
```

will occur. Unfortunately this guarantee is not by itself sufficient for my purposes. If I perform the following instruction

```
(transmit uninterruptible_divider ←
      (a Request
             [message: (a Problem
                          [numerator: 2^178]
                          [divisor: 3]
                          [remainder: 0])]
             [customer: Carl]))
```

then I would like not to be required to pay for the resulting computation! I would like to require that all computations which I initiate can be interrupted by me if I later change my mind. I agree to pay for all computation that has actually be performed before I interrupt plus a fixed fee agreed in advance for the interruption.

In this case my new stronger specification can be met by the following implementation:

```
(define interruptible_divider
 [is:
      (a Serialized_actor [receives:
                              (either
                                 (an Interrupt_request)
                                 (a Request [message: (a Problem)]))])
      (a Guaranteed_responder)]
 [definition:
    (create_serialized_actor
      [state: [interrupted: (a Boolean)]]

      [initialize: [interrupted ← false]]

      [receivers:
         ((an Interrupt_request [customer: =c]) received
           (transmit c ← [reply: interrupt_received])
           (unlock [interrupted ← true]))
         ((a Request
                [message: (a Problem
                               [numerator: =n]
                               [divisor: =d]
                               [partial_quotient: =q])]
                [customer: =c])
         received
          (select_one_of
           (if (not interrupted)
               then
                (select_case_for d
                  ((< n) then
                        (transmit c ← [reply: (an Answer [quotient: q] [remainder: r])])
                        (unlock))
                  ((≥ n) then
                        (transmit interruptible_divider ←
                              (a Request
                                   [message: (a Problem
                                                 [numerator: (n -d)]
                                                 [divisor: d]
                                                 [partial_quotient: (q + 1)])]
                                   [customer: c]))
                        (unlock))))
           (if interrupted then
                        (transmit c ← [complaint: (an Interruption)]))))]]])
```

The behavior of a serialized actor named **interruptible_divider** created by **create_interruptible_divider** is as follows. If it receives a communication asking it solve a division problem given a numerator **n**, divisor d, and partial quotient q before it is interrupted, then if the divisor d is less than the numerator n then the customer is told the answer. Otherwise interruptible_divider is sent a simpler version of the problem to solve.

IX -- CONCLUSIONS

Our research paradigm has been to discover and characterize the computations that are physically possible using the actor model of computation. It began with an intuitive account [Hewitt, Bishop, and Steiger: IJCAI-73] of the benefits that might be obtained from this approach. Important progress has been made [Hewitt and Baker: 1977, Baker: 1978] in characterizing the Actors were developed to synthesize a unified semantics that combined the message passing, pattern matching, and pattern directed invocation and retrieval in PLANNER [Hewitt: 1969; Sussman, Charniak, and Winograd: 1971; Hewitt: 1971], the modularity of SIMULA [Birtwistle et. al.: 1973, Palme: 1973], the message passing ideas of an early design for SMALLTALK [Kay: 1972], the functional data structures in the lambda calculus based programming languages, the concept of concurrent events from Petri Nets (although the actor notion of an event is rather different than Petri's), and the protection inherent in the capability based operating systems with their protected entry points.

The work which we have done on programming languages grows out of this semantic basis. PLASMA [Hewitt: 1977, Hewitt and Smith: 1975] adopted the ideas of pattern matching, message passing, and concurrency as the core of the language. The ACT1 language described in this paper and ETHER-0 [Kornfeld: 1979, Hewitt: 1979] represent the latest generation of the experimental languages which we have implemented.

An important issue that must be faced in any model of concurrent computation is resolving the potential conflict when an actor receives two communications at approximately the same time. One possibility is that the two communications do not interact in any way. This choice is made in the lambda calculus, pure recursive subroutines, streams [Kahn and MacQueen: IFIP-77 and Weng: 1975], and unserialized actors. Another possibility is that the communications must be serialized so that the reception of one is delayed until the actor is ready to receive it. This choice is made in monitors, Communicating Sequential Processes, and serialized actors. One of the achievements of the actor model is to combine both serialized and unserialized actors in a unified semantics of concurrent systems with a mathematical semantics in which each actor has a denotation.

