Working Paper 164c

Revised May 1979

Evolving Parallel Programs

Carl Hewitt

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Working Papers are informal papers intended for internal use.

This report describes research conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Office of Naval Research of the Department of Defense under Contract N00014-75-C-0522.

Evolving Parallel Programs

Carl Hewitt

M.I.T.

Room 813
545 Technology Square
Cambridge, Mass. 02139
(617) 253-5873

ABSTRACT

Message passing is directed toward the production of programs that are intended to execute efficiently in a computing environment with a large number of processors. The paradigm attempts to address the computational issues of state change and communication directly with appropriate primitives. Efficient programs are evolved for fast factorial and path existence determination in a directed graph.

This paper is a contribution to the continuing debate on programming methodology. It advocates that simple initial implementations of programs should be constructed and then the implementations should be **evolved** to meet their partial specifications where it is anticipated that the partial specifications will themselves evolve with time.

The programming methodology used in this paper is intended for use with an actor machine which consists of a large number of processors connected by a high bandwidth network. We evolve implementations for factorial and for the path existence problem that execute in the logarithm of the amount of time required on a conventional machine. The implementation (with no redundant exploration) of the path existence problem evolved in this paper is more efficient than any implementation that can be programmed in a dialect of pure LISP that allows the arguments to a function to be evaluated in parallel. This is evidence that applicative programming in languages like pure LISP is apparently less efficient in some practical applications. The efficiency of such applicative languages is important because many computer scientists are proposing to use them on future generation parallel machines whose architectures exploit ultra large scale integration.

I -- A DESCRIPTION SYSTEM

The main goal of the description system is to enable the following kinds of descriptions to be conveniently used:

PARTIAL descriptions are used to express whatever properties of an object happen to be known at particular point in time if they are incomplete. Partial descriptions are important in partial specifications because it is impossible to arrive at complete specifications for a large software system all at once. They are important in proofs because in a proof some properties are given whereas others must be derived.

INCREMENTAL descriptions which enable us to further describe objects when more information becomes available and are a necessary feature for the use of **partial descriptions**. Incremental descriptions are important in proofs and incremental specifications because all of the properties are not available at one time but must be derived and evolved with time.

MULTIPLE descriptions which enable us to ascribe multiple overlapping descriptions to an object which is used for multiple purposes. Multiple descriptions are important in multiple specifications and proofs because different properties of an object might be useful in different contexts.

Our description system is used in stating partial specifications of programs, as a powerful flexible notation to state type declarations, and as a notation to express conditions that are tested during program execution. The assumptions and the constraints on the objects manipulated by a program are an integral part of the program and can be used both as checks when the program is running and as useful information which can be exploited by other systems which examine the program, such as translators, optimizers, indexers, etc. We believe that bugs occurring in programs are frequently caused by the violation of implicit assumptions about the environment in which the program is intended to operate. Therefore many advantages can be drawn by a language that encourages the programmer to state such assumptions explicitly and by a system which is able to detect when they are violated.

Consider a *describe* command of the following form:

```
(describe d
        [preconditions: preconditions_on_d]   ;preconditions are optional
        [is: descriptions_of_d]
        [consequences: constraints_on_descriptions_of_d])   ;constraints are optional
```

The above command says that if x is an object which is described by d such that preconditions_on_d hold, then x is described by *all* the descriptions_of_d and, furthermore, *all* of the constraints_on_descriptions_of_d hold. In other words the command has the same meaning as the following statement:

```
((d such_that preconditions_on_d) is (descriptions_of_d such_that constraints_on_descriptions_of_d))
```

The syntax of the *describe* command was chosen to avoid the nesting of parentheses that can be seen in the above statement.

Our description system is based on the Axiom of Transitivity of Predication which can be stated as follows:

if (<description$_1$> *is* <description$_2$>) *and* (<description$_2$> *is* <description$_3$>)
then (<description$_1$> *is* <description$_3$>)

The importance of the above axiom is that it implies that inheritance holds in our description system.

Our description system is designed to allow us to provide multiple partial descriptions of objects. For example consider the following descriptions:

```
(describe (an Integer)   ;an integer
    [is:
        (a Real)   ;is a real number and
        (⊔ (an Odd) (an Even))])   ;is either an odd integer or an even integer
```

Notice that the *describe* command is assymetric so that it would be *incorrect* to say:

```
(describe (a Real)
    [is: (an Integer)])
```

Furthermore it would also be *incorrect* to say

```
(describe (a Real)
    [is: ¬(an Integer)])
```

since some real numbers are integers.

Our description system successfully deals with an important distinction that has plagued most previous systems which rely on inheritance. Given that π is a Real and that Real is a Real_closed_field, one should not make the mistake of concluding that π is a Real_closed_field. Note that this mistake will not occur in our system because the rule of transitivity of predication does not apply to the following two descriptions:

```
(describe π
    [is: (a Real)])

(describe Real
    [is: (a Real_closed_field)])
```

Note that we have described Real as being a real closed field. Describing an instance such as (a Real) as being a field would be a mistake. Logicians as long ago as Aristotle have known that Cartesian_complex must not be confused with (a Cartesian_complex). However, a good notation was lacking in which to axiomatize the difference.

Below I give more examples of multiple partial descriptions which will prove useful in the first programming example considered in this paper:

```
(describe (a Non_negative_integer) ;a non negative integer
    [is:
        (an Integer) ;is an integer and
        (⊔ 0 (a Positive_integer))]) ;is either 0 or a positive integer

(describe (a Positive_integer) ;a positive integer
    [is:
        (a Non_negative_integer) ;is a non-negative integer
        ¬0]) ;and is not 0
```

Notice that we have just established **interdependency** among our descriptions because we have described Non_negative_integer in terms of Positive_integer and vice versa. Partial descriptions like the ones above are illegal in almost all type systems. The desire to make coreferential descriptions such as these has been one of the driving forces in the evolution of our description system. For example the same kind of coreference shows up in the descriptions of Odd and Even below.

```
(describe (an Odd)  ;an odd integer
    [is:
        (an Integer) ;is an integer and
        ¬(an Even)  ;is not an even integer
        (1 + (an Even))]) ;and is 1 plus an even integer

(describe (an Even) ;an even integer
    [is:
        (an Integer) ;is an integer and
        ¬(an Odd) ;is not an odd integer
        (2 * (an Integer))]) ;and is 2 times an integer
```

The above descriptions enable us to state some relationships between Even and Odd integers. Of course there is more to understand than simply the relationships that have been stated above, but the above descriptions are a start.

It is important to realize that the descriptions given in this paper are not necessarily **definitional**. For example the following descriptions of 0 and 1 do not uniquely characterize them:

```
(describe 0
    [is:
        (a Non_negative_integer)
        (an Even)
        ¬1])
```

Note that multiple descriptions allow us to directly express the fact that 0 is ¬1 as well as that it is an Even number. Thus it is not necessary to continually rederive the former description from the latter.

```
(describe 1
    [is:
        (a Positive_integer)
        (an Odd)
        ¬0])
```

For example 3 satisfies the same description as as the one given above for 1.

```
(describe 3
    [is:
        (a Positive_integer)
        (an Odd)
        ¬0])
```

The description system also needs to be able to describe **attributes** of objects. For example a Triangle has three vertices v_1, v_2, and v_3 as well as three sides. In the description below we use the character "=" to mark the introduction of local identifiers. **Local** identifiers play a role in the description system similar to the role played by **free identifiers** in formulas in the quantificational calculus: they can be bound to any object.

```
(describe (a Triangle)
    [is:
        (a Polygon)
        (a Triangle
            [vertex1: =v1]
            [vertex2: =v2]
            [vertex3: =v3]
            [side1: (a Line_segment [end1: =v1] [end2: =v2])]
            [side2: (a Line_segment [end1: =v2] [end2: =v3])]
            [side3: (a Line_segment [end1: =v3] [end2: =v1])])])
```

where a Line_segment can be described as follows:

```
(describe (a Line_segment)
    [is:
        (a Geometric_figure)
        (a Line_segment
            [end1: (a Point)]
            [end2: (a Point)])])
```

Note that by using the concept Line_segment twice in the above description that we have specified that *every* Line_segment has two attributes end_1 and end_2 which each have as value a Point.

A Point can be described as follows:

```
(describe (a Point)
    [is:
        (a Geometric_figure)
        (a Cartesian_point
            [x_coordinate: =x (a Real)]
            [y_coordinate: =y (a Real)]
            [magnitude: (=x^2 + =y^2)^(1 / 2)])])
```

Thus if

```
(p1 is (a Cartesian_point [x_coordinate: 3] [y_coordinate: 4]))
```

then

```
(p1 is (a Cartesian_point [magnitude: 5]))
```

The above description of a Point does not make any commitments regarding the physical representation of Points. For example the length attribute need not necessarily be stored explicitly all the time but can be computed if and when it is needed. Furthermore the x_coordinate and y_coordinate need not be part of the physical representation of a point because the point might be described in polar form as follows:

```
(describe (a Cartesian_point [x_coordinate: =x] [y_coordinate: =y])
    [is:
        (a Polar_point [magnitude: =r (x^2 + y^2)^(1 / 2)])
        ((a Polar_point [phase: (sin^-1 (y / =r))]) if (=r > 0))])
```

Note that the phase is a **conditional** attribute of a Polar_point; it is guaranteed to be present only if the magnitude is nonzero.

