A MultiProcessor Simulator

Abhiram G. Ranade

YALEU/DCS/RR#452
January 1986

This work was supported in part by the National Science Foundation under grant number DCR-8106181, and in part by the Office of Naval Research under grant number N-00014-84-K-0043.

Table of Contents

1. Introduction

AMPS, A MultiProcessor Simulator, is a simulation system for modelling the behavior of a distributed computer program running on a distributed architecture. It consists of an architecture description language, a distributed programming language, and facilities to monitor the simulation and estimate timing.

In AMPS, a distributed architecture is modelled as a set of sequential or almost sequential processes. The processes can communicate with one another over different kinds of channels, compete for resources, and of course, perform independent computation. Architectures can be described hierarchically or iteratively, and these descriptions can be parameterized. The programs associated with the processes are written in the AMPS distributed programming language. This language contains primitives for communication, resource contention, dynamic process creation etc. A kernel language is also provided. Using this it is possible to create new primitives easily. This is especially useful for communication, where radically different primitives have been proposed. AMPS also allows the user to assign delays for computation or communication. These delays can be specified statically or dynamically.

AMPS is implemented by embeding its constructs in T [7], a dialect of Lisp developed at Yale. The architecture description language has been adapted from the Yale Digital Simulator [1].

1.1 Overview of the Report

The architecture description language is discussed in detail in chapters 2 and 3. The programming language is described in chapters 4 and 5. Chapter 6 describes how the communication/computation delays may be simulated. Chapter 7 discusses interactive simulation control. The implementation is briefly described in chapter 8. This chapter also describes the facilities provided for developing new communication primitives.

The appendices contain detailed definitions of three different architectures, and non trivial programs that run on them.

2. Architecture Description

This chapter describes the main construct of the AMPS architecture description language. The vector package, which is also useful in architecture description, is described in chapter 3.

In AMPS, a distributed computer system is modelled as a set of sequential or almost sequential processes. These processes can communicate with one another, compete for shared resources, and of course, perform independent computatation. Each process usually represents a single physical processing element; if it is necessary to simulate two or more processes residing on a single processor, it can be done by modelling the physical processor as a resource for which the processes must compete. Communication links and shared resources are modelled by objects called *ports*. The topology of an architecture is defined by allowing appropriate processes to share access to appropiate ports.

AMPS allows the user to define an architecture-type rather than a particular architecture. Once an architecture-type has been defined, it is possible to create several instances of the type. Architecture-type description is hierarchical. The elements of this hierarchy are called *boxes*. At the lowest level of the hierarchy, a box could be a single process, at the highest level, a box may encapsulate a complete architecture. A box may also be composed of *vectors* of sub-boxes. This allows architectures to be defined iteratively (e.g. systolic arrays) and recursively/hierarchically (e.g. tree connected processors). Box-type descriptions can be parameterized. The values of the parameters for a particular instance are specified at the time of its creation. Parameterization is useful in two ways. First, parameters can be used to specialize an instance in different ways, i.e. specify its size, shape, internal connection pattern etc. Secondly, parameters can be used to supply information about the environment to the instance being created. In particular, box-type description can be parameterized on the external communication ports. Thus, interconnection between boxes is modelled by specifying the same communication port as an actual parameter at creation of the boxes.

2.1 Boxes and Box-types

A box is the basic program structuring construct in AMPS. A box consists of a local environment with or without an associated program. A box that has a program associated with it is called a process. The environment of a box may contain other boxes or private data objects. A box may also contain procedures for structured manipulation of its environment. Thus, a box can be used to encapsulate a process, to group together other boxes, or to define a data structure that responds to specific operations.

A box-type is defined by parametrically describing a generic instance:

(DEF-BOX *name-of-box-type*   (*formal-parameter . . .*)
        section . . .)

Any T identifier can be used as *name-of-box-type*.