This paper has attempted to support the thesis that both serialized and unserialized actors are needed in concurrent systems. The examples in this paper show how sequences and recursive subroutines (implemented as unserialized actors) can be used to vastly increase the concurrency possible in many computations. On the other hand there are some specifications such as the ones for the air line system and interruptible divider considered in this paper which can only be implemented using serialized actors. In addition apparently there are some functions which are most efficiently implemented using both serialized and unserialized actors.

X -- ACKNOWLEDGEMENTS

I would like to thank Tony Hoare for suggesting the topic of this paper. Conversations with Gilles Kahn, Dave MacQueen, and Jerry Schwarz were particularly helpful in writing the section on sequences. Dave MacQueen and Richard Waters made some extremely perceptive and helpful comments which markedly helped us to improve the presentation in this paper. The hospitality and collaboration of Luigia Aiello, Stein Gjessing, Tony Hoare, Dave MacQueen, Kristen Nygaard, Gianfranco Prini, Jerry Schwarz, and Bob Tennent in the summer of 1978 greatly facilitated the development of the ideas in this paper.

XI -- BIBLIOGRAPHY

Atkinson R. and Hewitt, C. "Specification and Proof Techniques for Serializers" IEEE Transactions on Software Engineering. 1978. To appear.

Baker, H. H. Jr. "Actor Systems for Real-time Computation" MIT/LCS/TR-197. MIT Technical Report. March 1978.

Baker, H. J. Jr. and Hewitt, Carl "Incremental Garbage Collection of Processes" MIT A.I. Memo 454. December, 1977.

Bishop, P. B. "Computer Systems with a Very Large Address Space and Garbage Collection" MIT/LCS/TR-178. MIT Technical Report. May 1977.

Conway, M. E. "Design of a Separable Transition-Diagram compiler" CACM. Vol 6. No. 7. July 1963. pp 396-408.

Dijkstra, E. W. "Guarded Commands, Nondeterminancy, and Formal Derivation of Programs" CACM. Vol. 18. No. 8. August 1975. pp 453-457.

Good,D. I.; London, R. L.; and Bledsoe, W. W. "An Interactive Verification System" IEEE Transactions on Software Engineering. Vol. 1. 1975. pp 59-67.

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

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

Hewitt, C. "Viewing Control Structures as Patterns of Passing Messages" A.I. Journal. Vol. 8. No. 3. June 1977. pp. 323-364.

Hewitt, C. "Using Message Passing in Concurrent Programming" MIT Artificial Intelligence Working Paper. April 1977.

Hewitt, C. and Atkinson, R. "Synchronization in Actor Systems" Proceedings of Conference on Principles of Programming Languages. January 1977. Los Angeles, Calif.

Hewitt, C. and Attardi, G. "An Axiomatic Denotation Specification of a Concurrent Programming Language" MIT Working Paper. April 1978.

Hewitt, C. and Smith, B. "Towards a Programming Apprentice" IEEE Transactions on Software Engineering. SE-1, 1. March 1975. pp. 26-45.

Hewitt, C. and Baker, H. "Laws for Communicating Parallel Processes" IFIP-77. Toronto. August 1977. pp 987-992.

Hewitt, C. and Baker, H. "Actors and Continuous Functionals" IFIP Working Conference on Formal Description of Programming Concepts" August 1-5, 1977. St. Andrews, New Brunswick, Canada. MIT A. I. Memo 436A. MIT/LCS Technical Report 194. December 1977.

Hoare, C. A. R. "Communicating Sequential Processes" Department of Computer Science, The Queen's University, Belfast. March 1977.

Hoare, C. A. R. and McKeag, R. M. "Structure of an Operating System" Second Draft. October 1977.

Ingalls, D. H. H. "The Smalltalk-76 Programming System Design and Implementation" Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages. January 23-25, 1978. Tucson, Arizona. pp. 9-16.

KahnG. and MacQueen, D. B. "Coroutines and Networks of Parallel Processes" IFIP-77. Montreal. August 1977. pp 993-998.

Manna, Z. and McCarthy, J. "Properties of Programs and Partial Function Logic" Machine Intelligence 5. Edinburgh University Press. 1970.

McCarthy, J. "Recursive Functions of Symbolic Expressions and their Computation by Machine - I" CACM. Vol 3. No. 4. April 1960. pp 184-195.

McCarthy, J. "A Basis for a Mathematical Theory of Computation" in Computer Programming in Formal Systems P. Braffort and D. Hirschberg (eds.). North Holland. 1963. pp. 33-70.

Paterson, M. S. and Hewitt, C. E. "Comparative Schematology" ACM Conference Record of Working Conference on Concurrent Systems and Parallel Computation. 1970. Available from ACM.

Shoch, J. F. "An Overview of the Programming Language Smalltalk-72". Convention Informatique 1977. Paris, France.

Standish, T. A.; Harriman, D. C.; Kibler, D. F.; and Neighbors, J. M. "The Irvine Program Transformation Catalogue". Department of Information and Computer Science. University of California at Irvine. January 1976.

Strong, H. R. "Translating Recursion Equations into Flow Charts" Journal of Computer and System Sciences. June 1971. pp 254- 285.

Ward,S. A. "Functional Domains of Applicative Languages" Project MAC TR-136. September 1974.

Weng,K. "Stream-oriented Computation in Recursive Data Flow Schemas" October 1975. MIT Project MAC Technical Memorandum 68. October 1975.

Yonezawa, A. "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics" MIT/LCS/TR-191. December, 1977.

APPENDIX I --- The Unpack Construct

Our description system makes use of the unpack construct [Hewitt: 1971] which is denoted by "!" in relations. The unpack construct is used inside a sequence s to indicate that all of the elements of the sequence which follows "!" are elements of s. For example if x is the sequence [3 4] and y is the sequence [9 8] then the following equivalences hold:

