A SCHEME for Distributed Processes

by

Rex A. Dwyer

R. Kent Dybvig

Computer Science Department

Indiana University

Bloomington, Indiana 47405

TECHNICAL REPORT No. 107

A SCHEME FOR DISTRIBUTED PROCESSES

REX A. DWYER

R. KENT DYBVIG

APRIL, 1981

This material is based upon work supported by the National Science Foundation under Grant MCS79-04183.

A SCHEME for Distributed Processes

Rex A. Dwyer

R. Kent Dybvig

ABSTRACT

This paper presents an implementation of the process communication and synchronization concepts of Per Brinch Hansen's Distributed Processes [6] built from an applicative framework. This system is an object-oriented implementation of Steele and Sussman's SCHEME [21] coded in SIMULA. The system was expanded to include first facilities for quasi-parallel processing and then the communication and synchronization primitives of Distributed Processes.

SCHEME is a full-funarg dialect of LISP with lexical binding. Distributed Processes is a concurrent programming language in which communication between processes occurs when one process invokes a common procedure defined within another process. These invocations (external requests) must be handled one at a time by the invoked process; the invoking process remains idle until the request is satisfied.

A special system process, the Distributor, initially interacts with the user to execute single-process programs and to allow run-time (rather than compile-time) process creation. The Distributor supervises execution of user-created processes once the user initiates multi-processing.

Several example programs are examined, including two dealing with "inward" and "outward funargs" -- function closures used as arguments to and results of external requests. The implementation presented is easy to extend and would lend itself well to experimentation with other models of concurrency.

INTRODUCTION

This paper describes SDP (SCHEME Distributed Processes), an implementation of the process synchronization and communication concepts of Per Brinch Hansen's Distributed Processes model [6] in a SCHEME-based system. SCHEME is a lexically-bound full-funarg dialect of LISP described by Steele and Sussman [21]. Distributed Processes is a high-level programming language for describing algorithms as sets of essentially sequential processes executing in parallel with no common variables and limited communication through "external requests" (similar to "monitor calls" [2,4,5,16]) made of one process by another.

Implementation proceeded in three stages:

1. implementation of an object-oriented interpreter in SIMULA for a subset of SCHEME (This interpreter was partially based on the ALONZO interpreter [12], a terse non-recursive interpreter for a subset of SCHEME which permits no side effects.),

2. implementaition of pure multi-processing, including facilities for definition of process prototypes, creation of processes, and quasi-parallel execution of processes,

3. graftage of the communication primitives of Distributed Processes onto the multi-processing system.

Our implementation includes all of SCHEME's [21] important features except the fluid binding primitives FLUIDBIND, FLUID, and FLUIDSET, the function definition form DEFINE (ASET suffices here), and the SCHEME multiprocessing primitives. It differs slightly in some aspects from standard SCHEME, specifically:

1. the "truth values" are represented by TRUE and FALSE, not T and NIL;

2. a clear distinction is made between the atom FALSE and the empty list () ;

3. only true lists [1] are implemented rather than arbitrary S-expressions, i.e. no atom may appear in the CDR position of a CONS-cell;

4. arrays and partial application of functions are implemented as described in the brief user's guide.

Brinch Hansen's Distributed Processes is a descendant of his earlier Concurrent Pascal language and is statement-oriented whereas SCHEME is expression oriented. For this reason, and additionally because of the added generality of some aspects of our system, our interpretation of the DP model will be described in some detail.

A program in SDP consists of an expression to be evaluated by the interpreter. It may be a simple SCHEME expression, in which case the value of the expression is computed and printed. Or, it may be an expression which defines process prototypes, uses them to create processes, and then enters a multi-processing phase in which the created processes execute in quasi-parallel. A process prototype is defined by a GAMMA-expression, which has the form:

(GAMMA (<process parameter>*)
(<common function definition>*)
<initial statement>)

(Note that we have replaced common procedures having input and output parameters by common functions which take arguments and return a value. This is more in keeping with the applicative style of SCHEME. Note also the added generality of processes with parameters, allowing the initial values of certain variables to be specified at process creation time rather than at prototype definition time.)

Processes are created by invoking a process prototype with actual parameters using a syntax exactly like that of SCHEME function invocation. Several processes may be created using the same prototype and the same or different parameters. Such processes have nothing in common except the code of initial statements and function definitions. Process creation does not begin execution of the process; rather, it places a new process into the parallel processing queue so that its execution will take place when the parallel processing phase begins.

Each process has its own "global environment" which can be accessed only by the process itself. It also has access to a read-only "primitive environment", shared by all processes, which contains definitions of primitive functions available to all processes.

Processes may be given names by assigning the result of the prototype invocation to a variable or array element in the primitive environment using the pseudo-function PSET. PSET is similar to ASET, but operates on the primitive environment rather than the global environment. PSET can only be used by the "main process" (or Distributor -- see implementation notes); it cannot be used by processes to communicate through the primitive environment during the parallel-processing phase.

Consider example 1, an implementation of a bounded buffer [17]. This program defines three different process prototypes in the LABELS-expression beginning at line 2: a consumer prototype CON (line 3) with one parameter and no common functions; a producer prototype PROD (line 14) with one parameter and no common functions; and BUFFT (line 15) the bounded buffer prototype, with the buffer length BLEN as a parameter, and two common functions: PUT, for inserting into the buffer; and GET, for removing from the buffer.

Producer and consumer processes are created and automatically readied for execution in lines 37 and 39 respectively. The buffer process must be referenced by the producers and consumers, and is therefore assigned to the variable BUFF (line 34). (Alternatively, it could be passed to the other processes as a parameter.)

Communication between processes is carried out by external requests, or invocations of another process's common functions. This takes the following syntactic form:

(CALL <process-exp> <function-name-exp> <arg>*)

where <process-exp> evaluates to some process, but <function-name-exp> evaluates to the name of a common function defined in that process, i.e. an atom. Examples of CALLs are in lines 13 and 14 of example 1. When a process makes an external request, it must remain idle until the request is answered.

When the "main process" evaluates the RUN primitive, the parallel-processing phase begins. Those processes which have been created begin execution in quasi-parallel. No new processes may be created once this phase begins.

Initially, each process executes its initial statement. Either the initial statement executes to completion, or a guarded region with no true guard is encountered. In either case, the process is freed to handle external requests if any are present. Once the execution of an external request is undertaken, it is executed until either it is completed and a value is returned, or a guarded region is encountered.

Then the process is freed to handle other requests or return to its initial statement, etc.

Our interpretation of guarded commands [8,9] and guarded regions is quite similar to Brinch Hansen's; the reader is referred to his work [6] for a complete explanation. We use syntax similar to LISP's COND. Briefly, when no guard is true, the COND expression causes an error termination; DO terminates its repetition; WHEN "waits" by releasing the process for other requests; and CYCLE "waits" like WHEN, but repeats without terminating. If some guards are true, an expression corresponding to a true guard is selected and evaluated.

We will now consider the execution of our example. The "main program" (lines 36 - 43) creates a buffer named BUFF of length BLEN, then creates NPROD producers, each with a unique stamp to put on its work, and NCON consumers with their own indices. The parallel processing phase is begun by RUN.

The producers and consumers are fairly simple. Producers constantly try to put their product (their ME number) into the buffer by calling the buffer's PUT function (line 14). Consumers constantly try to remove the product from the buffer by calling the buffer's GET function (line 13). They package the product and print it out (lines 9 - 13). Producers and consumers never handle external requests, and their initial statements never terminate.

The buffer process has two variables -- SEQ and LEN -- which are

the focus of its efforts. SEQ is a list of the elements in the buffer, and LEN is the number of elements. The buffer is initialized to be empty, then the initial statement terminates and the process is freed to handle PUT and GET requests (line 32).

When the buffer process handles a GET request, it checks to see if the buffer is empty (lines 20 - 21). If so, the GET request must wait, and the process is freed to handle other requests. If not, the first element is removed from the buffer and returned as the value of GET. The consumer proceeds with this value, and the buffer advances to handle another request.

When the buffer process handles a PUT request, it checks to see if the buffer is full (lines 28 - 29). If so, the PUT request must wait; otherwise, the argument passed to PUT is appended to the end of SEQ and LEN is incremented (lines 30 - 31). The new value of LEN is returned, but ignored by the producers. (Conceivably, more sophisticated producers might use this information to choose among short and long production tasks, or to choose among several buffers as destinations for the next product.)