Figure 2-1 contains examples of box-type definitions. The following sections will use these for illustration.

```
(def-box adder (in1 in2 out delay)
    (locals
     (val1 nil)
     (val2 nil))
    (program
     (loop (do
              (fork
                ((receive in1 val1))
                ((receive in2 val2)))
              (wait delay)
              (send out (+ val1 val2))) ))
    (show ()                                    ;;; user-defined section
     (format t "~% Adder: ~a" self)
     (format t "~%  val1: ~a,  val2: ~a" val1 val2)))

(def-box cpu ()
 (locals
  (a (mailbox:new))
  (b (mailbox:new))
  (c (mailbox:new))
  ....)
 (boxes
  (alu (adder:new a b c 5))
  ....)
 ....)
```

Figure 2-1: Box-type Examples

A box-type definition may contain three predefined sections: LOCALS, BOXES and PROGRAM. These, respectively, contain information about local data structures, subboxes that the generic instance might have, and the program that is to be executed. All sections are optional.

In addition to predefined sections a box-type might have user defined sections. These sections describe how instances of the box-type would respond to user-defined or primitive T operations.

2.1.1 Box instantiation

The definition of a box-type creates an instantiation function:

name-of-box-type:NEW

To create a box the instantiation function of its box-type is called with actual parameters appropriate for the box-type definition. When called the instantiation function creates the data structures, instantiates subboxes (recursively) using the specified parameters, and then returns the box so created. For example the first statement in figure 2-2 would result in setting adder1 to an instance of the box-type adder, and it would have a delay of 8 units. The second statement would create a box of type cpu, and the variable cpu1 would be set to the box so created. Notice that boxes of type cpu contain subboxes of type adder. The creation of the box cpu1 would cause the subbox to be created as well. This box would have a delay of 5, as can be seen from the definition of box-type cpu.

```
;;; Assume a, b, c have been set appropriately
;;; For definitions of adder and cpu, see figure 2-1

(set adder1 (adder:new a b c 8))
(set cpu1   (cpu:new))
```

Figure 2-2: Box instantiation

2.2 Local objects: BOXES and LOCALS sections

The BOXES and LOCALS sections describe respectively the subboxes and other local objects that an instance of a box-type may have.

The LOCALS section has the form:

(LOCALS
 (*object-name object-definition*) . . .)

Each *object-definition* is a T expression which is evaluated to yield a value for the corresponding *object-name*.

The BOXES section has a similar form. In this case, however, each *object-definition* must return either an instance of a box-type or a vector consisting of such instances i.e. the definition would contain calls to appropriate box-type instantiation functions. As an example, consider the definition of box-type cpu in figure 2-1. This contains a local box called alu of box-type adder. It is also possible to have recursive box-type definitions. This is useful for defining structures like trees, as in figure 2-3.

```
;; The parent mailbox is used to communicate with the parent.  The lbox and
;; rbox with the left and right sons respectively.

(def-box tree-node (size parent)
  (locals
    (lbox (mailbox:new))
    (rbox (mailbox:new))
    (sizeby2 (div size 2)))
  (boxes
    (lson (if (> size 1)
              (tree-node:new sizeby2 lbox)
              nil))
    (rson (if (> size 1)
              (tree-node:new sizeby2 rbox)
              nil)) ))
```

Figure 2-3: Recursive Box-type Definition For A Tree

The information in the LOCALS and BOXES sections is used to create the box-type instantiation function. During instantiation the objects in the LOCALS section are created sequentially from top to bottom, followed by the objects in the BOXES section. The definitions in the BOXES section may refer to any objects in the LOCALS sections or subboxes defined earlier. Definitions in the LOCALS section may only refer to local objects defined earlier. Both, however, may refer to the formal parameters of the box-type definition.

2.2.1 Accessing Local Objects And Formal Parameters

Within the box-type definition, locals, boxes, and formal parameters may be accessed simply by using their names. The only requirement is that the appropriate datum be defined at the time it is accessed, as described in the preceding passage.