In many cases the attributes of a concept are **conditional**. For example the fact that a Real has an inverse only if it is nonzero can be expressed as follows:

```
(describe ((a Real) which_is =x)
    [preconditions: ¬0]
    [is: (a Real [inverse: =x' (a Real)])]
    [constraint: ((x * x') is 1)])
```

Using the description system, it is easy to describe result of operations. For example the difference of two points can be described as follows:

```
(describe (=p1 - =p2)
    [preconditions:
      (p1 is (a Point [x_coordinate: =x1] [y_coordinate: =y1]))
      (p2 is (a Point [x_coordinate: =x2] [y_coordinate: =y2]))]
    [is: (a Point [x_coordinate: (x1 - x2)] [y_coordinate: (y1 - y2)])])
```

I believe that it is important for a description system to allow information to be presented in an incremental fashion. For example it should be possible for the user to later **further describe** Line_segments as also having another attribute **length** which is the sum of the lengths of the sides:

```
(describe (a Line_segment [end1: =p1] [end2: =p2])
      [preconditions: ((p1 - p2) is (a Point [magnitude: =l]))]
      [is: (a Line_segment [length: l])])
```

Note that the above command is further describing the *entire* description

```
(a Line_segment [end1: =p1] [end2: =p2])
```

II -- FAST FACTORIAL

The first problem that will be investigated is to efficiently compute n!.

I would like to contrast the derivation of an efficient implementation of factorial presented here with the classical method of derivation which requires the invention of a **loop invariant.** The loop invariant method can be explained as follows[1] :

> *We want a program satisfying the specification*
>
> $\{n \geq 0\}$ "Compute n!" $\{y = n!\}$
>
> *To construct this program in a top-down fashion, without using goto's, we begin with the invariant*
>
> $I \equiv y * k! = n! \wedge k \geq 0$
>
> *Since this invariant embodies the essential idea behind the algorithm, one would expect the rest of the program development to be straightforward.*

In the above derivation y is the output variable and k is an internal variable of the program to be constructed. Some advocates of "structured programming" insist that the derivation of an implementation of factorial should *begin* with the invention of the above loop invariant.

It seems to me that requiring the derivation of an implementation to *begin* with the invention of a loop invariant often leads to an obscure and unmotivated derivation. Instead I plan to develop a recursive program for factorial and then make use of a standard transformation into iterative form. In this way I will *derive* the loop invariant as a product of the programming methodology advocated here.

1: Cf. Reynolds 1977

II.1 --- Preliminary Thoughts

To my derivation of an efficient implementation, notice the following elementary facts about factorial.

```
(describe (factorial 0)
    [is: 1])
```

The above fact is very useful in the derivation of an implementation because it enables me to create an implementation which can immediately solve the problem which is posed when (factorial 0) is evaluated.

```
(describe (factorial =n)
    [preconditions: (n is (a Positive_integer))]
    [is: (n * (factorial (n - 1)))])
```

The above fact enables me to see how to construct an implementation to simplify the problem of computing the value of n! for n>0 by solving the problem of computing (n-1)! and then multiplying the answer by n. Since this leads to the sequence of problems of computing n!, (n-1)!, ..., 0! that can all be delegated to the implementation which can always solve the last one.

II.2 --- 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 produces body1)
    ...
    (patternn produces 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 executed 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 the user will be warned demonstrate that the results of executing the bodies are indistinguishable. This rule has the advantage that it

makes body$_i$ more modular since it depends only on pattern$_i$, making it easy to add more selections later. Thus the rule of *concurrent* consideration of cases encourages the construction of programs which are more modifiable. The programs are also more robust since the addition of new cases is less likely to introduce bugs in already existing cases.

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 a large amount of time is required to determine whether or not conditions hold. Thus the rule of concurrent consideration of cases enables some programs to be implemented more efficiently.

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.

II.3 --- An Initial Implementation

Using the above construct, I propose the following implementation:

```
(describe (factorial =n)
   [preconditions: (n is (a Non_negative_integer))]
      ;n is a non-negative integer
   [is:
      (an Integer)] ;the value is an integer
   [implementation:
      (select_case_for n
         (0 produces 1)      ;in case n is 0, the value is 1
         ((> 0) produces (n * (factorial (n - 1)))))])
            ;in case n is not 0, the value is n times (n-1)!
```

We would like to prove that the above implementation satisfies its description. The problem is to show that the definition is an Integer. The proof can be done by induction on n.

Basis:

Suppose (n *is* 0)
Then (factorial n) *is* 1 *is* (*an* Integer)

Induction:

. *Suppose* (factorial (n - 1)) *is* (*an* Integer)
Then (factorial n) *is* (n * (factorial (n - 1))) *is*
((*an* Integer) * (*an* Integer)) *is* (*an* Integer)

II.4 --- An Iterative Implementation

The above implementation is inefficient because it uses extra storage in the recursive invocations of itself. Applying a standard transformation [Standish et. al.: 1976, Burstall and Darlington: 1977, Strong: 1971] for translating recursive procedures into iterative ones[1], I obtain the following implementation:

```
(describe (factorial =n)
 [preconditions: (n is (a Non_negative_integer))]
 [is:(an Integer)]
 [implementation:
  (select_case_for n
      (0 produces 1)      ;in case n is 0, the value is 1
      ((> 0) produces     ;in case n greater than 0
         (factorial_times_accumulation [index: n] [accumulation: 1])
         ;then the value is given by the definition below

         where

         (describe (factorial_times_accumulation
                                [index: =k (a Positive_integer)]
                                [accumulation: =y (an Integer)])
           [is:
              (an Integer)
              (y * (factorial k))
              (factorial n)]
           [implementation:
              (select_case_for k
                 (1 produces y)
                 ((> 1) produces
                    (factorial_times_accumulation [index: (k - 1)] [accumulation: (y * i)])))])))])
```

1: Actually the implementation given below is tail recursive. Recall, however, that in actor semantics tail recursive programs *are* iterative [Hewitt and Smith: 1975, Hewitt: 1977]

Note that I have *derived* the somewhat obscure loop invariant referred to earlier, namely:

```
y * k! = (factorial_times_accumulation [index: k] [accumulation: y]) = (factorial n) = n!
```

However the iterative implementation given above is still too slow for my purposes! The faster implementation of factorial which I develop below does not rely on the above loop invariant!

II.5 --- Greater Speed

The cause of the slowness of the iterative implementation lies in the fact that the multiplications are performed sequentially. They take the form

```
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 **recursive** implementation:

```
(describe (factorial =n)
   [preconditions: (n is (a Positive_integer))]
   [implementation: (subfactorial n 1)])

(describe (subfactorial =k =r)
   [preconditions:
      ({k r} each_is (a Positive_integer))
      (k ≥ r)]
   [is: ((factorial k) ÷ (factorial r))]
   [implementation:
      (select_case_for k
         (r produces 1)
         ((r + 1) produces k)
         ((> (r + 1)) produces
                  (*
                     (subfactorial k ((k + r) ÷ 2))
                     (subfactorial ((k + r) ÷ 2) r))))])
```

```
(describe (=n ÷ =m)
    [preconditions:
        (n is (a Positive_integer))
        (m is (a Non_negative_integer))]
    [is: (an Integer) =d]
    [constraints:
            (0 ≤ d < m)
            ((m * d) ≤ n)
            (n < (m * (d + 1)))])
```

The above implementation of factorial has great potential for parallelism. I would like to make this potential manifest in the structure of the code I will **annotate** it with pragmatic information that can be used in the execution of the code. An expression of the form (*create_future* E) will be used to indicate that the execution of E can be performed by a separate concurrent activity [Baker and Hewitt: 1977, Hibbard: 1977, Friedman and Wise: 1978]. The annotated program appears below:

```
(describe (subfactorial =k =r)
  [preconditions:
      ({k r} each_is (a Positive_integer))
      (k ≥ r)]
  [is: ((factorial k) ÷ (factorial r))]
  [implementation:
      (select_case_for k
          (r produces 1)
          ((r + 1) produces k)
          ((> (r + 1)) produces
                  (*
                      (create_future (subfactorial k ((k + r) ÷ 2)))
                      (create_future (subfactorial ((k + r) ÷ 2) r))))))])
```

The above implementation executes in the logarithm of the time of the iterative implementation on an actor machine [Hewitt: 1979] with a large number of processors connected by a high bandwidth network. Note that in contrast with the usual practice in program transformation systems that we have evolved a more efficient *recursive* implementation from an *iterative* implementation!

II.6 --- Retrospective

Before going on to the next example, I would like to highlight some aspects of evolutionary programming which I am advocating. I did not attempt to develop the most efficient program all at once. Instead important aspects of the background of the problem to be solved were described. Then I gradually worked to evolve the implementation towards greater speed. Notice that the programming did not proceed in a strictly top down fashion using stepwise refinement to give definitions of modules which have been introduced but not yet defined. Although important ideas are carried forward from one implementation to the next, the code itself underwent major perturbations with each refinement.

The initial recursive implementation used too much space and time so an iterative implementation was evolved using a standard transformation. The iterative implementation was an improvement but it was still too slow. However by studying the iterative implementation, I got an idea how to improve the speed by using parallelism. In this way each implementation *evolved* into its successor.

Note that the last implementation of factorial can be implemented in a dialect pure LISP that allows the arguments of a function to be evaluated in parallel.

III -- FAST PATH EXISTENCE DETERMINATION

The next problem is to write a procedure which will determine whether or not a path (of length zero or greater) exists in a directed graph from some member of a set [without duplicates] of nodes X to a member of a set of nodes Y.

The problem is important because it often arises in problem solving situations. For example one common method to prove a formula of the form (implies X Y) is to assume that X holds and attempt to prove Y. In such cases it is desirable to be able to draw conclusions by chaining forward from X while at the same time attempting to prove Y by chaining backward [Hewitt: IJCAI-75].

III.1 --- Describing Finite Graphs

We have the following description of a Node_of_finite_graph:

```
(describe ((a Node_of_finite_graph
                [immediate_successors: =the_immediate_successors ]
                [immediate_predecessors: =the_immediate_predecessors])
           which_is =this_node)
 [preconditions:
   ({the_immediate_successors the_immediate_predecessors} each_is (a Node_set_of_finite_graph))]
 [consequences:
      (implies (=n ∈ the_immediate_successors)
               (this_node ∈ (immediate_predecessors {=n})))
      (implies (=n ∈ the_immediate_predecessors)
               (this_node ∈ (immediate_successors {=n})))])
```

The above description says that every node of a finite graph is an immediate successor of each of its immediate predecessors and conversely every node is an immediate predecessor of each of its immediate successors.

The usual mathematical properties of predecessor and successor functions are described as follows:

$$\text{(immediate\_successors Z)} \equiv \bigcup_{n \in Z} \text{(immediate\_successors \{n\})}$$

$$\text{(successors W)} \equiv \bigcup_{i \in N} \text{(immediate\_successors}^i \text{ W)}$$

$$\text{(immediate\_predecessors Z)} \equiv \bigcup_{n \in Z} \text{(immediate\_predecessors \{n\})}$$

$$\text{(predecessors W)} \equiv \bigcup_{i \in N} \text{(immediate\_predecessors}^i \text{ W)}$$

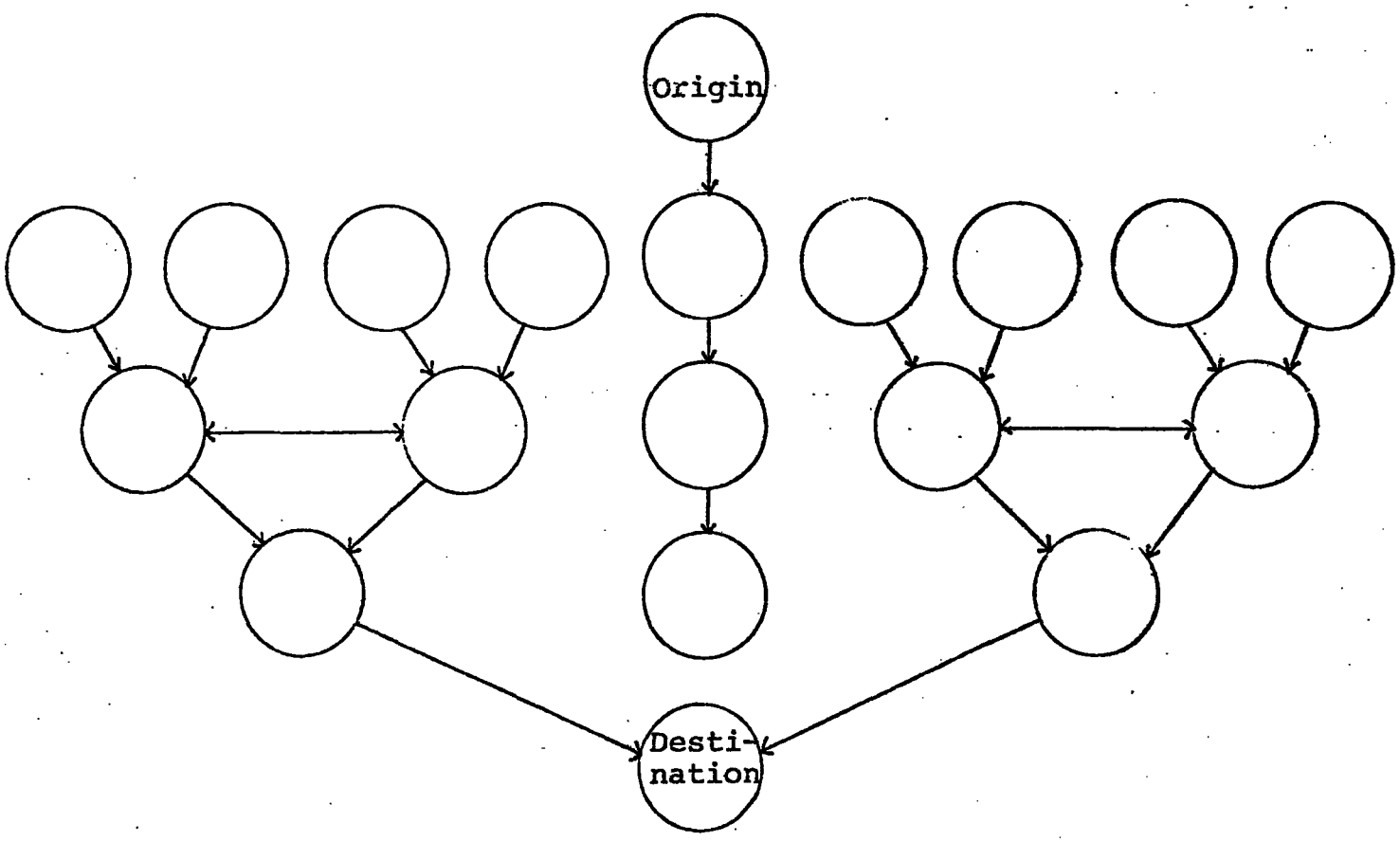
where **N** is the set of non-negative integers and for any function f the notation f^i denotes the i-fold composition of f with itself. I.e. for any function f, f^0 is the identity function and for any non-negative integer i, $(f^{i+1}\ x)$ is $(f\ (f^i\ x))$.

The above notation for composition of a function with itself is easily described as follows:

```
(describe (=f^0 =x)
    [preconditions: (f is (a Mapping))]
    [is: x])

(describe (=f^=n =x)
    [preconditions:
        (f is (a Mapping))
        (n is (a Positive_integer))]
    [is: (f (f^(n - 1) x))])
```

The usual mathematical properties of predecessors and successors on graphs given above can now be expressed using the following descriptions:

```
(describe (immediate_successors (a Node_set_of_finite_graph [immediate_successors: =s]))
    [is: s])

(describe (immediate_predecessors (a Node_set_of_finite_graph [immediate_predecessors: =p]))
    [is: p])

(describe ((a Node_set_of_finite_graph
                [immediate_successors: =the_immediate_successors]
                [successors: =the_successors]
                [immediate_predecessors: =the_immediate_predecessors]
                [predecessors: =the_predecessors])
        which_is =this_set)
    [preconditions:
        ({the_immediate_successors the_successors} each_is (a Node_set_of_finite_graph))
        ({the_immediate_predecessors the_predecessors} each_is (a Node_set_of_finite_graph))]
    [is: (a Finite_set)]
    [consequences:
        (the_immediate_successors is (immediate_successors this_set))
        (the_successors is (successors this_set))
        (the_immediate_predecessors is (immediate_predecessors this_set))
        (the_predecessors is (predecessors this_set))])
```



A Graph which Demonstrates that Exploring Successors of Origin is Sometimes More Efficient for Showing that no Path Exists from Origin to Destination.

III.2 --- A Partial Specification

Below, implementations will be defined such that invoking an expression of the form

```
(path_exists [from: X] [to: Y])
```

returns either true or false depending on whether or not there is a path from the node set X to the node set Y of length zero or greater. The problem to be solved can be partially specified as follows:

```
(describe (path_exists [from: =X (a Node_set_of_finite_graph [successors: =s])]
                       [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
    [is: (a Boolean)
        (false if_and_only_if ((s ∩ p) is {}))])
```

The only primitive procedures which are available for use in the implementation are immediate_successors and immediate_predecessors which execute in a time which is proportional to the size of the node set which is the argument.

```
(describe (time (a Transaction [procedure: immediate_successors] [argument: =X]))
  [is: (log (size X))])

(describe (time (a Transaction [procedure: immediate_predecessors] [argument: =X]))
  [is: (log (size X))])
```

In our application the finite graph is not physically stored before the execution of path_exists begins. Instead the graph is generated dynamically by the procedures immediate_successors and immediate_predecessors. Consequently we cannot speed up the process by somehow "marking" the nodes.

III.3 --- Some Initial Implementations

One obvious implementation is obtained by chaining forward from X:

```
(describe (forward_path_exists [from: =X (a Node_set_of_finite_graph [successors: =s])]
                               [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
   [is: (a Boolean)
        (false if_and_only_if ((s ∩ p) is {}))]
   [implementation:
      (select_case_for (X ∩ Y)
         (¬{} produces true)
         ({} produces
            (select_case_for ((immediate_successors X) - X)
               [{} produces false]
               [¬{} produces
                  (forward_path_exists
                     [from: ((immediate_successors X) ∪ X)]
                     [to: Y])])))])
```

Another obvious implementation can be obtained by chaining backward from Y:

```
(describe (backward_path_exists [from: =X (a Node_set_of_finite_graph [successors: =s])]
                                [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
   [is: (a Boolean)
        (false if_and_only_if ((s ∩ p) is {}))]
   [implementation:
      (select_case_for (X ∩ Y)
         (¬{} produces true)
         ({} produces
            (select_case_for ((immediate_predecessors Y) - Y)
               ({} produces false)
               (¬{} produces
                  (backward_path_exits
                     [from: X]
                     [to: ((immediate_predecessors Y) ∪ Y)])))))])
```

III.4 --- Concurrently Working Forward and Backward

Depending on the topology of the graph either chaining forward or chaining backward can be more efficient. For example if *Y* has no predecessors or X has a larger fan out then chaining backward is preferable A new implementation can be constructed which executes in parallel the implementation which chains forward and the implementation which chains backward. In order to do this we will make use of the *finishes_first* construct (called *amb* in [McCarthy: 1963] and *either* in [Ward: 1974]). The value of an expression of the form (*finishes_first* E_1 E_2) is computed by evaluating E_1 and E_2 in parallel and selecting the value of whichever one finishes first. When one of these two computations produces a value, work on the other one is terminated.