The output of an execution of this program is shown below. Several more examples with sample execution output and short commentaries follow. Section 2 deals with some implementation details; Section 3 is a brief user's guide. The code of the interpreter follows in the appendix.

example 1: Bounded Buffer

```
 1 ((LAMBDA(BLEN NPROD NCON)
 2   (LABELS
 3    ((CON
 4      (GAMMA
 5       (ME)
 6       ()
 7       (DO
 8        (TRUE
 9         (PRINT
10          (LIST 'CONSUMER
11                ME
12                'PRODUCER
13                (CALL BUFF 'GET)))))))
14     (PROD (GAMMA (ME) () (DO (TRUE (CALL BUFF 'PUT ME)))))
15     (BUFFT
16      (GAMMA
17       (BLEN)
18       ((GET
19         (LAMBDA ()
20          (WHEN
21           ((< 0 LEN)
22            (PROGN (ASET 'LEN (SUB1 LEN))
23                   (ASET 'RES (CAR SEQ))
24                   (ASET 'SEQ (CDR SEQ))
25                   RES)))))
26        (PUT
27         (LAMBDA(NEW)
28          (WHEN
29           ((< LEN BLEN)
30            (PROGN (ASET 'SEQ (APPEND SEQ (LIST NEW)))
31                   (ASET 'LEN (ADD1 LEN))))))))
32       (PROGN (ASET 'SEQ '()) (ASET 'LEN 0)))))
33    (PROGN
34     (PSET 'BUFF (BUFFT BLEN))
35     (DO
36      ((< 0 NPROD)
37       (PROGN (PROD NPROD) (ASET 'NPROD (SUB1 NPROD)))))
38     (DO
39      ((< 0 NCON) (PROGN (CON NCON) (ASET 'NCON (SUB1 NCON)))))
40     (RUN))))
41  3 4 3)
```

```
----> distributor <----
#(DSKIN 'BDBUFF.EXP)

run
----> multi-processing <----
(CONSUMER 1 PRODUCER 2)
(CONSUMER 3 PRODUCER 1)
(CONSUMER 2 PRODUCER 4)
(CONSUMER 1 PRODUCER 3)
(CONSUMER 3 PRODUCER 2)
(CONSUMER 2 PRODUCER 1)
(CONSUMER 1 PRODUCER 4)
(CONSUMER 3 PRODUCER 2)
(CONSUMER 2 PRODUCER 1)
(CONSUMER 1 PRODUCER 4)
(CONSUMER 3 PRODUCER 2)
(CONSUMER 2 PRODUCER 1)
(CONSUMER 1 PRODUCER 4)
(CONSUMER 3 PRODUCER 2)
(CONSUMER 2 PRODUCER 1)
(CONSUMER 1 PRODUCER 3)
(CONSUMER 3 PRODUCER 2)
(CONSUMER 2 PRODUCER 1)
(CONSUMER 1 PRODUCER 3)
(CONSUMER 3 PRODUCER 2)
(CONSUMER 2 PRODUCER 1)
(CONSUMER 1 PRODUCER 4)
(CONSUMER 3 PRODUCER 2)
(CONSUMER 2 PRODUCER 1)
(CONSUMER 1 PRODUCER 4)
```

example 2: Fibonacci numbers

In this example [11], an array of three processes is created. Each of these processes has a list SEQ of numbers. When this list contains 2 numbers (the last 2 Fibonacci numbers) they are added, the list is emptied, and the result of the addition (a new Fibonacci number) is passed on to the other two processes by calling their PUT functions. In this way, each process computes every third Fibonacci number.

At creation time, each process is given its own array subscript as well as the subscripts of the processes to which it must pass its results. In addition, it receives an initial sequence and its length.

```
----> distributor <----
#(DSKIN 'FIBON.EXP)

(BETA () (LABELS ((FIBT (GAMMA (ME ...
#(FIBON)

run
----> multi-processing <----
(FIB 1 COMPUTES 0)
(FIB 2 COMPUTES 1)
(FIB 3 COMPUTES 1)
(FIB 1 COMPUTES 2)
(FIB 2 COMPUTES 3)
(FIB 3 COMPUTES 5)
(FIB 1 COMPUTES 8)
(FIB 2 COMPUTES 13)
(FIB 3 COMPUTES 21)
(FIB 1 COMPUTES 34)
(FIB 2 COMPUTES 55)
```

```
(ASET 'FIBON
(LAMBDA ()
 (LABELS
  ((FIBT
    (GAMMA
     (ME ONE TWO SEQ LEN)
     ((PUT
       (LAMBDA(X)
        (PROGN (ASET 'SEQ (CONS X SEQ))
               (ASET 'LEN (ADD1 LEN))))))
     (CYCLE
      ((= LEN 2)
       (PROGN
        (PRINT
         (LIST 'FIB
               ME
               'COMPUTES
               (ASET 'RES (EVAL (CONS 'PLUS SEQ)))))
        (ASET 'SEQ '())
        (ASET 'LEN 0)
        (CALL (ACCESS (FIB ONE)) 'PUT RES)
        (CALL (ACCESS (FIB TWO)) 'PUT RES)))))))
  (PROGN (PSET 'FIB (ARRAY 3))
         (STORE (FIB 1) (FIBT 1 2 3 '(0 0) 2))
         (STORE (FIB 2) (FIBT 2 3 1 '(1) 1))
         (STORE (FIB 3) (FIBT 3 1 2 '() 0))
         (RUN))))
)
```

example 3: dining philosophers

In this example (problem due to E.W. Dijkstra, outlined by C.A.R. Hoare in [17]) five philosophers are seated around a table. Each philosopher is assigned a number from 1 to 5. To the right of each philosopher on the table lies a fork assigned the same number. A platter of spaghetti which the philosophers share rests on the table. When a philosopher wishes to eat he must use both the fork on his right and the fork on his left, for example philosoper 3 must use forks 2 and 3 to eat.

A potential problem is that one or more philosophers may be starved by bad timing. In the worst case all may be starved if each philosopher holds one fork. The solution given here requires that odd-numbered philosophers reach first for the right-hand fork and even-numbered philosophers for the left-hand fork. Actually, only one left-handed philosopher is needed in the group to prevent deadlock.

Two process prototypes, a FORK process and a PHIL process, are used in this example. Forks are merely binary semaphores [7] defined very elegantly with just one function, USE, taking advantage of the "inward funarg" capability of our system. USE simply evaluates the closure (TO-DO) passed to it. (Binary semaphores are normally implemented with two functions, P for getting the semaphore and V for releasing the semaphore.) To use a fork a philosopher specifies to the fork process what he wants to do, and the fork allows him to do it.

A philosopher first sets up THINK and EAT functions with the help of BIND (see User's Guide below). It then loops forever, THINKing, then USEing one fork to USE another fork to EAT.

The main program sets up the philosophers and forks and runs multiprocessing.

```
(ASET 'GOPHIL
 (LAMBDA (NPHILS)
  (LABELS
   ((FORKP (GAMMA (I) ((USE (LAMBDA (TO-DO) (TO-DO)))) '()))

    (PHILP (GAMMA
            (ME FORK1 FORK2)
            ()
            (LABELS
             ((ACT (LAMBDA (ACTION)
                           (PROGN (PRINT (LIST ME ACTION))
                                  (PRINT (LIST ME 'STOPPED ACTION))))))
             (PROGN
              (ASET 'THINK (BIND ACT 'ACTION 'THINKING))
              (ASET 'EAT (BIND ACT 'ACTION 'EATING))

              (DO (TRUE
                   (PROGN
                    (THINK)
                    (CALL FORK1 'USE (LAMBDA () (CALL FORK2 'USE EAT)))
                   ))))))))
   (LABELS
    ((RFORK (LAMBDA (X) (ACCESS (FORK X))))
     (LFORK (LAMBDA (X) (ACCESS (FORK (IF (= X 1) NPHILS (SUB1 X))))))
     (MAKEPHIL (LAMBDA (I) (IF (ODD I)
                               (PHILP I (RFORK I) (LFORK I))
                               (PHILP I (LFORK I) (RFORK I))))))
    (PROGN
     (ASET 'FORK (ARRAY NPHILS))
     (ASET 'X NPHILS)
     (DO ((NOT (= X 0))
          (PROGN (STORE (FORK X) (FORKP X)) (ASET 'X (SUB1 X)))))

     (ASET 'X NPHILS)
     (DO ((NOT (= X 0)) (PROGN (MAKEPHIL X) (ASET 'X (SUB1 X)))))

     (RUN))))))
```

```
#(GOPHIL 5)

run
----> multi-processing <----
(2 THINKING)
(3 THINKING)
(5 THINKING)
(1 THINKING)
(2 STOPPED THINKING)
(1 STOPPED THINKING)
(5 STOPPED THINKING)
(3 STOPPED THINKING)
(4 THINKING)
(4 STOPPED THINKING)
(2 EATING)
(5 EATING)
(2 STOPPED EATING)
(5 STOPPED EATING)
(2 THINKING)
(4 EATING)
(1 EATING)
(2 STOPPED THINKING)
(4 STOPPED EATING)
(5 THINKING)
(1 STOPPED EATING)
(5 STOPPED THINKING)
(4 THINKING)
(4 STOPPED THINKING)
(3 EATING)
(3 STOPPED EATING)
(1 THINKING)
(1 STOPPED THINKING)
(5 EATING)
(3 THINKING)
(2 EATING)
(2 STOPPED EATING)
(3 STOPPED THINKING)
(5 STOPPED EATING)
(4 EATING)
(2 THINKING)
(5 THINKING)
(1 EATING)
(1 STOPPED EATING)
(4 STOPPED EATING)
(2 STOPPED THINKING)
(5 STOPPED THINKING)
(1 THINKING)
(4 THINKING)
```

example 4: sorting array

In this example [6], a series of numbers is sorted by an array of processes. Each process has a list SEQ of 0, 1, or 2 numbers. When this list contains two elements, the process sends the larger on to its successor in the array.