Outside the definition, *accessor* functions must be used. These are created as a result of defining a box-type. Each accessor function has the form:

name-of-box-type : *name-of-local-object-or-formal-parameter*

This function, when applied to an instance of the box-type, yields the appropriate local object or actual parameter. For example, after execution of the code in figure 2-2, the alu box local to cpu1 could be referenced by:

```
(cpu:alu cpu1)
```

The val1 field of this adder could be referenced as:

```
(adder:val1 (cpu:alu cpu1))
```

2.2.2 The variable *self*

A box-type definition defines a class of boxes by describing a generic box that is a representative of that class. In describing the operations that the box can handle, it is often useful to have a variable that is bound to the box itself. *Self* is such a variable. For this reason, avoid having another local object called *self* within a box-type definition. The result of setting the variable *self* within the definition is unpredictable.

For an example refer to the *show* section in box-type definition of adder in figure 2-1. When the user defined operation (cf. section 2.4) *show* is invoked on an instance, self would be bound to the instance, and thus the format statement would print the full hierarchical name of the instance.

2.3 The Program Section

The program section has the form:

```
(PROGRAM . statements)
```

Each *statement* is either an AMPS language statement or a T expression. The program section and AMPS language statements are discussed in chapters 4 and 5.

2.4 User-defined sections

The general form is:

```
([CONCURRENT] operation (operation-formal-parameter ...)
    statement ...)
```

If the keyword CONCURRENT is not present, *Operation* should be a T identifier, possibly the name of a predefined T operation. Each *statement* should be a T expression. The operation may be invoked on a *box* by:

```
(operation box operation-actual-parameter ...)
```

Upon execution of this the actual parameters are bound to the corresponding formals, and the T expressions are sequentially evaluated, and the value of the last expression is returned. As an example, suppose the code in figure 2-2 has been executed. Then, invoking

```
(show (cpu:alu cpu1))
```

would cause val1 and val2 in the alu subbox of cpu1 to be printed, along with its full hierarchical name.

If the keyword CONCURRENT is present, then *operation* must be a T identifier. Each *statement* must be an AMPS statement or a T expression. The operation may be invoked on a *box* by using the CALL[1] statement:

```
(CALL (operation box operation-actual-parameter ...)
      (result-var-list))
```

Result-var-list should be a list of settable T expressions. The expressions in the *Result-var-list* are used to receive values returned by the *operation*.

The CALL statement causes the actual parameters to be bound to the corresponding formal parameters of the *operation*. The *statements* are then sequentially executed. Values are returned to the caller using the RETURN statement.

[1]Refer to chapter 4 for descriptions of CALL and RETURN statements.

3. Vectors

AMPS provides facilities to create and manipulate vectors of objects. One or two dimensional vectors are directly supported. By creating a two dimensional vector of one dimensional vectors, the user can build a three dimensional vector. Vectors having more dimensions can be similarly constructed.

Vectors know how to print themselves, i.e. they have a method for handling the operation 'print'. Vectors print as

(V> *vector-element*...)

The elements of the vector are printed as whatever they are - calls on accessor functions, simple elements, vectors when vectors are nested, etc.

The following sections discuss primitives for creating and manipulating vectors.

3.1 Creating And Manipulating Vectors

```
(def-box proc()
    (locals
      (a   nil)
      (b   nil)
      (c   nil)) ....)

(def-box proc1 (side_length)
    (locals
      (a   (* side_length side_length))
      (b   nil)
      (c   nil)) ....)
```

Figure 3-1: Example Boxes

The following command descriptions use the box-type definitions in figure 3-1 to illustrate the use of the commands.

(V:NEW *Init Size*)
(V:NEW *Init Lower Upper*)

Creates a vector. *Init* is a function of no arguments. Each element of the new vector is initially set to the result of a call on *Init*. The first form creates a vector with indices in the range 0 to *Size*-1. The second form creates a vector with indices in the range *Lower* to *Upper*. The lower bound must be less than or equal to the upper bound. For example

```
(V:New proc:new  10)
(V:New (lambda () (proc1:new 5)) 3 12)
```

The second form creates a vector whose indices range from 3 to 12. Further each of the elements is of box-type proc1, and each has its side_length parameter set to 5.