```
(describe (path_exists
            [from: =X (a Node_set_of_finite_graph [successors: =s])]
            [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
  [is: (a Boolean)
    (false if_and_only_if ((s ∩ p) is {}))]
  [implementation:
    (finishes_first
      (forward_path_exists [from: X] [to: Y])
      (backward_path_exists [from: X] [to: Y]))])
```

However, even the above implementation is not efficient enough for my application. A major problem is that forward_path_exists and backward_path_exists may duplicate the work of activities exploring predecessors.

We shall attempt to deal with this problem by merging the results of exploring forward and backward in parallel. More efficient implementations of the mappings successors and predecessors are needed. The following implementation is easily developed for successors:

```
(describe (successors =X)
   [preconditions: (X is (a Node_set_finite_graph [successors: =S]))]
   [is: S]
   [implementation: ((subsuccessors [from: X] [explored: {}])

     where

     (describe (subsuccessors [from: =Z (⊆ S)] [explored: =E (⊆ S)])
        [is: S]
        [implementation:
           (select_case_for Z
                ({} produces {})
                ({an_element_Z ...} produces
                  (select_case_for an_element_Z
                    ((∈ E) produces {})
                    (¬(∈ E) produces
                      (∪
                          {an_element_Z}
                          (create_future
                            (subsuccessors
                              [from: (Z - {an_element_Z})]
                              [explored: (E ∪ {an_element_Z})]))
                          (create_future
                            (subsuccessors
                              [from: (immediate_successors {an_element_Z})]
                              [explored: (E ∪ {an_element_Z})])))))))))])
```

The idea of the above implementation is that an activity should not duplicate the work of exploring any node which it has already seen. This can be accomplished by maintaining a record of the nodes that it has already explored. Unfortunately it is still possible for activities exploring successors to duplicate each others work.

III.5 --- 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 executed. If more than one of the $condition_i$ hold then an arbitrary one of the corresponding $body_i$ is selected to be executed. The user will be warned if more than one of the $condition_i$ can hold simultaneously and the execution of the corresponding $body_i$ do not 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 different from the conditionals of McCarthy, Dijkstra, etc. in several important respects. The conditions of *select_one_of* have been generalized to allow pattern matching as in the pattern directed programming languages PLANNER, QA-4, POPLER, CONNIVER, etc. Notice that our **concurrent** conditional expression is different from the usual **nondeterministic** conditional in that if *any* of the conditions hold then the body of one of them *must* be selected for execution even if the evaluation of some other condition does not terminate (cf. [Manna and McCarthy: 1970, Paterson and Hewitt: 1971, Friedman and Wise: 1978]).

III.6 --- Greater Concurrency

The following implementation has even greater concurrency:

```
(describe (path_exists
            [from: =X (a Node_set_of_finite_graph [successors: =s])]
            [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
  [is: (a Boolean)
      (false if_and_only_if ((s ∩ p) is {}))]
  [implementation:
      (subpath_exists
          [from: (create_future (successors X))]
             ;begin computing the successors of X in parallel
          [to: (create_future (predecessors Y))])])
             ;also begin computing the successors of Y in parallel

(describe (subpath_exists [from: =s] [to: =p])
  [implementation:
     (select_one_of
        (if (s is {}) then false)
        (if (p is {}) then false)
        (if (s is {=an_element_s ...}) then
          ;select an element of s even though all the elements have not yet been computed
            (or
              (create_future (an_element_s ∈ p))
              (create_future
                (subpath_exists
                   [from: (s - {an_element_s})]
                   [to: p])))))
        (if (p is {an_element_p ...}) then
            (or
              (create_future (an_element_p ∈ s))
              (create_future
                (subpath_exists
                   [from: s]
                   [to: (p - {an_element_p})])))))])
```

Each invocation of subpath_exists either selects a node from the successors and checks to see if it is in the predecessors or *vice versa*. In this case the concurrent conditional expression makes a choice that is determined by the concurrency of the system.

III.7 --- Prohibition of Redundant Exploration

Although the latest implementation has a great deal of concurrency, it is quite inefficient in its use of storage. Much of the inefficiency stems from the fact that it can repeatedly explore the same node. In order to deal with this problem, I will impose the restriction that **redundant exploration** of the same node is prohibited. More precisely I would like to prohibit the ability to take the immediate_predecessors or immediate_successors of the same node more than once. In many problem solving systems prohibiting redundant exploration is important because each node requires a large amount of storage.

In order to avoid redundant exploration, the implementation is evolved so that a level of successors of the origins is explored in strict alternation with a level of predecessors of the destinations.

```
(describe (path_exists
            [from: =X (a Node_set_of_finite_graph [successors: =s])]
            [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
  [is: (a Boolean)
      (false if_and_only_if ((s ∩ p) is {}))]
  [implementation: (forward_step [from: X] [to: Y])])

(describe (forward_step
            [from: =X (a Node_set_of_finite_graph [successors: =s])]
            [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
  [is: (a Boolean)
      (false if_and_only_if ((s ∩ p) is {}))]
  [implementation:
    (select_case_for (X ∩ Y)
      (¬{} produces true)
      ({} produces
        (select_case_for ((immediate_successors X) - X)
          [{} produces false]
          [¬{} produces
            (backward_step
              [from: ((immediate_successors X) ∪ X)]
              [to: Y])])))])
```

```
(describe (backward_step
            [from: =X (a Node_set_of_finite_graph [successors: =s])]
            [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
  [is: (a Boolean)
     (false if_and_only_if ((s ∩ p) is {}))]
  [implementation:
    (select_case_for (X ∩ Y)
       (¬{} produces true)
       ({} produces
          (select_case_for ((immediate_predecessors Y) - Y)
             ({} produces false)
             (¬{} produces
                (forward_step
                   [from: X]
                   [to: ((immediate_predecessors Y) ∪ Y)])))))])
```

The above implementation is still to slow for my purposes. Therefore I will try a somewhat different approach.

III.8 --- Message Passing and Serializers

My next idea for a more efficient implementation, is to use a shared procedural data structure called a **map** that records how much of the graph has been explored. It will be used to prevent redundant exploration of the graph. An expression of the form

```
(create_serialized_map [origins: X] [destinations: Y])
```

will create an actor which accepts messages of the form `(a Predecessor [node: n])` which says that the node `n` is a predecessor of the destination nodes `Y` and messages of the form `(a Successor [node: n])` which says that node `n` is a successor of the origin nodes `X`.

```
(describe (a Predecessor [node: (a Node_of_finite_graph)])
  [is: (a Message)])

(describe (a Successor [node: (a Node_of_finite_graph)])
  [is: (a Message)])
```

III.9 --- PRIMITIVE SERIALIZERS

The syntax of a simple primitive serializer in Act1 is:

(create_serialized_actor B)

Primitive serializers are used to create actors whose behavior may change after the receipt of a communication. A convenient way to express this is by means of the notion of behavior. At any given time a serialized actor has a behavior (which is another actor) and its behavior may change as a result of communications which it receives. The initial behavior of a serialized actor created by *create_serialized_actor* is the value of the expression B. The initial behavior of an actor created by *create_serialized_actor* is the value of the expression B.

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 message arrives, the serializer becomes *locked* and the message is sent to B. The value produced by B is installed as the next behavior of the serialized actor. The serialized actor then becomes **unlocked** and thus able to **accept** the next message.

A behavior will typically be implemented using a *create_unserialized_actor* expression which has the following syntax:

(create_unserialized_actor
(pattern_for_message$_1$ *produces* body$_1$)
...
(pattern_for_message$_j$ *produces* body$_j$))

If an actor created by a *create_unserialized_actor* expression **accepts** a message M which matches any of the pattern_for_message$_i$, then the corresponding body$_i$ is executed to produced the next behavior. If M matches more than one of the pattern_for_message$_i$, then an arbitrary one of the corresponding body$_i$ is selected to be executed.

As a result of **accepting** the message, the current behavior computes a reply RP and a new behavior NB. The reply is sent and the new behavior installed by using a command of the form

(unlock [reply: RP] [become: NB])

An important consideration in the design of efficient serializers is that they should remain locked for as brief a time as possible.

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

(*a* Transmission [target: T] [message: M])

Next it must **arrive** in a arrival event of the form

(*an* Arrival [target: T] [message: M])

Hardware modules called **arbiters** are used to establish an arrival ordering for all messages sent to T. Finally it must be **accepted** in an acceptance event of the form

(*an* Acceptance [recipient: T] [message: M])

Messages are accepted in the order in which they arrive. The acceptance 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 accepted until it unlocks.

III.10 --- Implementing a Map

The procedure create_serialized_map defined below creates a **map** abstraction that records how much of the graph has been examined. Evaluating an expression of the form (create_serialized_map [origins: X] [destinations: Y]) will produce an actor which records nodes explored from the origin node set X to the destination node set Y.