A list of numbers is sorted by first PUTting each of the numbers in the first process's list, then GETting the sorted numbers one at a time from the first process. This is the job of the USER process.

The "main program" sets up a USER process and an array of SORTP processes of sufficient length to sort the given input. The process's subscript ME is passed so that the process can calculate the subscript of its successor SUCC. SUCC = TRUE for the last element of the array.

```
----> distributor <----
#(DSKIN 'SORT.EXP)

(BETA (INPUT) (LABELS ....
#(RUNSORT '(11 44 66 55 22 33))

run
----> multi-processing <----
(66 55 44 33 22 11)
```

```
(ASET 'RUNSORT
(LAMBDA (INPUT)
  (LABELS
   ((SORTP
     (GAMMA (ME)
      ((PUT
        (LAMBDA(I)
         (WHEN ((< (LENGTH SEQ) 2) (ASET 'SEQ (CONS I SEQ))))))
       (GET
        (LAMBDA ()
         (WHEN
          ((= (LENGTH SEQ) 1)
           (PROGN (ASET 'RES (CAR SEQ)) (ASET 'SEQ '()) RES))))))
      (PROGN
       (ASET 'SEQ '())
       (ASET 'REST 0)
       (ASET 'SUCC (ACCESS (SORT (ADD1 ME))))
       (CYCLE
        ((= (LENGTH SEQ) 2)
         (PROGN
          (IF (< (CAR SEQ) (CAR (CDR SEQ)))
              (ASET 'SEQ (LIST (CAR (CDR SEQ)) (CAR SEQ)))
              TRUE)
          (CALL SUCC 'PUT (CAR SEQ))
          (ASET 'SEQ (CDR SEQ))
          (ASET 'REST (ADD1 REST))))
        ((AND (= (LENGTH SEQ) 0) (< 0 REST))
         (PROGN (ASET 'SEQ (LIST (CALL SUCC 'GET)))
                (ASET 'REST (SUB1 REST))))))))
    (USERP
     (GAMMA (INPUT INLEN)
      ()
      (PROGN
       (ASET 'OUTPUT '())
       (ASET 'SORT1 (ACCESS (SORT 1)))
       (DO
        ((NOT (NULL INPUT))
         (PROGN (CALL SORT1 'PUT (CAR INPUT))
                (ASET 'INPUT (CDR INPUT)))))
       (DO
        ((NOT (= (LENGTH OUTPUT) INLEN))
         (ASET 'OUTPUT (CONS (CALL SORT1 'GET) OUTPUT))))
       (PRINT OUTPUT)))))
    (PROGN
     (PSET 'USER (USERP INPUT (LENGTH INPUT)))
     (ASET 'X (LENGTH INPUT))
     (PSET 'SORT (ARRAY (ADD1 X)))
     (STORE (SORT (ADD1 X))  TRUE)
     (DO
      ((> X 0)
       (PROGN (STORE (SORT X) (SORTP X)) (ASET 'X (SUB1 X)))))
     (RUN)))))
```

example 5: classroom

In this example, a teacher process supervises a class of 5 pupils. Each pupil can focus his attention ATTN on his problems (ATTN = PROBLEM), or he can gaze out the window (ATTN = WINDOW). The teacher checks each pupil in turn, and assigns new problems to pupils who are gazing through the window. Pupils may be gazing through the window because they have no problems to do {the (NOT (NULL PROBSET)) condition in the PUPILT process prototype} or because of a random fit of laziness {the (> (TIMES ME 5) (RANDOM)) condition}.

The teacher does not check the status of a pupil by making an ordinary external request to the pupil via a common function -- if the teacher did this, she would have to wait until the pupil was not working to find out if he was working or not! Rather, the teacher collects closures of status-checking functions (FOCUS) at the beginning of the class session by making an external request to each pupil's GETFOCUS function, and stores these closures in her FOCI array. The teacher may then invoke the closure to check a pupil's status without waiting for the pupil to process an external request.

In this way (by returning closures as values of common functions -- an "outward funarg"), a process can grant permission to another process to perform specific operations in (and on) its environment without making an external request for each operation.

Note that in this example, assigning a new problem is still an external-request operation -- this insures that, if "team teaching" were undertaken, a pupil would not miss assignments because two teachers were talking to him at once!

```
----> distributor <----
#(DSKIN 'TEACH)

(LABELS ((TEACHER (GAMMA () () (CATCH QUIT ...
#
#(EVAL CLASS)

run
----> multi-processing <----
(STUDENT 2 SAYS 41 + 50 = 91)
(STUDENT 3 SAYS 64 + 43 = 107)
(STUDENT 4 SAYS 76 + 80 = 156)
(STUDENT 5 SAYS 43 + 75 = 118)
(STUDENT 2 SAYS 95 + 1 = 96)
(STUDENT 1 SAYS 97 + 17 = 114)
(STUDENT 3 SAYS 86 + 34 = 120)
(STUDENT 5 SAYS 18 + 29 = 47)
(STUDENT 4 SAYS 14 + 11 = 25)
(STUDENT 2 SAYS 82 + 92 = 174)
(STUDENT 3 SAYS 29 + 84 = 113)
(STUDENT 4 SAYS 94 + 21 = 115)
(STUDENT 1 SAYS 80 + 34 = 114)
(STUDENT 5 SAYS 27 + 100 = 127)
(STUDENT 1 SAYS 11 + 75 = 86)
(STUDENT 2 SAYS 20 + 61 = 81)
(STUDENT 4 SAYS 95 + 17 = 112)
(STUDENT 5 SAYS 25 + 74 = 99)
(STUDENT 3 SAYS 82 + 2 = 84)
(STUDENT 2 SAYS 99 + 86 = 185)
(STUDENT 4 SAYS 91 + 70 = 161)
(STUDENT 1 SAYS 17 + 44 = 61)
(STUDENT 5 SAYS 88 + 89 = 177)
(TEACHER QUITS)
(STUDENT 3 SAYS 43 + 65 = 108)
(STUDENT 2 SAYS 16 + 35 = 51)
(STUDENT 1 SAYS 70 + 70 = 140)
```

```
(ASET 'CLASS
  '(LABELS
   ((TEACHER
     (GAMMA ()
      ()
      (CATCH
       QUIT
       (PROGN
        (ASET 'LEFT 25)
        (ASET 'FOCI (ARRAY 5))
        (ASET 'I 5)
        (DO
         ((> I 0)
          (PROGN (STORE (FOCI I) (CALL (ACCESS (PUPIL I)) 'GETFOCUS))
                 (ASET 'I (SUB1 I)))))
        (ASET 'I 1)
        (DO ((< LEFT 0) (QUIT (PRINT '(TEACHER QUITS))))
            ((= I 6) (ASET 'I 1))
            ((< I 6)
             (IF (= ((ACCESS (FOCI I))) 'WINDOW)
                 (PROGN
                  (CALL (ACCESS (PUPIL I)) 'NEWPROB (RANDOM) (RANDOM))
                  (ASET 'LEFT (SUB1 LEFT))
                  (ASET 'I (ADD1 I)))
                 (ASET 'I (ADD1 I)))))))))
    (PUPILT
     (GAMMA (ME)
      ((GETFOCUS (LAMBDA () (LABELS ((FOCUS (LAMBDA () ATTN))) FOCUS)))
       (NEWPROB (LAMBDA (X Y)
                  (ASET 'PROBSET (CONS X (CONS Y PROBSET))))))
      (PROGN
       (ASET 'ATTN 'WINDOW)
       (ASET 'PROBSET '())
       (CYCLE
        ((AND (NOT (NULL PROBSET)) (> (TIMES ME 5) (RANDOM)))
         (PROGN
          (ASET 'ATTN 'PROBLEM)
          (PRINT
           (LIST 'STUDENT ME 'SAYS
                 (CAR PROBSET) '+ (CAR (CDR PROBSET)) '=
                 (PLUS (CAR PROBSET) (CAR (CDR PROBSET)))))
          (ASET 'PROBSET (CDR (CDR PROBSET)))
          (ASET 'ATTN 'WINDOW))))))))
   (PROGN
    (TEACHER)
    (PSET 'PUPIL (ARRAY 5))
    (ASET 'I 5)
    (DO
     ((< 0 I) (PROGN (STORE (PUPIL I) (PUPILT I)) (ASET 'I (SUB1 I)))))
    (RUN))))
```

II. IMPLEMENTATION NOTES

SCHEME

The implementation is coded in SIMULA and is object-oriented. Corresponding to each SCHEME "magic form" is a SIMULA EXPR- (for expression) class, e.g. an IFF class corresponding to the IF magic form. The parser receives SCHEME code from the read routine in the form of atoms and lists, examines the first element of every list, and replaces the LIST-object with an object of the form indicated by the first element, e.g. if the first element of a list is the atom QUOTE, the list is replaced by a QUOTE-object which has a pointer to the quoted expression (the second element in the list).

Each of these magic forms is evaluated in its own way. The method of evaluation is coded in the INSTANCE classes, one (or more) of which correspond to each magic form, e.g. IFFINST, IF1INST. INSTANCE objects are the building blocks of the control stack. When it is necessary to evaluate an EXPR-object (parsed expression), this is done by calling that EXPR-object's INSTANTIATE function, which returns as its value a new object of the proper INSTANCE-class, complete with return and environment pointers. This INSTANCE-object becomes the new top of the control stack and evaluation proceeds by RESUMEing the new stack-top object. This stack of INSTANCE-objects corresponds to the C-link of Steele and Sussman.