(V:NEWSP *Init Size*)

Creates a vector. *Init* is a function of one argument. The i^{th} element of the new vector is initially set to the result of a call on (*Init* i). The vector has indices in the range 0 to *Size*-1. For example

```
(V:Newsp proc1:new  10)
(V:Newsp (lambda (i) (proc:new)) 10)
```

The first form creates a vector of 10 elements. Each element is of box-type proc1. The i^{th} element of the vector has its side_length parameter set to i. The second form just illustrates how a function of no arguments can be created by throwing away one argument. It is equivalent to:

```
(v:new proc:new 10)
```

(V:REF *Vector Index*)

Accesses the *Index*th element of *Vector*. For example

```
(set V1 (V:New (lambda () 0) 4))
(set (V:Ref V1 0) 10)
(V:Ref V1 0)  => 10
(V:Ref V1 3)  => 0
```

(V:LOWER *Vector*)

Returns the lowest legal index for *Vector*. For example

```
(set V1 (V:New proc:new 4))
(V:Lower V1) => 0
```

(V:UPPER *Vector*)

Returns the highest legal index for *Vector*. For example

```
(set V1 (V:New proc:new 13))
(V:Upper V1) => 12
```

(V:SIZE *Vector*)

Returns the number of elements in *Vector*. For example

```
(set V1 (V:New proc:new 3 12))
(V:Size V1) => 10
```

(V:MAP *Func* $Vector_1$ $Vector_2$ ... $Vector_n$)

Maps function *Func* over $Vector_1$ through $Vector_n$. *Func* should be a function of N arguments. All vectors should be of the same length. The result of V:Map is a vector of the results of applying *Func* to the elements of the vectors. For example

```
(set V1 (v:newsp proc1:new 4))
(V:Map proc1:side_length V1) => (V> 0 1 2 3)
(V:Map proc1:a        V1) => (V> 0 1 4 9)
```

(V:MAKE Arg_1 ... Arg_n)

Make a new vector with each Arg_i a separate element. The result vector is indexed from 0. For example

```
(V:Make 1 2 3 4) => (V> 1 2 3 4)
```

(V:CAT *Arg*$_1$ . . . *Arg*$_n$)

Splice *Arg*$_1$ through *Arg*$_n$ into a vector. Vector arguments are flattened by one nesting level. Scalars produce a single element of the result. The result vector is indexed from 0. For example

```
(set V1 (V:Cat 1 2 (V:Make 'A 'B 'C)))
   => (V> 1 2 A B C)
(V:Lower V1) => 0
(set V2 (V:Make 1 2))
(set V3 (V:Make 'A 'B V2))
   => (V> A B (V> 1 2))
(V:Cat 1 2 V3)
   => (V> 1 2 A B (V> 1 2))
```

(V:FLAT *Arg*$_1$ . . . *Arg*$_n$)

A recursive V:Cat. Splices its arguments into a single vector. Nonvectors are spliced in as single elements, vectors have V:Flat applied to the list of their members before splicing. For example

```
(set V2 (V:Make 1 2))
(set V3 (V:Make 'A 'B V2))
   => (V> A B (V> 1 2))
(V:Cat 1 2 V3)
   => (V> 1 2 A B (V> 1 2))
(V:Flat 1 2 V3)
   => (V> 1 2 A B 1 2)
```

(V:FROM-LIST *List*)

Make *List* into a vector. For example

```
(V:From-List '(A B C))  => (V> A B C)
```

(V:TO-LIST *Arg*$_1$ . . . *Arg*$_n$)

Returns a list of the elements of the argument vectors. For example

```
(set V1 (V:Make 'A 'B 'C))
(set V2 (V:Make 1 2))
(V:To-List V1 V2)  => (A B C 1 2)
```