```
(describe (create_serialized_map [origins: =the_origins] [destinations: =the_destinations])
 [preconditions:
    ({the_origins the_destinations} each_is (a Node_set_of_finite_graph))
    ((the_origins ∩ the_destinations) is {})]
    ;to create a map to record the nodes explored from the origins to the destinations
 [is: (a Serialized_actor [responds_to: (□ (a Predecessor) (a Successor))])]
 [implementation:
   (create_serialized_actor
     (a Map [successors: the_origins] [predecessors: the_destinations]))])
```

```
(describe (a Map
              [successors: (a Node_set_of_finite_graph)      (⊇ the_origins)]
              [predecessors: (a Node_set_of_finite_graph) (⊇ the_destinations)])
   [preconditions: ((predecessors ∩ successors) is {})]
   [implementation:
    (create_unserialized_actor
     ((a Predecessor [node: =n]) produces
        ;the message received says that the node n is a predecessor of destinations
        (select_one_of
          (if (n ∈ predecessors) then
            ;if n is already known to be an element of the final nodes
              (unlock [reply: (a Known_predecessor)]))
               ;then this is reported and the serializer is unlocked
          (if (n ∈ successors) then
              (unlock [reply: (a Known_successor)]))
          [none_of_the_above:
              (unlock [reply: (an Unexplored_node)]
                 ;the fact that it is not already present in the map is reported and
                   [become: (a Map [predecessors: (predecessors ∪ {n})])])])
                ;the serializer is unlocked with the node n added to predecessors
     ((a Successor [node: =n]) produces
       (select_one_of
          (if (n ∈ successors) then
              (unlock [reply: (a Known_successor)]))
          (if (n ∈ predecessors) then
              (unlock [reply: (a Known_predecessor)]))
          [none_of_the_above:
              (unlock
                  [reply: (an Unexplored_node)]
                  [become: (a Map [successors: (successors ∪ {n})])])])])])
```

III.11 --- Exploring the Graph using the Map

I will use expressions of the form (*send_to* t m) to denote the result of sending the message m to the actor t.

```
(describe (path_exists
            [from: =X (a Node_set_of_finite_graph [successors: =s])]
            [to: =Y (a Node_set_of_finite_graph [predecessors: =p])])
 [is: (a Boolean) (false if_and_only_if ((s ∩ p) is {}))]
 [implementation:
  (select_case_for (X ∩ Y)
    (¬{} produces true)  ;if X intersect Y is nonempty then the value is true
    ({} produces   ;else if they have no nodes in common then
       (let (the_map be (create_serialized_map [origins: X] [destinations: Y]))
            ;create a new map called the_map to explore the territory from X to Y
        then
          (finishes_first
             (explore_successors (immediate_successors X))
                ;determine whether or not the immediate successors of X reach Y
             (explore_predecessors (immediate_predecessors Y)))
        where ;the following definitions are placed here in order to share the_map
          (describe (explore_successors =S) ;to explore the successors of a node set S
            [implementation:
              (select_case_for S
                ({} produces false)      ;in case S is the empty set, the value is false
                ({=an_element_S ...} produces ;select an element of S and call it an_element_S
                   (or
                      (create_future (explore_successors (S - {an_element_S})))
                        ;determine whether or not the successors of REST_S reach Y
                      (create_future
                        (select_case_for (send_to the_map (a Successor [node: an_element_S]))
                           ;the cases for the reply from the_map as a result of
                           ;telling it that an_element_S is a successor node are
                           ((a Known_successor) produces false)
                             ;return false if an_element_S is already present as a successor of X
                           ((a Known_predecessor) produces true)
                             ;return true if an_element_S is known to be a predecessor of Y
                           ((an Unexplored_node) produces
                              ;in case an_element_S is unexplored in the map m
                             (explore_successors (immediate_successors {an_element_S})))))))))])
                               ;the value is determined by whether or not its successors reach Y
```

```
(describe (explore_predecessors =S)
 [implementation:
   (select_case_for S
     ({} produces false)
     ({=an_element_S ...} produces
        (or
           (create_future (explore_predecessors (S - {an_element_S})))
           (create_future
            (select_case_for (send_to the_map (a Predecessor [node: an_element_S]))
              ((a Known_predecessor) produces false)
              ((a Known_successor) produces true)
              ((an Unexplored_node) produces
                 (explore_predecessors (immediate_predecessors {an_element_S})))))))))])
```

III.12 --- Removing the Bottleneck

The map actor m created in the above implementation may be a serious bottleneck in a multiprocessor system with a large number of processors. The reason is that a rapidly growing number of activities must be serialized through the map. The implementation of the **map** abstraction given below eliminates this bottleneck.

The bottleneck is removed by the ability to **replace** a serialized actor with an unserialized actor. If a result of the form

```
(unlock
   [reply: v]
   [become: r])
```

is computed then the serializer replies with the value of the expression v and is replaced by the actor which is the value of r. Our use of replacement is similar to the use of a similar operator in [Kahn and Macqueen: 1977].

The idea of the implementation below is that when the **successors** or **predecessors** of a serialized map **m** reach a certain size called the **break_size**, then **m** can become an unserialized map **u**. The possible future growth of **u** can be accommodated by giving it a left submap and right submap selected on the basis of node number for storing any new nodes which it receives. Node numbers can be described as follows:

```
(describe ((a Node) which_is =this_node)
   [is: (a Node [number: =n (an Integer)])]
   [constraints:
     (implies
          (=m is (a Node [number: n]))
          (m is this_node))])

(describe number
   [is: (a Projective_relation [concept: Node])])
```

Projective relations are described in an appendix. The upshot is that two nodes are coreferential if and only if they have the same node number.

The left and right submaps are created as serialized actors but will also become unserialized if they exceed the break_size. In this way the potential bottleneck is eliminated.

```
(describe (create_serialized_map [origins: =the_origins] [destinations: =the_destinations])
  [preconditions:
       ({the_origins the_destinations} each_is (a Node_set_of_finite_graph))
       ((the_origins ∩ the_destinations) is {})]
  [is: (an Actor [responds_to: (□ (a Predecessor) (a Successor))])]
  [implementation:
    (create_serialized_actor
            (a Serialized_map [successors: the_origins] [predecessors: the_destinations]))])
```

```
(describe (a Serialized_map
                [successors: (a Node_set_of_finite_graph) (⊇ the_origins)]
                [predecessors: (a Node_set_of_finite_graph) (⊇ the_destinations)])
  [preconditions: ((predecessors ∩ successors) is {})]
  [implementation:
    (create_unserialized_actor
      ((a Predecessor [node: =n]) produces
        (select_one_of
          (if (n ∈ predecessors) then
              (unlock [reply: (a Known_predecessor)]))
          (if (n ∈ successors) then
              (unlock [reply: (a Known_successor)]))
          [none_of_the_above:
              (select_case_for (size predecessors)
                ((< break_size) produces
                  (unlock
                    [reply: (an Unexplored_node)]
                    [become: (a Serialized_map [predecessors: (predecessors ∪ {n})])]))
                ((≥ break_size) produces
                  (unlock
                    [reply: (an Unexplored_node)]
                    [become: (create_unserialized_map (predecessors ∪ {n}) successors)])))])))
      ((a Successor [node: =n]) produces
        (select_one_of
          (if (n ∈ successors) then
              (unlock [reply: (a Known_successor)]))
          (if (n ∈ predecessors) then
              (unlock [reply: (a Known_predecessor)]))
          [none_of_the_above:
              (select_case_for (size successors)
                ((< break_size) produces
                  (unlock
                    [reply: (an Unexplored_node)]
                    [become: (a Serialized_map [successors: (successors ∪ {n})])]))
                ((≥ break_size) produces
                  (unlock
                    [reply: (an Unexplored_node)]
                    [become: (create_unserialized_map predecessors (successors ∪ {n}))])))])))])
```