An environment is an object of class ENVIRON. A ribcage implementation of environments is used, so that supplying too few or too many arguments to a function is not necessarily a detectable error. The former is detected only if the formal parameter with no corresponding actual parameter is referenced. The latter is never detectable, but it makes possible a programming trick -- supplying extra arguments in a function call whose evaluation will cause side effects. The side effects are carried out before the function is invoked even though the values of the extra arguments are not bound to formal parameters. (A similar trick is possible in SNOBOL4[15].)

Objects of class ENVIRON cannot be created directly by a user program, making it impossible for the user to manufacture arbitrary environments to use in calls to EVAL in the manner possible in most LISP systems.

Our SCHEME system is designed as a subsystem similar to the ILISP subsystem. The top level of our system is a read-eval-print loop. We have implemented standard LISP functions DSKIN and DSKOUT for saving function definitions and variable bindings. A structure editor containing the most useful commands from the UCI LISP editor is invoked by calling the function EDIT with the expression to be edited as an argument. Also, the system function ECHO can be used to record the input and output of a SCHEME session on a disk file.

Many interesting problems in distributed processing involve arrays of processes. This implementation includes arrays which may contain elements of any type -- processes, function closures, other arrays, or simple atoms and lists. Arrays of any dimension are implemented as one-dimensional arrays using dope vectors. A call to ARRAY with one or more numeric arguments produces an array accessing function with a pointer to the one-dimensional array representing it. Evaluation of this function with the subscripts of an element in the array produces a pointer to this element which is used by primitives STORE and ACCESS.

Processes and Parallel Processing

The base of the SCHEME multiprocessing system is a network of processes. This network contains zero or more user-defined processes in addition to one system process, the Distributor. Each process has its own global environment and its own C-link. Initially the Distributor runs alone and answers requests (runs user programs). With no other processes running this is simply uni-processing running on the Distributor.

The Distributor no longer answers requests from the user when there are other processes running. It exists only to drive the processes that have been placed in the circularly linked multi-processing queue. The distributor randomly RESUMEs each process's C-link between zero and ten times as it goes around the multi-processing queue. Even after every process has completed its initial statement, the Distributor continues around the queue trying to find a process to RESUME. This loop is

interrupted only when the user types ^G. This returns the Distributor to uni-processing mode, where it responds to user input.

Processes

The user may define process prototypes and create processes only in the uni-processing mode. The definition of process prototypes is very similar to the definition of functions in SCHEME. Just as LAMBDA is used for functions, GAMMA is used for processes.

GAMMA form is a SCHEME magic form with 3 elements. The second is a "deflist" which is exactly the same as in a SCHEME LABELS expression. This deflist defines functions which will be global to the process being defined (These are necessary for implementation of Distributed Processes -- see the following section). The first and third elements in a GAMMA expression, the parameter list and the initial statement (body), correspond to the 2 elements of a SCHEME LAMBDA expression.

Evaluation of a GAMMA expression results in a closure (called an ALPHA-closure) with a special environment. This environment contains only the function definitions given in the second element of the GAMMA expression.

Application of an ALPHA-closure places a process in the multi-processing queue with a global environment which now contains both the arguments to the call and the global function definitions.

The user may create as many processes as desired before running multi-processing. Evaluation of the primitive RUN by the Distributor causes it to drop any user program currently executing and begins multi-processing. The multi-processing queue is circularly linked at this time.

Process Environments

There is a hierarchy of environments in the parallel processing system. Each process (including the Distributor) has its own global environment. Also, as each C-link grows, so does the local environment. In addition to these global environments there is a primitive environment. This environment originally contains system functions (primitives) such as CONS and ADD1. New variables and bindings may be placed in the primitive environment only by a program executed by the Distributor. Although all processes can 'see' the primitive environment, they cannot alter it. This means that once parallel processing begins the primitive environment remains static, preventing processes from communicating with or effecting another process through a shared environment.

ASET is the standard SCHEME primitive used by a process to make bindings in its own global environment. PSET is used to set the primitive environment and can only be used in a program executed by the Distributor.

Distributed Processes Queueing Mechanism

The crux of the DP model is the CALL command with which processes communicate. When the CALL command is invoked, the invoking process requests that another process evaluate one of its common functions using arguments supplied by the invoking process. Since the process invoked can handle only one request at a time, some queueing mechanism is appropriate. This mechanism will now be described.

Associated with each process are three local variables: NEXTINQ points to the process's successor in its own queue when the process is not waiting for the answer to a request from another process, and it points to the process's successor in another process's queue when it has made a request to that process. SAVEQ points to the process's successor in its own queue whenever the process is waiting for the answer to a request. NOWSERVING points to the process for which the process is currently handling a request. Note that a process may be in its own queue, and that a NOWSERVING variable may point to its owner. In fact, this is the case initially for each process as it is executing its own initial statement. If a process makes a request to another process (CALL), then it must save its successor in its own request queue in SAVEQ and join the other process's request queue by having its NEXTINQ pointer changed. (It must also remove itself from the multiprocessing queue, since it can proceed no further without the answer to its request.) When the request is answered, it is removed from the called process's request queue and its own request queue successor is restored from SAVEQ.

The following situation involving three processes, A, B, and C, should clarify the use of these variables:

Fig. 1: Initially, all processes are, intuitively speaking, handling their own requests. They are in their own queues, and their NOWSERVING pointers point back to themselves. All are in the multi-processing queue and are executing their initial statements.

Fig. 2: A, in executing its initial statement, has made a CALL to one of C's common functions. A saves its own queue (it is its only member currently) in SAVEQ, removes itself from the multi-processing queue, and joins C's queue. C continues to execute its initial statement.

Fig. 3: B has also made a request to C during the execution of its initial statement.

Fig. 4: C has finished its initial statement. It removes itself from its own queue and advances its NOWSERVING pointer around the queue to handle A's request.

Fig 1: running **A**, **B**, **C**.

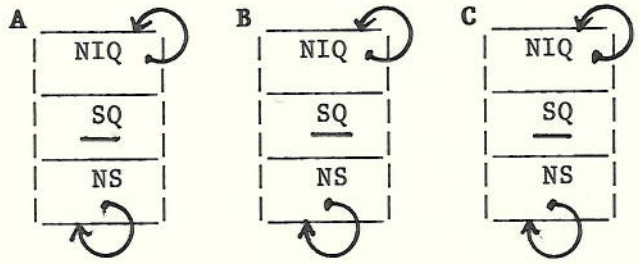


Fig. 2: running **B**, **C**.

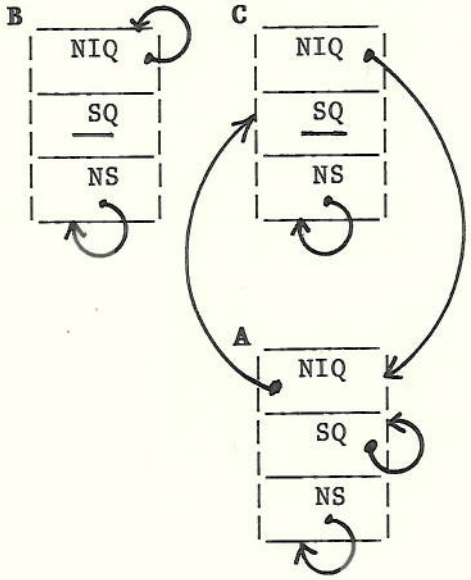


Fig. 3: running **C**.

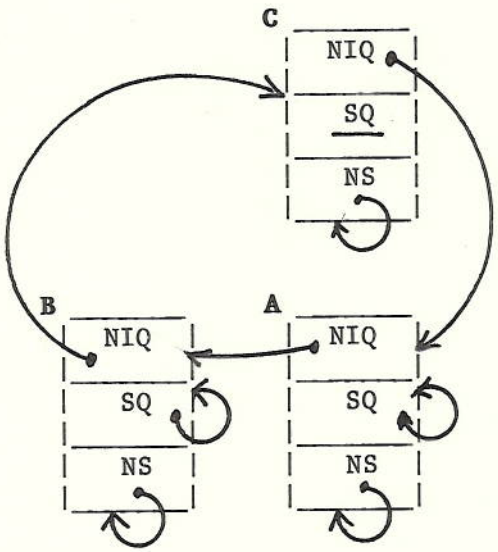


Fig. 4: running **C**.

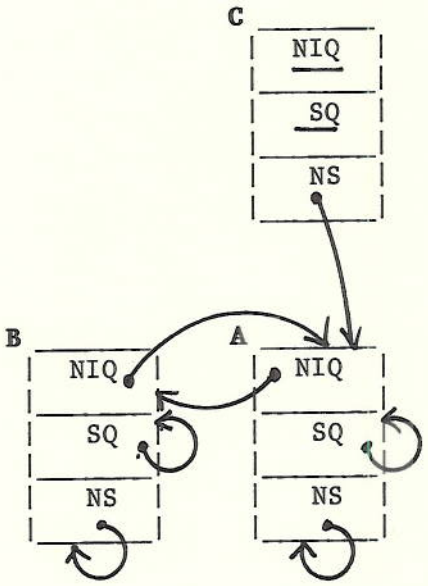


Fig. 5: running **A**, **C**.