```
   [x !y]  =  [[3 4] ![9 8]]  =  [[3 4] 9 8]
  [!x !y]  =  [![3 4] ![9 8]]  =  [3 4 9 8]
[!x 5 !x]  =  [![3 4] 5 ![3 4]]  =  [3 4 5 3 4]
```

The unpack operator is also used in pattern matching:

if the pattern [=x !=y] *is matched against* [1 2 3] *then*
x *is bound to* 1 *and*
y *is bound to* [2 3]

if the pattern [4 =x !=y] *is matched against* [4 5] *then*
x *is bound to* 5 *and*
y *is bound to* []

if the pattern [=x [4 !=y] !=z] *is matched against* [9 [4 3] 7 8 9] *then*
x *is bound to* 9
y *is bound to* [3] *and*
z *is bound to* [7 8 9]

APPENDIX II --- Implementation of Cells using Serializers

In this appendix I present an implementation of cells [Greif and Hewitt: POPL-75, Hewitt and Baker: IFIP-77] using primitive serializers.

```
(define (create_cell =initial_contents)
  [is: (a Serialized_actor [receives_messages: (either (a Contents_query) (an Update))])]
  [definition:
    (create_serialized_actor
      [state:
        [current_contents: (an Actor)]]
      [initialize:
        [current_contents ← initial_contents]]
      [receivers:
        [(a Contents_query) then
            (unlock [reply: current_contents])] ;unlock the serializer for the next message
        [(an Update [next_contents: =n] then
            (unlock [reply: did_update] [current_contents ← n])]])])
             ;unlock the serializer with the current contents being n
```

The above definition shows how serializers subsume the ability of cells to efficiently implement synchronization and state change in concurrent systems.

APPENDIX III --- Implementation of Semaphores using Serializers

Semaphores are an unstructured synchronization primitive that are used in the implementation of some systems. The definition below shows how primitive serializers can be used to efficiently implement semaphores.