```
(describe (create_unserialized_map =successors =predecessors)
 [preconditions:
     ({successors predecessors} each_is (a Node_set_of_finite_graph))
     ((successors ∩ predecessors) is {})]
 [is: (an Unserialized_actor [responds_to: (⊔ (a Predecessor) (a Successor))])]
 [implementation:
  (let
     (median_node_number be (median (node_numbers (predecessors ∪ successors))))
     (left_submap be (create_serialized_map [origins: {}] [destinations: {}]))
     (right_submap be (create_serialized_map [origins: {}] [destinations: {}]))
   then
    (create_unserialized_actor
      ((a Predecessor [node: =n (a Node [number: =number_of_n])]) =the_message produces
         ;the message received says that the node n is a predecessor of destinations
         (select_one_of
           (if (n ∈ predecessors) then
             ;if n is already known to be an element of the final nodes
                (a Known_predecessor))
                 ;then this is reported
           (if (n ∈ successors) then
                (a Known_successor))
           [none_of_the_above:
                (select_case_for number_of_n
                  ((≤ median_node_number) produces (send_to left_sub_map the_message))
                  ((> median_node_number) produces (send_to right_sub_map the_message)))]))
      ((a Successor [node: =n (a Node [number: =number_of_n])]) =the_message produces
        (select_one_of
           (if (n ∈ successors) then
                (a Known_successor))
           (if (n ∈ predecessors) then
                (a Known_predecessor))
           [none_of_the_above:
                (select_case_for number_of_n
                  ((≤ median_node_number) produces (send_to left_sub_map the_message))
                  ((> median_node_number) produces (send_to right_sub_map the_message)))])))))])
```

On an actor machine [Hewitt: 1979] with sufficiently many processors, the above implementation executes in the logarithm of the amount of time of the implementation given by [Reynolds: 1977] for a conventional serial computer.

The implementation (with no redundant exploration) of the path existence problem evolved in this paper is more efficient than any implementation that can be programmed in an applicative language such as a dialect of pure LISP that allows the arguments to a

function to be evaluated in parallel. The reason for the inefficiency is that in such an applicative language it is necessary to reconstruct the map for each level of depth of the graph explored in order to avoid redundant exploration.

This result is important because it bears on the question whether all programming can be done efficiently in an applicative language like pure LISP.

IV -- CONCLUSIONS

In this paper I have evolved concurrent implementations for two simple programming problems. It is interesting to contrast the approache taken here with the one in [Naur: 1972]. The methodolody advocated in this paper can produce significantly faster implementations for an actor machine [Hewitt: 1979] than the methods used in [Reynolds: 1977]. Also the development of these implementations has caused me to evolve my criteria of what I am willing to accept as a useful implementation for the two problems considered in this paper: *future implementations must be at least as efficient as the implementations given here.* The derivations in this paper illustrate the co-evolution of implementations and partial interface specifications which often occurs in more substantial software projects. Only at a very late stage did I realize that (factorial n) could be implemented in the time of (log n) multiplications. *Explorations of what is possible to implement provide guidance on what are reasonable partial interface specifications.* As experience accumulates in using an implementation, more of the real needs and possible benefits are discovered causing the partial interface specifications to change with time.

V -- ACKNOWLEDGEMENTS

This paper is a somewhat belated and much revised version of talks which I delivered at the Electrotechnical Laboratory in June 1976. In addition to absence of the "goto", the *individual* assignment command is also entirely absent from this paper.

Valdis Berzins, Dick Waters, and Kenneth Kahn suggested some changes in emphasis that helped to make this paper less contentious. Giuseppe Attardi, Jerry Barber, Henry Lieberman, Bill Martin, and Luc Steels made many useful suggestions for improvements in the form and content of this paper. The description system represents joint work with Attardi. The comments and criticisms of Bill Ackerman, Russ Atkinson, Jerry Barber, Roger Duffey, Barbara Liskov, Henry Lieberman, Hideyuki Nakashima, Jerry Roylance, Steve Smoliar, Guy Steele, and the referees have materially improved the presentation and content of a previous version of this paper. Conversations with John Reynolds, Robin Milner, and Rod Burstall at an Edinburgh outing in July 1977 provided the impetus for the development of the ideas in this paper. The path existence problem treated in this paper was proposed as an interesting one to investigate by [Reynolds: 1977]. I would like to thank Professor Reynolds for providing me with an extremely interesting problem to investigate in terms of the programming methodology advocated here.

This paper builds on previously published work on the relationship between iterative and recursive control structure [Greif and Hewitt: 1975 and Hewitt: 1977]. Since the first version of this paper was completed, Manna and Waldinger have written a memo entitled "Structured Programming with Recursion" which supplements some of the points about the relationship between recursive and iterative control structure.

Our description system is a synthesis of the mechanisms in logic and type systems. It unifies and mutually strengthens the best aspects of both. The powerful logical rules of deduction can be applied to descriptions of statements. Predication between descriptions provides a **hierarchy** in which knowledge can be organized in the fashion of Roget's Thesaurus and the Micropaedia of the Encyclopaedia Britannica.

The intellectual roots of our description system go back to von Neumann-Bernays-Godel set theory [Godel: 1940], the ω-order quantificational calculus, and the lambda calculus. Its development has been influenced by the property lists of LISP, the pattern matching constructs in PLANNER-71 and its descendants QA-4, POPLER, CONNIVER, etc., the multiple descriptions and beta structures of MERLIN, the class mechanism of SIMULA, the frame theory of Minsky, the packagers of PLASMA, the

stereotypes in [Hewitt: 1975], the tangled hierarchies of NETL, the attribute grammars of Knuth, the type system of CLU, the descriptive mechanisms of KRL-0, the partitioned semantic networks of [Fikes and Hendrix: 1977], the conceptual representations of [Yonezawa: 1977], the class mechanism of SMALLTALK [Ingalls: 1978], the goblets of Knowledge Representation Semantics [Smith: 1978], the selector notation of BETA, the inheritance mechanism of OWL, the mathematical semantics of actors [Hewitt and Attardi: 1978], the type system in Edinburgh LCF, the XPRT system of Luc Steels, the constraints in [Borning: 1977, 1979 and Steele and Sussman: 1978]. Conversations with Alan Borning, Scott Fahlman, William Martin, Allen Newell, Alan Perlis, Dana Scott, Brian Smith, and the participants in the "Message Passing Systems" seminar were extremely helpful in getting the description system nailed down.

PLASMA adopted the ideas of pattern matching, message passing, and concurrency as the core of the language. It was developed in an attempt to synthesize a unified system that combined the message passing, pattern matching, and pattern directed invocation and retrieval in PLANNER [Hewitt: 1969; Sussman, Charniak, and Winograd: 1972; Hewitt: 1971], the modularity of SIMULA [Dahl and Nygaard: 1968], 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 **subclass** concept originated in [Dahl and Nygaard: 1968] and adapted in [Ingalls: 1978] has provided useful ideas.

The pattern matching implemented in PLASMA was developed partly to provide a convenient efficient method for an actor implemented in the language to bind the components of a message which it receives. This decision was based on experience using message passing for pattern directed invocation which originated in PLANNER [Hewitt: IJCAI-69] (implemented as MICRO-PLANNER by [Charniak, Sussman, and Winograd: 1971]). A related kind of simple pattern matching has also be used to select the components of messages by [Ingalls: 1978] in one of the later versions of SMALLTALK and by [Hoare: 1978] in a design for Communicating Sequential Processes. However CSP uses *assignment* to pattern variables instead of *binding* which is used in PLANNER, SIMULA, and PLASMA.

SCHEME [Steele and Sussman: 1978] is a simpler language which does not deal with either pattern matching or concurrency. The attempt by Sussman and Steele to relate PLASMA and the actor message passing model of computation to the lambda calculus

influenced them to abandon dynamic [fluid] scoping in favor of lexical scoping so that their language could also use the paradigm for iteration developed for PLASMA. Lexical scoping was chosen for PLASMA because it much more closely mirrors the underlying message passing semantics of actors than dynamic scoping. Modeling dynamic scoping would require that a dictionary be passing with every request. The description system discussed in this paper also relies very heavily on lexical scoping. Guy Steele [Steele 1977] has developed a compiler for SCHEME. Henry Lieberman is developing a more general compiler that deals with pattern matching, concurrency, and synchronization entirely within the message passing paradigm.

VI -- BIBLIOGRAPHY

Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" CACM. August 1978. pp. 613-641.

Birtwistle, G. M.; Dahl, O.; Myhrhaug, B.; and Nygaard, K. "SIMULA Begin" Auerbach. 1973.

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

Bobrow, D. G. and Winograd, T. "An Overview of KRL-0 , a Knowledge Representation Language" Cognitive Science Vol. 1 No. 1. 1977.

Borning, A. "ThingLab -- An Object-Oriented System for Building Simulations Using Constraints" Proceedings of IJCAI-77. August, 1977.

Bourbaki, N. "Theory of Sets" Book I of Elements of Mathematics. Addison-Wesley. 1968.

Burstall, R. M. and Darlington, J. "A Transformation System for Developing Recursive Programs" JACM. Vol. 24, No. 1. Jan. 1977. pp. 44-67.

Burton, R. and Brown J. S. "A Tutoring and Student Modeling Paradigm for Gaming Environments" SIGCSE Bulletin. Vol. 8 No. 1. February 1976. pp. 236-246.

Cheatham, T. E.; Holloway, G. H.; and Townley, J. A. "Symbolic Evaluation and the Analysis of Programs" Aiken Computation Laboratory. Harvard University. TR-19-78. November 1978.

Clarke, L. "A System to Generate Test Data and Symbolically Execute Programs" IEEE TSE-2 No. 3. Sept. 1976. pp 215-222.

Dahl, O. J. and Nygaard, K. "Class and Subclass Declarations" In Simulation Programming Languages J. N. Buxton (Ed.) North Holland. 1968. pp. 158-174.

Darlington, J. and Feather, M. "A Transformational Approach to Modification" February 1979.

Davies, D. J. "POPLER 1.5 Reference Manual" Theoretical Psychology Unit. University of Edinburgh. Report No. 1. May, 1973.

Dewar, R.; Grand, A.; Liu, S.; Schonberg, E.; and Schwartz, J. "Programming by Refinement, as Exemplified by the SETL Representation Sublanguage"

Deutsch, P. "An Interactive Program Verifier" Report No. CSL-73-1. Xerox PARC. May 1973.

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

Fahlman, Scott. "Thesis Progress Report" MIT AI Memo 331. May, 1975.

Fikes, R. and Hendrix, G. "A Network-Based Knowledge Representation and its Natural Deduction System" IJCAI-77. Cambridge, Mass. August 1977. pp 235-246.

Friedman, D. P. and Wise, D. S. "Aspects of Applicative Programming for Parallel Processing" IEEE Transactions of Computers. vol. C-27. No. 4. April 1978. pp. 289-296.

Godel, K. "The Consistency of the Axiom of Choice and of the Generalized Continuum Hypothesis with the Axioms of Set Theory" Annals of Mathematics Studies. No. 3, Princeton, 1940.

Goldstein, Ira P. "The Computer as Coach: An Athletic Paradigm for Intellectual Education" MIT AI Memo 389. December 1976.

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 on Principles of Programming Languages. Palo Alto, California. January, 1975.