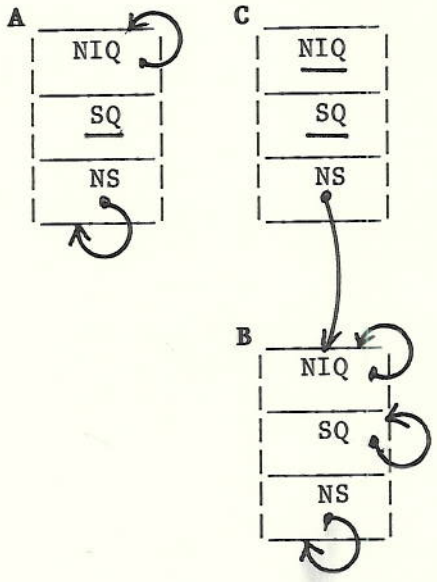


Fig. 5: C has completed A's request. C has removed A from C's queue, returned A to the multi-processing queue, restored A's saved queue, and advanced its own NOWSERVING pointer to handle B's request. With the answer to its request, A can resume execution of its initial statement.

Fig. 6: C, in handling B's request, had to make a request to A. (Fortunately this did not occur while A was in C's queue -- C and A would have waited for each other's answers forever!) C has joined A's queue, but A continues with its initial statement.

Fig. 7: A has finished its initial statement, removed itself from its own queue, and advanced its NOWSERVING pointer to handle C's request (made on B's behalf).

Fig. 8: A has finished C's request. A has removed C from A's queue, restored C to the multi-processing queue, and, since its queue is empty, awaits further requests. (A's initial statement is completed.) C continues to work on B's request.

Fig. 9: C has completed B's request, removed B from C's queue, and returned B to the multi-processing queue. C awaits further requests. B, with the answer to its request, continues with its initial statement.

Fig. 10: B completes its initial statement and removes itself from its own queue. All processes are waiting for requests, so the program has practically terminated, although theoretically the processes continue to make themselves available to answer requests forever.

Fig. 6: running **A**.

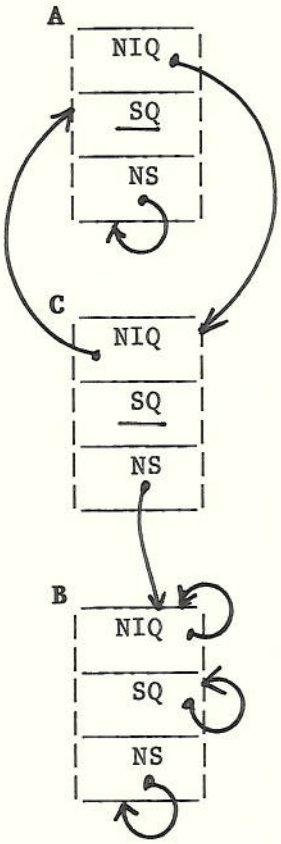


Fig. 7: running **A**.

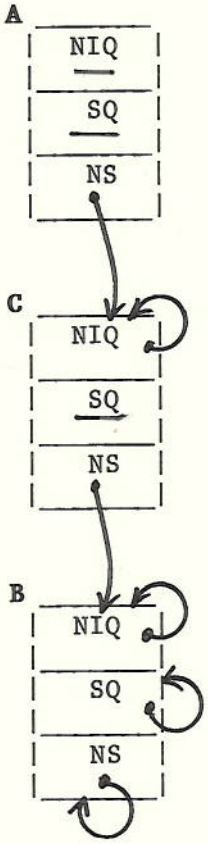


Fig. 8: running **A**, **C**.

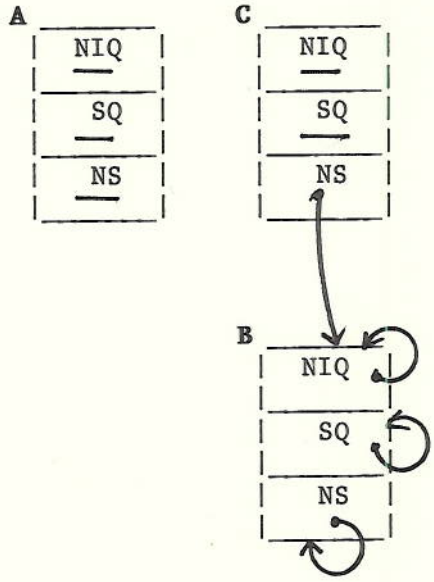


Fig. 9: running **A**, **B**, **C**.

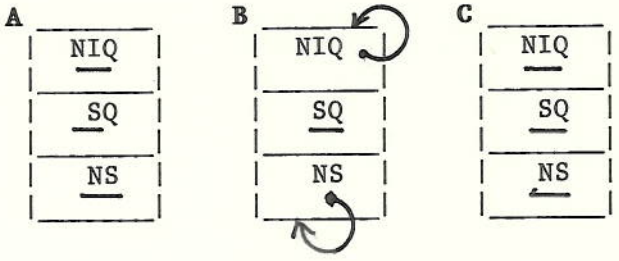


Fig. 10: running **A**, **B**, **C**.

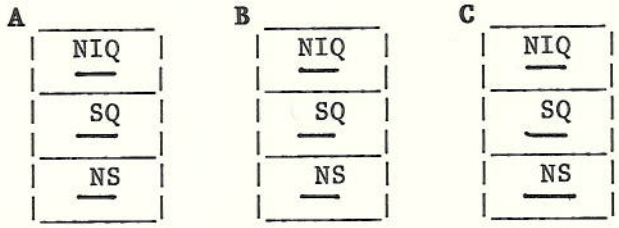
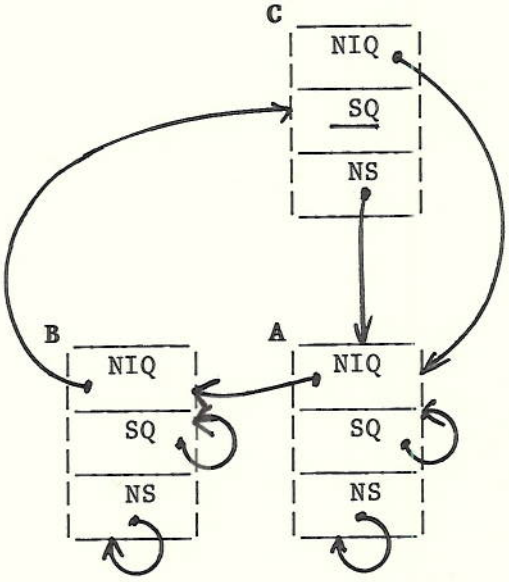


Fig. 11: running **C**.



Guarded Commands and Regions

Other important components of the Distributed Processes model are the guarded command and guarded region constructs -- IF, DO, WHEN, and CYCLE. All of these involve a group of predicates paired with statements. When a guarded construct is evaluated, the predicates are evaluated and a statement corresponding to one of the true predicates is nondeterministically chosen for evaluation. In this implementation, the choice is made by evaluating the predicates in the order they appear until a true one is found and then evaluating the corresponding statement, as with LISP's COND. In fact, the syntax used is identical to that of COND, and the atom COND is used instead of IF since IF has another meaning in standard SCHEME.

The guarded constructs can be divided into repetitive (DO, CYCLE) versus non-repetitive (COND, WHEN) and waiting (WHEN, CYCLE) versus non-waiting (COND, DO). Repetition might easily be implemented by any of several mechanisms, including macros with CATCH, and will not be described further here.

The evaluation of the waiting contructs (WHEN and CYCLE) differs from that of their non-waiting counterparts when none of the predicates is true. For COND, this is an error. For DO, this is the signal to terminate repetition. But WHEN and CYCLE wait until one (or more) of the predicates becomes true. The request currently being handled by the process relinquishes all control of the process. The process's NOWSERVING pointer (see above) is advanced to the next process in the

request queue, but the waiting request remains in the queue.

Consider fig. 3 above. If C's initial statement contained a WHEN construct, none of whose predicates were currently true, then the situation of fig. 11 would be the result.

When the NOWSERVING pointer was advanced to C again, either upon completion of the requests of A and B or upon encounter of guarded regions in those processes' requests, the predicates of C's guarded region would be re-evaluated. The entire sequence would be repeated until at least one of the predicates was found to be true. At this point, C could continue evaluation with the corresponding statement.