```
(define (create_semaphore)
   [is: (a Serialized_actor [receives_messages: (either (a P_request) (a V_request))])]
   [definition:
      (create_serialized_actor
          [state:
             [q: (an Queue [each_element: (a customer)])]
             [capacity: (a Non_negative_integer)]]
          [initialize:
             [q ← (a Queue [sequence: []])]  ;initially there are no waiting P requests
             [capacity ← 1]]  ;the capacity is initially 1
          [receipt_invariants:
             (xor
                 (capacity > 0)
                 (q is (a Queue [sequence: ¬[]])))]
          [receivers:
             [(a Request [message: P] [customer: =c]) received
                 (ponder [q ← (q enqueue c)])]
                   ;ponder is a transition defined below
             [(a  Request [message: V] [customer: =c]) received
                 (transmit c ← [reply: Did_V])
                 (ponder [capacity ← (capacity + 1)])]]
          [transitions:
             [(define ponder
                     (select_one_of
                         (if (q is (a Queue
                                            [front: =c]
                                            [all_but_front: =rest_waiting_customers]))
                          and
                                (capacity > 0)
                          then
                                (transmit c ← [reply: Did_P])
                                (unlock
                                          [capacity ← (capacity - 1)]
                                          [q ← rest_waiting_customers]))
                         (if (capacity is 0) then (unlock))))]])])
```

In [Hoare: 1975] there is an elegant construction showing that monitors can be implemented using semaphores.

APPENDIX IV --- Thumbnail Sketch of the Description System

This appendix presents a brief sketch of the syntax and semantics of our description system. A paper which more fully presents the description system and compares it with other formalisms which have been proposed is in preparation.

XI.1 --- Syntax

If <x> is a syntactic category then an expression of the form <x>* will be used to denote an arbitrary sequence of zero or more items separated by blanks in the syntactic category <x>. An expression of the form <x>† will be used to denote an arbitrary sequence of one or more items separated by blanks in the syntactic category <x>.

The following is the syntax for descriptions and statements:

```
<description> ::= <identifier> |
                  =<identifier> |      ;the character = is used to mark local identifiers
                  <statement> |  ;note that statements (which are described below) are descriptions
                  <attribute_description> |
                  <instance_description> |
                  <criterial_description> |
                  <mapping_description> |
                  <sequence_description> |
                  <set_description> |
                  <multiset_description> |
                  <instance_description> |
                  (<description> viewed_as <description>) |
                  (<description> if <statement>) |
                  (<description> that_is <description>) |
                  (<description> such_that <statement>) |
                  (∧ <description>†) |
                  (v <description>†) |
                  (either <description>†) |
                  ¬<description> |
                  (<relation> <description>*)
```

```
<criterial_description> ::= (the_only <description>†)

<instance_description> ::= <indefinite_instance> | <definite_instance>
<indefinite_instance> ::= (<indefinite_article> <concept> <attribution>*)
<definite_instance> ::= (the <concept> <attribution>*)
        ;definite_instances are used only for criterial descriptions
<indefinite_article> ::= a | an
        ;there is no semantic significance attached to the choice of which article is used
<concept> ::= <description> ;note that this is ω order

<attribution> ::= [<binary_relation_description>†: <description>†]
<binary_relation_description> ::= <description>  ;note that this is ω order

<attribute_description> ::= <projective_attribute_description> |
                            (<indefinite_article> <binary_relation_description> of <description>)
<projective_attribute_description> ::= (the <binary_relation_description> of <description>)
    ;expresses that <binary_relation_description> is projective for <description>
    ;see example below for an explanation of projective binary relations

<mapping_description> ::= [<description>†⟼ <description>†]

<sequence_description> ::= [<elements_description>*] |
<set_description> ::= {<elements_description>*} |           ;{ and } are used to delimit sets
<multiset_description> ::= {|<elements_description>*|} | ;{| and |} are used to delimit multisets
<elements_description> ::= ... |
                           <description> |
                           !<description>  ;! is the unpack construct

<statement>   ::= (<predicate> <description>*) |
                  <predication> |
                  (<description> coref <description>) | ;statement of coreference
                  ({<description>*} each_is <description>) |
                  (and <statement>†) |
                  (or <statement>†) |
                  (xor <statement>†) |
                  (not <statement>) |
                  (implies <statement> <statement>)