(V:MEMQ *X* *Vector*)

If *X* is an element of *Vector* then its index in *Vector* is returned. If *X* is not an element of *Vector* then *nil* is returned. The T function eq? is used for comparison. For example

```
(set V1 (V:Make 'A 'B 'C))
(V:Memq 'A V1)  => 0
(V:Memq 'X V1)  => ()
```

(V:SUBRANGE *Vector Lower Upper*)

Create a new vector which is that part of *Vector* with indices from *Lower* to *Upper*. The created vector actually shares elements with Vector. Thus changes to either will affect the other. For example

```
(set V1 (V:Make 'A 'B 'C 'D 'E))
(set V2 (V:SubRange V1 1 3))
   => (V> B C D)
(set (v:ref V2 0) 'X)
V1  => (V> A X C D E)
```

(V:FROM-SPECS (Ref_1 ... Ref_n))
(V:FROM-SPECS (Ref_1 ... Ref_n) *StartIndex*)

Creates a vector from the Ref_i. Each Ref_i specifies a data object and the way in which it is to be included in the result. Possible reference formats are lists with the following forms.

- To include *Thing* as a single element of the result, use:

  (SCALAR *Thing*)

- To have *Vector* spliced into the result:

  (VEC *Vector*)

- To splice the subrange of *Vector* from *Lower-Bound* to *Upper-Bound* into the result:

  (SUBRANGE *Vector Lower-Bound Upper-Bound*)

- To insert the *Index*th element of *Vector* into the result:

  (VEC-REF *Vector Index*)

Vec, Subrange, and Vec-Ref items cause data to be shared with the source vector. Scalar items are represented by a pointer to the actual data. The bounds of a Subrange may be reversed. If *StartIndex* is not specified, indices of cells in the vector will range from 0 to S-1 where S is the total number of cells specified. If *StartIndex* is specified, then the indices will range from *StartIndex* to *StartIndex*+S-1. For example

```
(set V1 (V:Make 'A 'B 'C 'D 'E))
(V:From-Specs ((Scalar 1)
               (Vec V1)
               (Subrange V1 0 1)
               (Vec-Ref V1 3)) )
    => (V> 1  A B C D E  A B  D)
```

Extra spaces group elements according to the V:From-Specs clause which produced them. This printing convention is used here only for expositional purposes. The simulator does not keep track of where the various elements came from.

Vec-Ref is shorthand for a single element subrange.

(V:SERIAL-BUILD *init-template fill-template*)

A vector is created as a result of a call to *init-template*. *Fill-template* is T code that may selectively modify elements of the vector. *Fill-template* may contain the following statement which sets the value of the i^{th} element to *new-val*:

(V:MAKE-ELT i *new-val*)

This is useful for creating vectors containing different kinds of elements. Further, the elements need not be built in any particular order. Thus if a particular processor array is easier to configure in an order different from the way it is numbered, then this command may be used.

The following code creates a vector whose elements are set to the elements of a Fibonacci sequence. Empty-elt is a function that returns the literal 'uninitialised'.

```
(v:serial-build (v:new empty-elt size)
                (loop (incr i .in 0 to size)
                      (initial ((prev 0)(cur 1)(temp 0)))
                      (do
                         (v:make-elt i cur))
                      (next    ((temp prev)
                                (prev cur)
                                (cur  (+ temp cur))))))
```

Appendix II contains an example of the use of this command to configure an array of processors as a tree.

(EMPTY-ELT)

A function that returns the literal 'uninitialised'. Useful in connection with v:serial-build to create uninitialised structures.

3.2 Two dimensional Vectors

Two dimensional vectors consist of a one dimensional vector ('outer') each of whose elements is a one dimensional vector ('inner'). The following primitives have been built to manipulate two dimensional vectors. They are completely analogous to those in one dimension and hence are not elaborated upon.