III. USER'S GUIDE

SCHEME and LISP primitives:

CAR, CDR, CONS, NULL, = (EQ), LIST, MEMBER

REVERSE, LENGTH, APPEND (*APPEND), REMOVE

ADD1, SUB1, PLUS (*PLUS), TIMES (*TIMES), < (*LESS), > (*GREAT)

NOT, AND (*AND), OR (*OR)

PRINT, ASET, PROGN

SCHEME magic forms:

(IF <predicate> <exp> <exp>) {2-branch only}

(QUOTE <exp>) or '<exp>

LAMBDA, CATCH, LABELS

(MACRO (<atom>) <exp>)

Distributed Processes primitives:

(PSET <vble-name-exp> <exp>)

(CALL <process-name-exp> <function-name-exp> <arg>*)

(RUN)

Distributed Processes magic forms:

(GAMMA <param-list> <common-function-deflist> <initial-exp>)

(COND <option>*)

(DO <option>*)

(WHEN <option>*) where

(CYCLE <option>*) <option> ::= (<guard> <exp>)

Miscellaneous:

(BIND <closure-exp> <vble-name-exp> <exp>)
        returns a new closure produced by adding a new binding to the
        closure which is the value of <closure-exp>. This new binding
        binds the variable name found by evaluating <vble-name-exp> to
        the value of <exp>. If the newly bound variable is a parameter
        of the closure, it is removed from the parameter list of the
        new closure (partial application of the closure). E.g.:
        (ASET 'SUB4 (BIND (LAMBDA (X Y) (PLUS X Y)) 'Y -4))

(EVAL <exp>)
        evaluates the value of <exp> in the current environment.
        An environment may NOT be specified.

(DSKIN <filename-exp>)
        reads and evaluates one expression from the file specified
        by the value of <filename-exp>, e.g. (DSKIN 'FOO.BAZ).

(DSKOUT <filename-exp> <vble-name-exp>)
        saves the value of the value of <vble-name-exp> on the file
        specified by the value of <filename-exp>, e.g.
        (DSKOUT 'XVAL.FIL 'X). The output is in a DSKIN-readable form.

(ECHO <filename-exp>)
        causes all following terminal I/O to be echoed on the file
        specified by the value of <filename-exp>. ECHOs may be nested
        to an arbitrary depth -- if you are not in a hurry.

(NOECHO)
        cancels the most recent ECHO and closes the corresponding file.

(ARRAY <dimension-exp>*)
        returns an array function for an array with the dimensions
        specified by the values of the <dimension-exp>'s.

(STORE (<array-vble> <subscript>*) <exp>)
        stores the value of <exp> in the array element specified by the
        <array-vble> and the <subscript-exp>'s.

(ACCESS (<array-vble> <subscript>*))
        returns the value of the array element specified, e.g.:
                #(ASET 'X (ARRAY 2 2))
                        (ARRAY 2 2)
                #(STORE (X 1 2) 7)
                        7
                #(ACCESS (X 1 2))
                        7

SCHEME EDITOR

The scheme editor takes an expr as input. It creates the list representation for this expr and edits this list. When OK is typed it returns the parsed value of the expression.

Editing of closures is allowed. When the editor is entered, the closure is "unclosed" (e.g. a BETA-closure is made into a corresponding LAMBDA expression). At the end of an editing session the expression is parsed and then reclosed with the same environment with which it entered.

This editor works much like the UCI LISP editor. To edit something, type (ASET 'E (EDIT E)) where E is the expr you wish to edit.

Commands:

note:
n = length of cursor expression
(1 <= i<= n) unless otherwise noted.

(i e1 e2 ... eN)	replace ith expression with e1 e2 ... eN
(i)	delete ith expression
(A i e1 e2 ... eN)	insert e1 e2 ... eN after expr. i (0 <= i <= n)
(SW i j)	switch expressions i and j
(BI i j)	place parens around expr. i to expr. j
(BO i)	remove parens from around expr. i
(P i)	printlev to a level of i
P	printlev cursor expressionto level of 2
PP	print cursor expression completely.
UNDO	undo last command
i	focus cursor on ith expression
0	focus cursor up one level
^	focus cursor on top-level expression
OK	to quit

IV. CONCLUSION

We have implemented the essence of the Distributed Processes model of concurrency within an expression-oriented system. Distributed Processes is not only compatible with SCHEME but is also more powerful and general in this form. This power and generality may not be advantageous in some distributed computing environments, however, where restrictions are used to ensure correct execution of concurrent programs. Some of the interesting and useful modifications to Distributed Processes made possible by the use of SCHEME as a base are:

1. Common function invocations are more flexible than Brinch Hansen's common procedure calls. Instead of requiring simple identifiers, we allow expressions to specify the process and the function name in a call. The invocation itself is an expression with a useful value and can therefore appear in many more contexts than Brinch Hansen's CALL statement. Lack of output parameters presents no difficulties in a list-oriented language such as SCHEME.

2. Guarded commands and regions were easily extended to return a useful value -- that of the expression selected for evaluation.

3. Distributed Processes allows arrays of identical processes. Process prototypes with parameters (GAMMA expressions) allow creation of several similar processes. lacks this facility.

4. Processes need not have a name, even if they are to be CALLed by other processes (see example 3). References to processes may be passed as arguments to external requests, or returned as values of external requests. This may be useful, for example, to allow process P to inform process Q that a certain situation exists in process R by passing Q a reference to R.

5. Function closures may be passed into or returned by common functions. "Outward funargs", closures passed out of the common function, may be used to allow quick communication where mutual exclusion is not essential or is guaranteed in some other way (see example 5). "Inward funargs" may be used to enforce certain scheduling constraints (example 3).

6. It is also possible to pass continuations into and out of common functions. This "C-link swapping" is hard to justify in a distributed computing system since it is probably more dangerous than useful.

The simplicity and elegance of SCHEME as well as the exceptional modularity made possible by SIMULA's CLASS facilities permitted the development of an easily extensible interpreter for SCHEME with minimal programming effort. Further, with facilities for process creation and quasi-parallel execution this system provides an excellent basis for experiments with other models of process communication, e.g. Hoare's CSP [17,18], Feldman's PLITS [10], Ada [19,24], etc.

ACKNOWLEDGEMENTS

We wish to express our appreciation to Professor Daniel P. Friedman for his support and guidance in supervising our project. We would also like to thank IUPUI Computing Services and the Wrubel Computing Center for making available the necessary computing resources.

V. BIBLIOGRAPHY

1. Allen, J., The Anatomy of LISP, McGraw-Hill, New York, 1978.

2. Brinch Hansen, P., "Structured Multi-programming," CACM 15, 7 (July, 1972), 574-578.

3. Brinch Hansen, P., "Concurrent Programming Concepts," Computer Surveys 5, 4 (December, 1973), 223-245.

4. Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering 1 (June, 1975), 199-207.

5. Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1977.

6. Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," CACM 21, 11 (November, 1978), 934-941.

7. Dijkstra, E.W., "Co-operating Sequential Processes," in Programming Languages ed. F. Genuys, Academic Press, New York, 1968, 43-112.

8. Dijkstra, E.W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," CACM 18, 8 (August, 1975), 453-457.

9. Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.

10. Feldman, J.A., "High Level Programming for Distributed Computing," CACM 22, 6 (June, 1979), 353-367.

11. Filman, R.E. and Friedman, D.P., Languages, Models and Heuristics for Coordinated Computing, book not yet completed.

12. Friedman, D., ALONZO interpreter, unpublished class notes.

13. Friedman, D. and Wise, D., "An Approach to Fair Applicative Multiprogramming," in Semantics of Concurrent Computation, ed G. Kahn, Lecture Notes in Computer Science, VOL. 70, Springer, Berlin, 1979, 203-226.

14. Friedman, D. and Wise, D., "An Indeterminate Constructor for Applicative Programming," Conference Record of the 7th Annual Symposium on Principles of Programming Languages, 1980.

15. Griswold, R.E., Poage, J.F., and Polonsky, I.P., The SNOBOL4 Programming Language, 2nd edition, Prentice-Hall, Englewood Cliffs, N.J., 1971, p. 96.

16. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM 17, 10 (October, 1976), 549-557.

17. Hoare, C.A.R., "Communicating Sequential Processes," CACM 21, 8 (August, 1978), 666-677.

18. Hoare, C.A.R., "A Model for Communicating Sequential Processes," Corrected proof of working paper, (July, 1979).

19. Ichbiah, J.D., et al., "Rationale for the Design of the ADA Programming Language," Sigplan Notices 14, 6, (June, 1979).

20. Milne, G., and Milner, R., "Concurrent Processes and Their Syntax," JACM 26, 2 (April, 1979), 302-321.

21. Steele, G. and Sussman, G., "The Revised Report on SCHEME, a Dialect of LISP," AI Memo 452 (January, 1978), MIT.

22. Stoy, J.E., "The Lambda-Calculus," in Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, The MIT Press, Cambridge, Massachusetts, and London, England, 1977, 52-77.

23. Wand, M., "Continuation-Based Multiprocessing," Conference Record of the 1980 LISP Conference, 19-28.

24. Wegner, P., "Multitasking," in Programming with ADA, an Introduction by Means of Graduated Examples, Prentice-Hall, Englewood Cliffs, New Jersey, 1980, 169-201.

APPENDIX

The code is divided into eleven separately compiled modules. DEC-10 SIMULA provides separate compilation of classes and procedures. By concatenating classes we were able to have ten "module classes", each containing a share of the classes and procedures which would normally be defined in one large file. The last file is the main program -- a block prefixed by the tenth module class and hence by all ten module classes. The structure of the files is shown below along with a short explanation of each module.

There is no explicit facility for declaring forward references in DEC-10 SIMULA; however, within this file structure it is easy to delay definition of a procedure by declaring it as a VIRTUAL attribute and defining it further down the chain of modules.