Greussay, P. "Iterative Interpretation of Tail-Recursive LISP Procedures" Department d' Informatique. University of Vinceenes. September 1976. TR-20-76.

Hammer, M. and McLeod, D. "The Semantic Data Model: A Modeling Mechanism for Data Base Applications. SIGMOD Conference on the Management of Data. Austin Texas. May 31-June 2, 1978.

Hawkinson, Lowell. "The Representation of Concepts in OWL" Proceedings of IJCAI-75. September, 1975. Tiblisi, Georgia, USSR. pp. 107-114.

Hewitt, C. "PLANNER: A Language for Proving Theorems in Robots" IJCAI-69. Washington, D. C. May 1969. pp 295-302.

Hewitt, C. "Protection and Synchronization in Actor Systems" ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication, March 1975.

Hewitt, C. "Stereotypes as an ACTOR Approach Towards Solving the Problem of Procedural Attachment in FRAME Theories" "Proceedings of Interdisciplinary Workshop on Theoretical Issues in Natural Language Processing" June 1975. Cambridge, Mass.

Hewitt, C. "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot" Unpublished doctoral dissertation. MIT. 1971.

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

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 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" Proceedings of IFIP Congress 77. Toronto, August 8-12, 1977. pp. 987-992.

Hewitt, C. "Preliminary Design of the APIARY for VLSI Support of Knowledge-Based Systems" MIT AI Lab Working Paper 186. May 1979.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Specifying and Proving Properties of Guardians for Distributed Systems" MIT AI Lab Working Paper 172. December 1978. Revised April 1979. Proceedings of International Symposium on the Semantics of Concurrent Computation. Evian-les-bains, France. July 1979.

Hewitt, C. "Concurrent Systems Need Both Sequences and Serializers" MIT AI Lab Working Paper 179. December 1978. Revised April 1979.

Hewitt, C. "Some Controversial Conjectures Concerning the Semantics of Concurrent Systems" MIT, Berkeley, Stanford, PARC, Yale, CMU, and Harvard: fall 1978. MIT AI Working Paper. Forthcoming.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Security and Modularity in Message Passing" MIT AI Lab Working Paper 180. December 1978. Revised April 1979.

Hibbard, P. G.; Hisgen, A.; and Rodeheffer, T. "A Language Implementation Design for a Multiprocessor Computer System" IEEE/ACM 5th Annual Conference on Computer Architecture. April 1978. pp 66-72.

Hibbard, P. G.; Knueven, P.; and Leverett, B. W. "Issues in the Efficient Implementation and Use of Multiprocessing in Algol 68" Proceedings of the Fifth International Conference on the Design and Implementation of Algorithmic Languages. IRIA, Rocquencourt, France. May 1977. pp. 202-221.

Hoare, C.A.R. "Communicating Sequential Processes" CACM, Vol 21, No. 8. August 1978. pp. 666-677.

Ichbiah, J. D. and Morse, S. P. "General Concepts of the SIMULA-67 Programming Language" pp. 65-93.

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.

Kahn,K. M. "DIRECTOR Guide" MIT AI Memo 482. June 1978.

Kahn,G. and MacQueen, D.. "Coroutines and Networks of Parallel Processes" IFIP-77. Toronto. August 8-12, 1977. pp. 987-992.

King,J. C. "A New Approach to Program Testing" IBM Research Report RC-5037. September 1974.

Knuth, D. E. "Examples of Formal Semantics" Stanford AIM-126. July 1970.

Knuth, D. E. "Structured Programming with GO TO Statements" Computing Surveys. December 1974.

Kristensen, B. B.; Madsen, O. L.; Moller-Pedersen, B.; and Nygaard, K. . "A Definition of the BETA Language" TECHNICAL REPORT TR-8. Aarhus University. February 1979.

Landin, P. J. "A Correspondence between ALGOL 60 and Church's Lambda-notation" Part I, CACM Vol 8 pp 89-101. Part II . 1965.

Luckham, D. C. "Program Verification and Verification-Oriented Programming" Invited paper at IFIP-77. Montreal. August 1977. pp 783-793.

Manna, Z. and Waldinger R. "Structured Programming with Recursion" Stanford Artificial Intelligence Laboratory Memo AIM-307.

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.

Moore, J. and Newell, A. "How Can MERLIN Understand?" CMU AIM. November, 1973.

Moriconi, Mark S. "A Designer/Verifier's Assistant" SRI Technical Report CSL-80. October, 1978.

Naur, P. "An Experiment in Program Development" BIT Vol. 12. 1972. pp 347-365.

Palme, J. "Protected Program Modules in SIMULA-67" Swedish Research of National Defense. Report FOAP C8372-m3 (E5). July 1973.

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.

Reynolds, J. C. "Programming with Transition Diagrams" Report DAI-37. Department of Artificial Intelligence, University of Edinburgh. Also Report CSR-4-77. Department of Computer Science, University of Edinburgh. July, 1977.

Rich, C. and Shrobe, H. "Initial Report on a LISP Programmer's Apprentice" December 1976. AI-TR-354. IEEE Transactions on Software Engineering. SE-4. No. 6. November 1978. pp. 456-467.

Rich, C.; Shrobe, H. E.; Waters, R. C.; Sussman, G. J.; and Hewitt, C. E. "Programming Viewed as an Engineering Activity" MIT A.I. Memo 459. January 1978.

Rulifson, J. F.; Derksen, J. A.; and Waldinger, R. J. "QA4: A Procedural Calculus for Intuitive Reasoning" SRI Technical Note 73. November 1972.

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

Shrobe, H. "Logic and Reasoning For Complex Program Understanding" MIT PhD. Thesis, October 1978.

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.

Steele, G. L. "Debunking the 'Expensive Procedure Call' Myth" MIT Artificial Intelligence Memo 443. October 1977.

Steele, G. L. and Sussman, G. J. "The Revised Report on SCHEME a Dialect of LISP" Artificial Intelligence Memo 452. January 1978.

Steele, G. L. and Sussman, G. J. "Constraints" MIT Artificial Intelligence Memo 502. November 1978.

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

Sussman, G. J.; Winograd, T.; and Charniak, E. "MICRO-PLANNER Reference Manual" MIT AI Memo 203A. December 1972. Cambridge, Mass.

Sussman, G. J. "SLICES: At the Boundary between Analysis and Synthesis" MIT AI Lab Memo 433. July 1977.

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

Waters, R.C. "Automatic Analysis of the Logical Structure of Programs" MIT AI Laboratory TR-492. December 1978.

Wulf, W. A. "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology" ISI/RR-76-46. June 1976.

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

APPENDIX I --- Communication and State Change

The programming problems arising from use of individual **GOTO** and **ASSIGNMENT** commands have become well known over the last decade. However, the removal of these constructs from traditional programming languages has been found by many to be constricting and to result in inefficient programs. A number of proposals have been advanced to allow these commands in a restricted way [e.g. Knuth: 1974 and Reynolds: 1977].

For the past several years I have been investigating the message passing metaphor of computation and have found that it can be used to conveniently and efficiently implement iterative programs without ever thinking about individual **GOTO** or **ASSIGNMENT** commands. PLASMA [Hewitt and Smith: 1975, Hewitt: 1977, Hewitt and Atkinson: 1977 and 1979, Yonezawa: 1977] was one of the first concurrent programming languages with no special *primitive* control constructs for iteration. Iteration emerges as one of the patterns of passing messages that is inherent in the basic structure of PLASMA. The underlying message passing semantics of the language [Greif and Hewitt: 1975, Hewitt 1977, Hewitt and Baker: 1977, Hewitt and Attardi: 1978] led directly to the development of this paradigm for iteration. Before the development of PLASMA the general view [Allen and Cocke: 1972; Paterson and Hewitt: 1970; Strong: 1971] was that certain recursive procedures [sometimes called "tail-recursive" or "semi-recursive"] could be recognized by the compiler and translated as load, store, and transfer of control instructions. On a message passing machine in a language like PLASMA, no such "recognition" and "translation" is necessary.[1] However, in order to make PLASMA run efficiently on conventional machines it is necessary to recognize these special cases. Fortunately the language has been designed to make this task relatively easy. In fact programs in a language like PLASMA without individual **GOTO** and **ASSIGNMENT** commands can be optimized more easily than conventional programs.

However, this does not mean that Reynolds [1977] and Knuth [1974] are incorrect in their claim that their absence can be constricting in a more traditional programming methodology. In my view a fundamental limitation of the **GOTO** command is that it provides for the transfer of control to the target site of the **GOTO** without allowing for the possibility of sending along a message with the transfer of control. Thus using the

---

1: This may sound a bit mysterious to readers unfamiliar with the semantics of message passing. For an explanation of the reason see [Hewitt: 1977] and [Hewitt and Attardi: 1978].

GOTO construct the only way to provide information to the target site is to use a side effect on some resource shared with the target site. The side effect gratuitously limits the concurrency that is possible. Naively it might be thought that message passing is just an **ASSIGNMENT** followed by a **GOTO**. However this is incorrect because there is no state change inherent in sending and receiving messages between actors [Hewitt and Smith: 1975, Hewitt: 1977, Hewitt and Baker: 1977]. If the target actor of a message is **serialized** then arbitration is required to decide the order in which messages are processed. However, if the target actor is **unserialized** then no such arbitration is required and the target actor can process arbitrarily many messages in parallel.

The only way to cause a state change in the programming language used in this paper is to use a **serializer** [Hewitt: 1975, Hewitt and Atkinson: 1977, Hewitt, Attardi, and Lieberman: 1978, Hewitt and Attardi: 1978]. Serializers have the advantage that they encapsulate state change in a much more modular fashion than is accomplished by *individual* **ASSIGNMENT** and **GOTO** commands. Instead both state change and transfer of control are encapsulated in a single primitive that can accomplish both concurrently. We have found that this encapsulation tremendously increases the readability and modularity of modules that implement state change. The techniques used for efficiently optimizing the implementation of serializers for execution on conventional machines are similar to the ones which have been developed for tail recursion.

The result that concurrent programs can always be written efficiently without use of the **GOTO** construct does not in and of itself imply that programs can be written efficiently without use of state change (in which some actor changes its behavior). Greif and Hewitt [1975] have developed a procedure for effectively transforming any *sequential* program which makes use of state change into an equivalent program which does not cause any state changes. However, it is not known whether or not efficiency bounds which we derived are the best that can be obtained for typical sequential algorithms. State change is a significant source of difficulty in concurrent programs. It limits the amount of concurrency that is possible. Furthermore it can easily be a source of subtle bugs because of race conditions. The only use for state change in the programs considered in this paper is to implement communication between independent concurrent activities. It is currently an open research question what the important uses of state change are on an actor machine beyond implementing communication between independent concurrent activities.

Because of the development of extremely large scale integrated circuit chips and geographically distributed computer networks, it appears that at some point in the not too distant future almost all programs will execute in a multiprocessor environment. Thus we

should design our programming languages and algorithms to be appropriate for this new environment. The programming techniques and implementations presented in this paper have been constructed with this new environment in mind. In this new environment the message passing paradigm advocated in this paper produces much faster implementations for factorial and the path existence problem.

APPENDIX II --- Implementation of Cells using Serializers

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