(V2:NEW *init size-in-dimension1 size-in-dimension2*)

(V2:NEWSP *init size-in-dimension1 size-in-dimension2*)
Here *init* is a function of two arguments.

(V2:REF *vec subscript1 subscript2*)

(V2:MAP *function vec*)

(V2:SERIAL-BUILD *init-template fill-template*)

The first subscript always refers to the outer vector, and the second to the inner vector.

Since a two dimensional vector is also a one dimensional vector, one dimensional vector primitives may be used with two dimensional vectors. Thus if v is a two dimensional vector, then

```
(v:ref v i)
```

would refer to the i^{th} inner vector of v.

3.2.1 Triangular arrays

The following function can be used to create a triangular array:

(TRIANGLE:NEW *size init*)

Returns a vector of *size* elements, indexed 0 through *size*-1. The i^{th} element is itself a vector of i+1 elements, each of which is created as result of a call to *init*. *Init* is a function of no arguments.

Note that a triangle is a special two dimensional vector, so that all two dimensional primitives can be used to manipulate it.

4. The AMPS Language: Program Control Statements

The AMPS language contains statements for program control and communication. Program control statements form the language kernel in that the communication statements are built on top of these and can be tailored to suit the requirements of the system being simulated. Chapter 5 discusses communication statements.

The AMPS language contains most constructs commonly found in general purpose programming languages. Although the language is embedded in T, and has a syntax similar to T's, AMPS constructs are fundamentally different in that they are not applicative. Thus they do not evaluate to values and are only used for side-effects. AMPS statements can be composed using other AMPS statements and T expressions. In fact, T expressions can be freely used in AMPS code. This allows use of all the T libraries.

Besides the program section in a box-type definition, AMPS statements can also be used in *subroutines*. AMPS subroutines are distinct from T functions; they are defined and called using special syntax.

AMPS code can also contain calls to T macros.

The following sections describe the AMPS language statements. The syntax is very similar to corresponding statements from T.

4.1 The BLOCK statement

(BLOCK *statement* ...)

Statements are T expressions or AMPS statements. These are evaluated from left to right. A BLOCK statement is useful for grouping together statements in places where, syntactically, a single statement is needed.

4.2 The CALL statement

(CALL *procedure-call-form return-list*)

Procedure-call-form is a list of the form (*subroutine-name arg* ...), with *subroutine-name* defined in a SUBROUTINE statement. *Return-list* is a list of settable T expressions.

The statement results in a call to *subroutine-name*. The expressions in the *return-list* are used to receive values returned as a result of the call. The length of *return-list* must be the same as the number of values returned by the subroutine.

4.3 The CASE statement

(CASE *key clause* ...)

CASE performs a multiway dispatch on the value of the *key*, a T expression. Each *clause* is of the form (*key-values statements* ...), where *key-values* is a list of values against which the value of the *key* expression is compared (using EQ?), and *statements* are AMPS statements or T expressions. The *statements* following the *clause* which matches the *key* are evaluated sequentially. The last *clause* may be of the form (ELSE *statements* ...); this designates the default action to be taken if there is no match. If the *key* never matches and there is no default clause, then no action is performed.

4.4 The DECLARE statement

(DECLARE *declarations statement* ...)

Virtually identical to the LET* statement of T. Each *declaration* is of the form (*var initial*), where *var* is a T identifier and *initial* is a T expression. Each *statement* is a T expression or an AMPS statement. The statement creates a scope in which the var_i are local variables with initial values $initial_i$. The *statements* are then executed sequentially in this scope.

4.5 The FORK statement

(FORK *thread* ...)

Each *thread* is a list of T expressions or AMPS statements. When control reaches the fork statement, all *threads* start execution concurrently. The execution of a single thread consists of sequential execution of the statements or expressions in it. No assumptions can be made about the relative speed of the execution of the various threads. The fork statement terminates when all the *threads* complete their execution.