Summary of Source Files:

IO.SIM	CLASS io
BASIS.SIM	io CLASS basis
ED.SIM	basis CLASS ed
MAGIC.SIM	ed CLASS magic
PRIM1.SIM	magic CLASS prim1
PRIM2.SIM	prim1 CLASS prim2
DPBASIS.SIM	prim2 CLASS dpbasis
DPMAGIC.SIM	dpbasis CLASS dpmagic
DPPRIM.SIM	dpmagic CLASS dpprim
PARSER.SIM	dpprim CLASS parser
SCHEME.SIM	parser BEGIN ... END

Brief Description of File Contents:

IO contains code for input and output, including the lexical scanner.

BASIS contains definitions of the most basic data types -- environments, expressions, atoms, lists, and instances.

ED contains the code of the structure editor.

MAGIC contains definitions of standard SCHEME magic forms.

PRIM1 and PRIM2 contain standard SCHEME and LISP primitives.

DPBASIS contains definitions of the process and distributor classes necessary for multi-processing.

DPMAGIC contains definitions of Distributed Processes magic forms -- GAMMA, COND, WHEN, DO, and CYCLE.

DPPRIM contains code for the Distributed Processes primitives -- RUN, PSET, and CALL.

PARSER contains both the parser and the procedure to check for ^G interrupt used by the Distributor.