```
(describe (create_cell =initial_contents)
   [is: (a Serialized_actor [responds_to: (⍁ (a Contents_query) (an Update))])]
   [implementation:
      (create_serialized_actor
         (a Cell [current_contents: initial_contents]))])

(describe (a Cell [current_contents: (an Actor)])
   [implementation:
         (create_unserialized_actor
            ((a Contents_query) produces
               (unlock [reply: current_contents])) ;reply sending back the current contents
                     ;unlock the serializer for the next message without changing the behavior
            ((an Update [next_contents: =n]) produces
               (unlock [become: (a Cell [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 --- 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 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.

The description system is intended to be used as a language of communication with the proposed Programming Apprentice. Its syntax looks somewhat like a version of **template** English [Hewitt: 1975, Bobrow and Winograd: 1977, Wilks: 1976] Thus for example we write (*an* Integer) in this paper instead of writing (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 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

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

where (*a* Set [element: 2] [element: 4]) is a partial description of {2 4 6}. 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⟼ ...x...]

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

(λ x. ...x...)

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⊢→ n^3]])
```

VI.1 --- Examples

VI.1.a --- Articulation

Articulation is an important capability of a description 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.

VI.1.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])])
```

which says that {|a b a|} can be viewed as a mapping in which a occurs with multiplicity 2, b occurs with multiplicity 1, and all other elements occur with multiplicity 0.

VI.1.c --- Transitive Relations

If (3 *is* (*an* Integer [<: 4])) and (4 *is* (*an* Integer [<: 5])), we can immediately conclude that

```
(3 is (an Integer [<: (an Integer [<: 5])]))
```

by the transitivity of predication. From this last statement, it 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 below 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 relationship R with m where R is a transitive relationship for concept, then x has the relationship R with m.

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

The desired conclusion can be reached by using the above description with concept bound to Integer, R bound to <, and m bound to 5.

VI.1.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 bound to Complex, R bound to real_part, description1 bound to (> 0), and description2 bound to (*an* Integer).

This example cannot be done in most type systems; the above solution makes use of the ω-order capabilities of our description system.

VI.1.e --- Self Description

Self description provides the ability for the programming Apprentice to reason about its own procedures. However we must beware of paradoxes. For example the following sentence clearly holds in ω order logic:

$$\forall P\ \forall x\ (P\ x)\ \textit{if\_and\_only\_if}\ (P\ x)$$

From the above sentence, we obtain the following by the usual rules for quantifiers:

$$\forall P\ \exists Q\ \forall x\ (Q\ x)\ \textit{if\_and\_only\_if}\ (P\ x)$$

Substituting the following mapping

$$(=s \longmapsto (\textit{not}\ (s\ s)))$$

for P, we get

$$\exists Q\ \forall x\ (Q\ x)\ \textit{if\_and\_only\_if}\ (\textit{not}\ (x\ x))$$

Using $\exists$-elimination with Q_0 for Q we get

$$\forall x\ (Q_0\ x)\ \textit{if\_and\_only\_if}\ (\textit{not}\ (x\ x))$$

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: (⊔ 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.

VI.2 --- Axioms

The description system is defined by its underlying behavioral 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, similar axiomatizations for FRL, KRL, OWL, MDS, etc. did not exist.

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$_3$>))

The descriptions in our system are **completely intentional.** I.e. the fact that the extension of two descriptions is the same does not force the conclusion that the descriptions are coreferential. Suppose we define **snarks** to be set of all animals which are both **herbivores** and **carnivores.** Then in Zermelo-Fraenkel set theory it follows that **(snarks ⊆ cows)** because the empty set is a subset of every other set. From the following statements

((*a* Snark) *is* (*a* Carnivore))
((*a* Snark) *is* (*a* Herbivore))
((*a* Carnivore) *is* ¬(*a* Herbivore))

we can conclude that

((*a* Snark) *is* ¬(*a* Herbivore))

by transitivity of predication. Thus we can conclude that nothing is a **Snark** because anything which is a **Snark** would necessarily be both a **Herbivore** and not a **Herbivore.** However this does *not* force the conclusion that

((*a* Snark) *is* (*a* Cow))

Another important axiom is

Reflexivity
(<description> *is* <description>)

which says that every description describes itself.

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. Note that <attributions> is a string of zero or more elements of category <attribution>.

Deletion
((*a* <description$_1$> <attributions$_1$> <attribution$_2$> <attributions$_3$>) *is*
(*a* <description$_1$> <attributions$_1$> <attributions$_3$>))

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[1] 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$_1$>)

1: We are grateful to Dana Scott, Maria Simi, and Jerry Barber for helping us to remove some bugs from these axioms

Criteriality

(*implies*
  (*and*
    (<description$_1$> *is* (*the_only* <description$_3$>))
    (<description$_2$> *is* (*the_only* <description$_3$>)))
  (<description$_1$> *coref* <description$_2$>))

Constrained Description

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

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$_2$>)))

Shift in Focus

(<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$_2$>)))))

Definite Selection

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

Complementation

(¬¬<description> *coref* <description>)

(<description$_1$> *is* ¬<description$_2$>) *if_and_only_if* (<description$_2$> *is* ¬<description$_1$>)

((<description$_1$> *is* ¬<description$_2$>) *implies*
  (∀ =d
    (*implies*
      (d *is* <description$_1$>)
      (*not* (d *is* <description$_2$>)))))

Meet

(⟨description$_1$⟩ *is* (⊓ ⟨description$_2$⟩ ⟨description$_3$⟩)) *if_and_only_if*
  (*and*
    (⟨description$_1$⟩ *is* ⟨description$_2$⟩)
    (⟨description$_1$⟩ *is* ⟨description$_3$⟩))

((⊓ ⟨description$_1$⟩ ⟨description$_2$⟩) *is* ⟨description$_2$⟩)

(¬(⊓ ⟨description$_1$⟩ ⟨description$_2$⟩ *coref* (⊔ ¬⟨description$_1$⟩ ¬⟨description$_2$⟩)))

Join

((⊔ ⟨description$_2$⟩ ⟨description$_3$⟩) *is* ⟨description$_1$⟩) *if_and_only_if*
  (*and*
    (⟨description$_2$⟩ *is* ⟨description$_1$⟩)
    (⟨description$_3$⟩ *is* ⟨description$_1$⟩))

(⟨description$_1$⟩ *is* (⊔ ⟨description$_1$⟩ ⟨description$_2$⟩))

(¬(⊔ ⟨description$_1$⟩ ⟨description$_2$⟩ *coref* (⊓ ¬⟨description$_1$⟩ ¬⟨description$_2$⟩)))

Disjoint Join

((⊍ ⟨description$_1$⟩ ⟨description$_2$⟩) *coref*
  (⊔.
    (⊓ ⟨description$_1$⟩ ¬⟨description$_2$⟩)
    (⊓ ¬⟨description$_1$⟩ ⟨description$_2$⟩)))

Conditional Description

(⟨description$_1$⟩ *is* (⟨description$_2$⟩ *if* ⟨statement⟩)) *if_and_only_if*
  (⟨statement⟩ *implies* (⟨description$_1$⟩ *is* ⟨description$_2$⟩))

VI.3 --- 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> |
                  <attribution> |
                  <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>†) |      ;⊓ designates the meet of desciptions
                  (⊔ <description>†) |      ;⊔ designates the join of descriptions
                  (⊍ <description>†) |      ;⊍ designates disjoint join of descriptions
                  ¬<description> |      ;¬ designates the complement of a description
                  (<relation> <description>*)

<criterial_description> ::= (the_only <description>†)
            ;only used for descriptions that describe exactly one thing

<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>†]
<attributions> ::= <attribution>*
<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>
```