4.6 The IF statement

(IF *condition consequent [alternate]*)

The *condition* is a T expression, and *consequent* and *alternate* are AMPS statements or T expressions. The *condition* is evaluated, and if it is not nil, then the *consequent* is executed, else the *alternate* is executed if it is present.

4.7 The LOOP statement

(LOOP *clause* ...)

The LOOP statement and also the description below is adapted from the Yale LOOP macro written by John Ellis, which itself is based on the LOOP from McDermott, Charniak and Riesbeck [2]. It is a powerful construct especially suited for complicated loops that have several exit tests, loop variables, etc.

A LOOP consists of various components: a body which may be executed several times, code to be executed before entering the body for the first time or after leaving it for the last time, loop termination tests, and local variables which can be stepped or or otherwise changed every iteration. The LOOP statement provides for all these using the following clauses:

```
(INITIAL initializations ...)
(BEFORE statements ...)
(WHILE condition)
(UNTIL condition)
(INCR v FROM|.IN.|.IN|IN.|IN init [TO final] [BY delta]) ;;; also STEP
(DECR v FROM|.IN.|.IN|IN.|IN init [TO final] [BY delta]) ;;; also DOWNSTEP
(FOR v IN e)
(DO statements ...)
(NEXT updates ...)
(AFTER statements ...)
```

In general, there may be more than one clause of each type in a LOOP. For example, the following loop has two exit tests, one in the middle and one at the end:

```
(loop ...
      (do ...)
      (while condition)
      (do ...)
      (until condition) )
```

4.7.1 *LOOP* clauses

In the discussion below, "declares a new variable" means causes a declaration of a new variable and that the variable is scoped over the entire loop.

(INITIAL *initialization* ...)

Each *initialization* is of the form (*var-name initial [next]*). where each *var-name* is a T identifier, and *initial* and *next* are T expressions.

The statement causes new variables *var-name*$_i$ to be declared and sequentially initialized at the beginning of the loop to the value of the expressions *initial*$_i$. If *next*$_i$ is present, then *var-name*$_i$ is set to *next*$_i$ at the bottom of the loop.

(BEFORE *statement* ...)

Each *statement* is an AMPS statement or a T expression. The *statements* are executed sequentially after all the variables have been intialized and before the first loop iteration. BEGIN is a synonym for BEFORE.

(WHILE *condition*)

Condition is a T expression. The *condition* is evaluated, and if nil, causes the loop to be terminated.

(UNTIL *condition*)

Condition is a T expression. The loop is terminated if *condition* evaluates to non-nil.

(INCR *v* {FROM|.IN.|.IN|IN.|IN} *init* [TO *final*] [BY *delta*])
(DECR *v* {FROM|.IN.|.IN|IN.|IN} *init* [TO *final*] [BY *delta*])

V is a T identifier, and *init*, *final* and *delta* are T expressions.

Declares a new variable *v* and steps it from initial value *init* to the final value *final*, incrementing (decrementing) by *delta* (defaults to 1). If [TO *final*] is omitted, the variable is stepped indefinitely. Otherwise, the loop terminates when the variable reaches the final value. The *init*, *final* and *delta* expressions are evaluated once, at the beginning of the loop. STEP and DOWNSTEP are synonyms for INCR/DECR. The FROM/.IN./... keywords are provided to make writing loops that step from m to n-1 more convenient:

```
(INCR i .IN.  0 TO 10)   varies i in 0..10
(INCR i .IN   0 TO 10)   varies i in 0..9
(INCR i  IN.  0 TO 10)   varies i in 1..10
(INCR i  IN   0 TO 10)   varies i in 1..9
(DECR i .IN. 10 TO  0)   varies i in 10..0
(DECR i .IN  10 TO  0)   varies i in 10..1
(DECR i  IN. 10 TO  0)   varies i in 9..0
(DECR i  IN  10 TO  0)   varies i in 9..1
```

FROM is a synonym for .IN.

(FOR *v* IN *list-exp*)

Declares a new variable *v* and steps it through each element of list *list-exp*, which is a T expression evaluating to a list. The loop terminates when the list is exhausted.