<predication> ::= (<subject> is <complement>)
<subject> ::= <description>
<complement> ::= <description>
```

Note that the syntax of our description system reads somewhat like template English [Hewitt: 1975, Bobrow and Winograd: 1977, Wilks: 1976] Thus for example we write (*an* Integer) in this paper instead of writing that (integer) as was done in PLANNER-71. However we also allow the use of instance descriptions such as (*the* Integer [>: 0] [<: 2]) to describe the Integer which is greater than 0 and less than 2.

We feel that it is quite important that a description expressed in template English correspond in a natural way with the intuitive English meaning. For this reason we use the indefinite article in attribute descriptions of such as the one below:

(4 *is* (*an* element *of* {2 4 6}))

where the binary relation element can occur multiply in an instance description such as the following:

(({2 4 6} *is* (*a* Set [element: 2] [element: 4])))

Attribute descriptions only make use of the definite article in cases like the one below

((*the* imaginary_part *of* (*a* Real)) *is* 0)

where the binary relation imaginary_part projectively selects the imaginary part of a Real. In this case the relation imaginary_part might be inherited from Complex via the following description:

((*a* Real) *is* (*a* Complex [imaginary_part: 0]))

For the purpose of describing mappings, I prefer the syntax

$$[=x \longmapsto \ldots x \ldots]$$

[cf. Bourbaki: Book I, Chapter II, Section 3] to the syntax

$$(\lambda\, x.\ \ldots x \ldots)$$

of the lambda calculus. For example the mapping cubes which takes a number to its cube can be described as follows:

(*describe* cubes
  [is: [=n$\longmapsto$ n^3]])

XI.2 --- **Axioms**

The behavioral semantics of the description system is defined by its underlying message-passing semantics. The axiomatization given below is significant in that it represents a first attempt to axiomatize a description system of the power of the one described here. As far as I know previous to the development of this one, there did not exist similar axiomatizations for FRL, KRL, OWL, MDS, etc.

The most fundamental axiom is Transitivity of Predication which says that for any <description$_3$>

Transitivity of Predication

```
(implies
   (and
      (<description_1> is <description_2>)
      (<description_2> is <description_3>))
   (<description_1> is <description_3>))
```

Another fundamental axiom is Reflexivity of Predication which says that for any <description>

Reflexivity of Predication

```
(<description> is <description>)
```

Other important axioms are Commutativity, Deletion, and Merging:

Commutativity

```
((a <description_1> <attributions_1> <attribution_2> <attributions_3> <attribution_4> <attributions_5>) is
     (a <description_1> <attributions_1> <attribution_4> <attributions_3> <attribution_2> <attributions_5>))
```

which says that the order in which attributions of a concept are written is irrelevant,

Deletion

```
((a <description_1> <attributions_1> <attributions_2>) is (a <description_1> <attributions_2>))
```

which says that attributions of a concept can be deleted, and

Merging

```
(implies
   (and
      (<description_1> is (a <description_2> <attributions_1>))
      (<description_1> is (a <description_2> <attributions_2>)))
   (<description_1> is (a <description_2> <attributions_1> <attributions_2>)))
```

which says that attributions of the same concept can be merged.

Additional axioms are given below for other descriptive mechanisms:

Coreference

```
(<description_1> coref <description_2>) if and only if
    (<description_1> is <description_2>) and (<description_2> is <description_1>)
```

Criteriality