SCHEME contains the main program, a block prefixed by class PARSER. It initializes the primitive environment and starts the Distributor.

```
OPTIONS(/E);
CLASS io;
BEGIN
    REF(input) inf;  REF(noecho) outf;

    CLASS input(file); REF(Infile) file;
    BEGIN
        TEXT 1,buf,prompt;

        TEXT PROCEDURE readtoken;
        BEGIN
            CHARACTER window;
            INTEGER startpos;

            CHARACTER PROCEDURE Getchar;
            Getchar := window := IF 1.More THEN 1.Getchar ELSE Char(0);

            BOOLEAN PROCEDURE blank(c); CHARACTER c;
            blank := c = ' ' OR c = Char(9);

            BOOLEAN PROCEDURE delim(c); CHARACTER c;
            delim := blank(c) OR c = Char(0) OR c = '(' OR c = ')';

            WHILE blank(Getchar) DO;
            WHILE window = Char(0) DO
            BEGIN
                outf.Outtext(prompt);
                outf.Breakoutimage;
                file.Inimage;
                1 :- Blanks(buf.Length + file.Image.Length);
                1.Sub(1,buf.Length) := buf;
                1.Sub(buf.Length + 1,file.Image.length) := file.Image;
                buf :- NOTEXT;
                outf.readecho(1);
                WHILE blank(Getchar) DO
            END;

            startpos := 1.Pos -1;
            IF window = Char(39) THEN readtoken :- Copy("'")
            ELSE IF window = ')' THEN readtoken :- Copy(")")
            ELSE IF window = '(' THEN readtoken :- Copy("(")
            ELSE
            BEGIN
                IF Digit(window) THEN
                BEGIN WHILE Digit(Getchar) DO END
                ELSE WHILE NOT delim(Getchar) DO;
                1.Setpos(1.Pos - 1);
                readtoken :- Copy(1.Sub(startpos,1.Pos - startpos))
            END
        END;

        prompt :- Copy("#")

    END ***** of class input *****;
```

```
    CLASS noecho(file);  REF(Outfile) file;
    VIRTUAL:
    PROCEDURE readecho;          PROCEDURE Outtext;
    PROCEDURE Outint;            PROCEDURE Outimage;
    PROCEDURE Breakoutimage;
    BEGIN
        PROCEDURE readecho(s); TEXT s;;
        PROCEDURE Outtext(s); VALUE s; TEXT s; file.Outtext(s);
        PROCEDURE Outint(i,n); INTEGER i,n; file.Outint(i,n);
        PROCEDURE Outimage; file.Outimage;
        PROCEDURE Breakoutimage; file.Breakoutimage;

    END ***** of class noecho *****;


    noecho CLASS echo(oldoutf); REF(noecho) oldoutf;
    BEGIN
        PROCEDURE readecho(s); TEXT s;
        BEGIN
            file.Outtext(s);
            file.Outimage;
            oldoutf.readecho(s)
        END;

        PROCEDURE Outtext(s); VALUE s; TEXT s;
        BEGIN
            oldoutf.Outtext(s);
            file.Outtext(s)
        END;

        PROCEDURE Outint(i,n); INTEGER i,n;
        BEGIN
            oldoutf.Outint(i,n);
            file.Outint(i,n)
        END;

        PROCEDURE Outimage;
        BEGIN
            oldoutf.Outimage;
            file.Outimage
        END;

        PROCEDURE Breakoutimage;
        BEGIN
            oldoutf.Breakoutimage;
            file.Breakoutimage
        END;

    END ***** of class echo *****;

    inf :- NEW input(Sysin);
    outf :- NEW noecho(Sysout);
    linesperpage(-1)
END ---------- of class io ----------;
```

```
OPTIONS(/e);
EXTERNAL CLASS io;
io CLASS basis;
VIRTUAL: REF(expr) PROCEDURE parse;
BEGIN
    REF(list) primvars,primvals;  REF(environ) primenv;
    REF(proc) running; REF(litconst) t,f; REF(nils) nil;

    CLASS expr;
    VIRTUAL:
    BOOLEAN PROCEDURE eecue;     REF(expr) PROCEDURE getrep;
    REF(expr) PROCEDURE first;   REF(list) PROCEDURE rest;
    PROCEDURE print;             REF(instance) PROCEDURE instantiate;
    BEGIN
        PROCEDURE print; getrep.print;

        REF(expr) PROCEDURE getrep; getrep :- THIS expr;

        REF(expr) PROCEDURE first;
        BEGIN
            outf.Outtext("FIRST OF EXPR  ");
            print;
            outf.Outimage
        END;

        REF(list) PROCEDURE rest;
        BEGIN
            outf.Outtext("REST OF EXPR  ");
            print;
            outf.Outimage
        END;

        BOOLEAN PROCEDURE eecue(Exp); REF(expr) Exp;
        eecue := THIS expr == Exp;


    END ***** of class expr *****;


    CLASS instance(Exp,env,ret);
    REF(expr) Exp; REF(environ) env; REF(instance) ret;
    BEGIN
        1:      Detach;
        INNER;
        GOTO 1
    END ***** of class instance *****;
```

```
instance CLASS forminst;
running.control :-
Exp.first.instantiate(env,NEW formlinst(Exp,env,ret));


instance CLASS formlinst;
running.control :- running.box.instantiate(env,ret,Exp.rest,nil);


list CLASS nils;
BEGIN
    PROCEDURE print; outf.Outtext("()");

    PROCEDURE printrest; outf.Outtext(")");

    REF(instance) PROCEDURE instantiate(env,ret);
    REF(environ) env; REF(instance) ret;
    BEGIN
        outf.Outtext("ATTEMPT TO EVALUATE ()");
        outf.Outimage
    END;

    REF(expr) PROCEDURE first;
    BEGIN
        outf.Outtext("FIRST OF ()");
        outf.Outimage
    END;

    REF(list) PROCEDURE rest;
    BEGIN
        outf.Outtext("REST OF ()");
        outf.Outimage
    END;

    REF(expr) PROCEDURE getrep; getrep :- THIS nils;

    BOOLEAN PROCEDURE eecue(Exp); REF(expr) Exp;
    eecue := Exp IS nils;

    REF(list) PROCEDURE reverse; reverse :- nil;

    REF(list) PROCEDURE revl(last); REF(list) last; revl :- last;

    REF(list) PROCEDURE remove(s); REF(expr) s; remove :- nil;

    REF(list) PROCEDURE append(l); REF(list) l; append :- l;

    BOOLEAN PROCEDURE member(s); REF(expr) s;;

    INTEGER PROCEDURE Length; Length := 0;

END ***** of list class nils *****;
```

```
expr CLASS list(ff,rr); REF(expr) ff; REF(list) rr;
VIRTUAL:                              PROCEDURE printrest;
REF(list) PROCEDURE reverse;          REF(list) PROCEDURE revl;
REF(list) PROCEDURE remove;           REF(list) PROCEDURE append;
BOOLEAN PROCEDURE member;             INTEGER PROCEDURE Length;
BEGIN
    PROCEDURE print;
    IF ff.eecue(NEW vble("QUOTE")) THEN
    BEGIN
        outf.Outtext("'");
        rr.first.print
    END
    ELSE
    BEGIN
        outf.Outtext("(");
        ff.print;
        rr.printrest
    END;

    PROCEDURE printrest;
    BEGIN
        outf.Outtext(" ");
        ff.print;
        rr.printrest
    END;

    REF(instance) PROCEDURE instantiate(env,ret);
    REF(environ) env; REF(instance) ret;
    instantiate :- NEW forminst(THIS list,env,ret);

    REF(expr) PROCEDURE first; first :- ff;

    REF(list) PROCEDURE rest; rest :- rr;

    BOOLEAN PROCEDURE eecue(Exp); REF(expr) Exp;
    IF Exp IS list THEN
    eecue := ff.eecue(Exp.first) AND rr.eecue(Exp.rest);

    REF(list) PROCEDURE reverse; reverse :- revl(nil);

    REF(list) PROCEDURE revl(last); REF(list) last;
    revl :- rr.revl(NEW list(ff,last));

    REF(list) PROCEDURE remove(s); REF(expr) s;
    remove :- IF ff.eecue(s) THEN rr.remove(s)
    ELSE NEW list(ff,rr.remove(s));

    REF(list) PROCEDURE append(l); REF(list) l;
    append :- NEW list(ff,rr.append(l));

    BOOLEAN PROCEDURE member(s); REF(expr) s;
    member := IF ff.eecue(s) THEN TRUE ELSE rr.member(s);

    INTEGER PROCEDURE Length; Length := 1 + rr.Length;
END ***** of expr class list *****;
```

```
atom CLASS const;
BEGIN
    REF(instance) PROCEDURE instantiate(env,ret);
    REF(environ) env; REF(instance) ret;
    instantiate :- NEW constinst(THIS const,env,ret);
END ***** of atom class const *****;


instance CLASS constinst;
BEGIN
    running.box :- Exp;
    running.control :- ret;
END ***** of instance class constinst *****;


const CLASS litconst(pname); VALUE pname; TEXT pname;
BEGIN
    PROCEDURE print; outf.Outtext(pname);

    BOOLEAN PROCEDURE eecue(Exp); REF(expr) Exp;
    IF Exp IS litconst THEN
    eecue := pname = Exp QUA litconst.pname;

END ***** of const class litconst *****;


const CLASS number(num); INTEGER num;
BEGIN
    PROCEDURE print; outf.Outint(num,len);

    INTEGER PROCEDURE len;
    len := IF num = 0 THEN 1 ELSE Entier(Ln(num) / Ln(10)) + 1;

    BOOLEAN PROCEDURE eecue(Exp); REF(expr) Exp;
    IF Exp IS number THEN eecue := num = Exp QUA number.num;

END ***** of const class number *****;
```

```
expr CLASS atom;;


atom CLASS vble(pname); VALUE pname; TEXT pname;
BEGIN
    PROCEDURE print; outf.Outtext(pname);

    REF(instance) PROCEDURE instantiate(env,ret);
    REF(environ) env; REF(instance) ret;
    instantiate :- NEW vbleinst(THIS vble,env,ret);

    BOOLEAN PROCEDURE eecue(Exp); REF(expr) Exp;
    IF Exp IS vble THEN eecue := pname = Exp QUA vble.pname;

END ***** of atom class vble *****;


instance CLASS vbleinst;
BEGIN
    running.box :- env.atval(Exp QUA vble);
    running.control :- ret
END ***** of instance class vbleinst *****;
```

```
expr CLASS function;;


instance CLASS funinst(un1,ev1); REF(list) un1,ev1;
IF un1 =/= nil THEN
BEGIN
    running.control :- un1.first.instantiate(env,
    Exp.instantiate(env,ret,un1.rest,NEW list(running.box,ev1)));
    Detach
END
ELSE ev1 :- NEW list(running.box,ev1).reverse.rest;


expr CLASS magexp;;


magexp CLASS closable;
VIRTUAL: REF(closure) PROCEDURE Close;
BEGIN
    REF(instance) PROCEDURE instantiate(env,ret);
    REF(environ) env; REF(instance) ret;
    instantiate :- NEW closableinst(THIS closable,env,ret);
END ***** of magexp class closable *****;


instance class closableinst;
BEGIN
    running.box :- exp QUA closable.Close(env);
    running.control :- ret
END;


expr CLASS closure(envsav); REF(environ) envsav;
VIRTUAL:
REF(closure) PROCEDURE bind;    REF(closable) PROCEDURE unclose;;


expr CLASS proc(globenv); REF(environ) globenv;
BEGIN
    BOOLEAN done;
    REF(expr) box;
    REF(instance) control;
END ***** of expr class proc *****;
```

```
CLASS environ(nextenv,varrib,valrib);
REF(environ) nextenv; REF(list) varrib,valrib;
BEGIN

    REF(environ) PROCEDURE pairlis(vars,vals);
    REF(list) vars,vals;
    pairlis :- NEW environ(THIS environ,vars,vals);

    REF(environ) PROCEDURE bind1(var,val);
    REF(atom) var; REF(expr) val;
    bind1 :- NEW environ(THIS environ,
    NEW list(var,nil),NEW list(val,nil));

    REF(environ) PROCEDURE adddefs(defs); REF(list) defs;
    BEGIN
        REF(environ) env;
        adddefs :- env :- NEW environ(THIS environ,nil,nil);
        INSPECT env DO
        WHILE defs =/= nil DO
        BEGIN
            varrib :- NEW list(defs.first.first,varrib);
            valrib :- NEW list(
            defs.first.rest.first QUA closable.Close(env),valrib);
            defs :- defs.rest
        END
    END;


    REF(list) PROCEDURE assoc(var); REF(vble) var;
    BEGIN
        REF(list) vars,vals;

        vars :- varrib;
        vals :- valrib;
        WHILE NOT(vars==nil OR vals==nil OR var.eecue(vars.ff)) DO
        BEGIN
            vars :- vars.rr;
            vals :- vals.rr
        END;
        assoc :- IF vars == nil THEN nil ELSE vals
    END;
```

```
    REF(expr) PROCEDURE atval(var); REF(vble) var;
    BEGIN
        REF(list) riblet;
        REF(environ) env;

        env :- THIS environ;
        riblet :- nil;
        WHILE env =/= NONE AND riblet == nil DO
        BEGIN
            riblet :- env.assoc(var);
            IF riblet == nil THEN env :- env.nextenv
        END;
        IF riblet == nil THEN
        BEGIN
            outf.Outtext("ATVAL - unbound atom  ");
            var.print;
            outf.Outimage
        END
        ELSE atval :- riblet.ff
    END;


    PROCEDURE atset(var,val); REF(vble) var; REF(expr) val;
    BEGIN
        REF(list) riblet;
        REF(environ) env;
        BOOLEAN finished;

        riblet :- nil;
        env :- THIS environ;
        WHILE NOT finished DO
        BEGIN
            riblet :- env.assoc(var);
            finished := riblet =/= nil OR env.nextenv == primenv;
            IF NOT finished THEN env :- env.nextenv
        END;
        IF riblet =/= nil THEN riblet.ff :- val
        ELSE
        INSPECT env DO
        BEGIN
            varrib :- NEW list(var,varrib);
            valrib :- NEW list(val,valrib)
        END
    END;

END ***** of class environ *****;
```

```
    REF(expr) PROCEDURE read; read :- read1(inf.readtoken);

    BOOLEAN PROCEDURE checkint(s); VALUE s; TEXT s;
    checkint := Digit(s.Getchar);

    REF(expr) PROCEDURE read1(s); TEXT s;
    INSPECT inf DO
    read1 :- IF s = ")" THEN read1(readtoken)
    ELSE IF s = "(" THEN readb(readtoken)
    ELSE IF s = "'" THEN list2(NEW vble("QUOTE"),read1(readtoken),nil)
    ELSE IF checkint(s) THEN NEW number(s.Getint)
    ELSE IF s = "TRUE" THEN t ELSE IF s = "FALSE" THEN f
    ELSE NEW vble(s);

    REF(list) PROCEDURE readb(s); TEXT s;
    readb :- IF s = ")" THEN nil
    ELSE NEW list(read1(s),readb(inf.readtoken));


    REF(list) PROCEDURE list2(e0,e1,rest);
    REF(expr) e0,e1; REF(list) rest;
    list2 :- NEW list(e0,NEW list(e1,rest));

    REF(list) PROCEDURE list3(e0,e1,e2,rest);
    REF(expr) e0,e1,e2; REF(list) rest;
    list3 :- NEW list(e0,list2(e1,e2,rest));

    REF(list) PROCEDURE list4(e0,e1,e2,e3,rest);
    REF(expr) e0,e1,e2,e3; REF(list) rest;
    list4 :- list2(e0,e1,list2(e2,e3,rest));

    REF(list) PROCEDURE list8(e0,e1,e2,e3,e4,e5,e6,e7,rest);
    REF(expr) e0,e1,e2,e3,e4,e5,e6,e7; REF(list) rest;
    list8 :- list4(e0,e1,e2,e3,list4(e4,e5,e6,e7,rest));

    nil :- NEW nils(NONE,NONE);
    t :- NEW litconst("TRUE");
    f :- NEW litconst("FALSE");
    primvars :- nil;
    primvals :- nil;


END ----------------of class basis-------------------;
```

```
OPTIONS(/E);
EXTERNAL CLASS io,basis;

basis CLASS ed;
BEGIN
    function CLASS edit;
    BEGIN
        PROCEDURE print; outf.Outtext("edit");

        REF(instance) PROCEDURE instantiate(env,ret,unl,evl);
        REF(environ) env; REF(instance) ret; REF(list) unl,evl;
        instantiate :- NEW editinst(THIS edit,env,ret,unl,evl);
    END;


    funinst CLASS editinst;
    BEGIN
        INSPECT evl.first WHEN closure DO
        running.box :- editproc(unclose) QUA closable.Close(envsav)
        OTHERWISE running.box :- editproc(evl.first);
        running.control :- ret
    END;
```