```
(implies
      (and
            (<description_1> is (the_only <description_3>))
            (<description_2> is (the_only <description_3>)))
      (<description_1> coref <description_2>))
```

Definite Selection

((*the* <description$_1$> *of* (*a* <description$_2$> [<description$_1$>: <description$_3$>])) *is* <description$_3$>)

Indefinite Selection

(<description$_1$> *is* (*a* <description$_2$> [<description$_3$>: <description$_4$>])) *if and only if*
    (<description$_4$> *is* (*a* <description$_3$> *of* (<description$_1$> *viewed_as* (*a* <description$_2$>))))

Constrained Description

(<description$_1$> *is* (<description$_2$> *such_that* <statement>)) *if_and_only_if*
    (*and*
        (<description$_1$> *is* <description$_2$>)
        <statement>)

Qualified Description

(<description$_1$> *is* (<description$_2$> *that_is* <description$_3$>)) *if_and_only_if*
    (*and*
        (<description$_1$> *is* <description$_2$>)
        (<description$_1$> *is* <description$_3$>))

View Point

((<description$_1$> *viewed_as* <description$_2$>) *is* (<description$_2$> *such_that* (<description$_1$> *is* <description$_2$>)))

Negation

(<description$_1$> *is* ¬<description$_2$>) *if_and_only_if*
    (*not* (<description$_1$> *is* <description$_2$>))

Conjunction

(<description$_1$> *is* (∧ <description$_2$> <description$_3$>)) *if_and_only_if*
    (*and*
        (<description$_1$> *is* <description$_2$>)
        (<description$_1$> *is* <description$_3$>))

Inclusive Disjunction

(<description$_1$> *is* (∨ <description$_2$> <description$_3$>)) *if_and_only_if*
    (*or*
        (<description$_1$> *is* <description$_2$>)
        (<description$_1$> *is* <description$_3$>))

Exclusive Disjunction

(<description$_1$> *is* (*either* <description$_2$> <description$_3$>)) *if_and_only_if*
    (*xor*
        (<description$_1$> *is* <description$_2$>)
        (<description$_1$> *is* <description$_3$>))

Conditional Description

(<description$_1$> *is* (<description$_2$> *if* <statement>)) *if_and_only_if*
    (<statement> *implies* (<description$_1$> *is* <description$_2$>))

XI.3 --- Examples

XI.3.a --- Articulation

Additional axioms hold for each of the primitive descriptive mechanisms of the system. For example

(*describe* cubes
    [is: (*a* Mapping [=n⟼ n^3])])

can be articulated as follows:

(cubes *is* (*a* Mapping [1⟼ 1] [2⟼ 8] [3⟼ 27] [4⟼ 64] [5⟼ 125] ...))

where ... is ellipsis.

XI.3.b --- Sets and Multisets

Sets and multisets can be described in terms of mappings using the usual mathematical isomorphisms. For example

(*describe* {a b}
    [is: (*a* Mapping [a⟼ 1] [b⟼ 1] [¬a ∧ ¬b⟼ 0])])

describes the set {a b} as a mapping from a and b onto 1 since they are present in the set and everything else maps to 0 since there are no occurrences of other elements. Extending the same idea to multisets gives the following example:

(*describe* {|a b a|}
    [is: (*a* Mapping [a⟼ 2] [b⟼ 1] [¬a ∧ ¬b⟼ 0])])

XI.3.c --- Transitivity

If (3 *is* (*an* Integer [<: 4])) and (4 *is* (*an* Integer [<: 5])), it should be possible to conclude that (3 *is* (*an* Integer [<: 5])). This goal can be accomplished by the command

(*describe* <
   [is: (*a* Transitive_relation [for: Integer])])

which says that < is a transitive relation for Integer and by the command

```
(describe (a =concept [=R: (a =concept [=R: =m])])
   [preconditions: (=R is (a Transitive_relation [for: =concept]))]
   [is: (a concept [R: m])])
```

which says that if x is an instance of a concept which has a relationship R with something which is the same **concept** which has the the relationship R with m where R is a transitive relationship for **concept**, then **x** has the relationship R with m. This example of transitivity cannot be done in most type systems; the above solution makes use of the ω-order capabilities of our description system.

XI.3.d --- Projective Relations

If (z *is* (*a* Complex [real_part: (> 0)])) and (z *is* (*a* Complex [real_part: (*an* Integer)])) then by merging it follows that (z *is* (*a* Complex [real_part: (> 0)] [real_part: (*an* Integer)])) However in order to be able to conclude that (z *is* (*a* Complex [real_part: (> 0) (*an* Integer)])) some additional information is needed. One very general way to provide this information is by

```
(describe real_part
   [is: (a Projective_relation [concept: Complex])])
```

and by the command

```
(describe (a =C [=R: =description1] [=R: =description2])
   [preconditions: (R is (a Projective_relation [concept: C]))]
   [is: (a C [R: description1 description2])])
```

The desired conclusion is reached by using the above description with C with bound to **Complex**, **R** bound to **real_part**, description1 bound to (> 0), and description2 bound to (*an* Integer).

XI.3.e --- Quantification and Existence

The treatment of local identifiers in our description system differs in an important respect from the treatment of universally quantified variables in naive ω-order logic where universal quantification implies existence. For example the following sentence clear holds in ω order logic:

$$\forall P\ \forall x\ (P\ x)\ \textit{if and only if}\ (P\ x)$$

From the above sentence the following follows by the usual rule for quantifiers:

$$\forall P\ \exists Q\ \forall x\ (Q\ x)\ \textit{if and only if}\ (P\ x)$$

Using the following definition for P

```
(define (P =x)
  [definition: (not (x x))])
```

we get

$$\exists Q\ \forall x\ (Q\ x)\ \textit{if and only if}\ (\textit{not}\ (x\ x))$$

Using ∃-elimination with Q_0 for Q we get

$$\forall x\ (Q_0\ x)\ \textit{if and only if}\ (\textit{not}\ (Q_0\ Q_0))$$

Substituting Q_0 for x we obtain Russell's paradoxical formula:

$$(Q_0\ Q_0)\ \textit{if and only if}\ (\textit{not}\ (Q_0\ Q_0))$$

However the above formula is a contradiction in our description system only if $(Q_0\ Q_0)$ is a **Boolean** which are described as follows:

```
(describe (a Boolean)
  [is: (either true false)])

(describe true
  [is:
     ¬false
     (a Boolean)])

(describe false
  [is:
     ¬true
     (a Boolean)])
```

We propose to restrict the rules of logic to statements which are Boolean. For example the rule of double negation elimination can be expressed as follows:

```
(describe (not (not =p))
   [precondition: (p is (a Boolean))]
   [is: p])
```

In this way we hope to avoid contradictions in our description system. In the course of the next year we will attempt to adapt one of the standard proofs to demonstrate its consistency.

XI.4 — Description of Queues

Queues play an important role in the implementation of nontrivial guardians which must schedule access to their resources. In this section we formally describe unserialized queues [Hewitt and Smith: 1975].

```
(describe (a Queue)     ;a Queue
    [is: (a Queue [sequence: (a Finite_sequence)])])
              ;has an attribute called sequence which is the sequence of elements

(describe empty_queue                    ;the empty_queue
   [is: (a Queue [sequence: []])])       ;is a Queue with the empty sequence of elements
```

Describing descriptions enables the user to declare the properties of new attributes that are conditional on other attributes. For example a nonempty queue has attributes named **sequence**, **front**, **all_but_front**, **rear**, and **all_but_rear** which are conditional on the **sequence** being nonempty.

```
(describe (a Queue [sequence: ¬[]])
   [is:
    (a Queue
         [front: (an Actor)]
         [all_but_front: (a Queue)]
         [rear: (an Actor)]
         [all_but_rear: (a Queue)])])
```

We can further describe Queues to state relationships among these attributes using the **unpack** construct for sequences [see appendix]:

```
(describe
    (a Queue
         [sequence: =sequence]
         [front: =the_front]
         [all_but_front: (a Queue [sequence: =sequence_of_all_but_front])]
         [rear: =the_rear]
         [all_but_rear: (a Queue [sequence: =sequence_of_all_but_rear])])
  [is: ¬empty_queue]
  [postconditions:
     (sequence is [the_front !sequence_of_all_but_front])
     (sequence is [!sequence_of_all_but_rear the_rear])])
```

The description of the result of enqueuing an element x on a queue is described as follows:

```
(describe ((a Queue [sequence: =s]) enqueue =x)
   [is: (a Queue [sequence: [!s x]])])
```