(DO *statement* ...)

The body of the loop-- the *statements* are AMPS statements or T expressions. They are executed sequentially each time through the loop.

(NEXT *update* ...)

Each *update* is of the form *(var val)* where *var* is a settable T expression and *val* is a T expression. The expressions var_i are equentially assigned to val_i (the assignment is NOT done in parallel). Note that the assignment does not occur at the bottom of the loop (as in the INITIAL) clause, but wherever the NEXT statement is located.

(AFTER *statement* ...)

Each *statement* is a T expression or an AMPS statement. The *statements* are executed sequentially after the LOOP is found to have terminated.

4.8 The RETURN statement

`(RETURN` *value ...*)

The RETURN statement is legal only inside a SUBROUTINE statement or a userdefined section, provided the section definition begins with the keyword CONCURRENT. Each *value* is a T expression. The T expressions are evaluated, and the resulting values are returned. The control then returns to the calling program.

4.9 The SUBROUTINE statement

`(SUBROUTINE` *(subroutine-name formal-parameter ...) statement ...*)

Subroutine-name and each *formal-parameter* are legal T identifiers. *Statements* are AMPS statements or T expressions. They must contain one or more RETURN statements.

The SUBROUTINE statement defines *subroutine-name* to be a subroutine that can be called using a CALL statement. When a subroutine is called, the actual parameters are bound to the formals, then the *statements* are executed. When the execution encounters a RETURN statement, the values mentioned in it are returned.

4.10 The WAIT statement

`(WAIT` *delay*)

Delay is a T expression that evaluates to a positive integer. The calling process suspends for the amount of *delay* specified and then continues execution.

5. The AMPS Language: Communication Primitives

This chapter describes primitives for modelling communication, synchronization or resource contention. Other primitives can be created, this will be discussed in chapter 8.

The set of primitives described here uses the notion of *ports*. The primitives themselves are operations on ports, and their semantics are different for the different types of ports. Two kinds of ports are predefined: *mailboxes* and *pools*. Three primitives: SEND, RECEIVE and ND-CHOOSE can be used with these. These primitives are legal AMPS statements.

This chapter concludes with a discussion of how to model resource contention, with brief program examples. The appendices contain numerous examples of the use of the primitives for communication.

5.1 Ports

A port is an intermediary for communication. Processes do not communicate directly, but through ports. Thus unlike several other concurrent language proposals [4, 5] a message receiver does not need to know which process sent a message, and vice versa. Senders and receivers only need to know the name of the port through which communication takes place. This has the advantage that multiple server processes can be modelled easily by letting them read their requests from a single port. Ports do not enforce readership/writership on processes i.e. a process can send or receive messages from the same port, and in fact, it may even receive a message sent by itself.

There may be different types of ports. The message queuing strategies, the amount of buffering at the port, and other communication parameters are determined by the port type. Two type of ports are predefined in AMPS: *pools* and *mailboxes*. Ports can be created by invoking

(*type-of-port*:`NEW`)

which returns the port object. Like other AMPS objects, it is possible to build vectors of ports.

5.1.1 Mailboxes

A mailbox is a port for modelling one to one communication without any buffering. When a (sender) process sends a message to a mailbox, it must wait until there is a (receiver) process that wants to receive a message from the mailbox. If there are more than one waiting receiver processes, a receiver is selected at random, the message is transferred to the selected process, and both the selected receiver and the sender continue. If there are no waiting receiver processes, then the sender process must wait (possibly along with any other process waiting to send to the mailbox) until it is selected by a receiver process. The SEND and RECEIVE operations are thus synchronizing and symmetric.

5.1.2 Pools

A pool is a port for one to one communication with infinite buffering. SEND and RECEIVE on a pool are asymmetric. Sender processes do not wait for receivers, but the messages to be sent are stored in the pool. Receiver processes must, however, wait for messages.

5.2 The SEND and RECEIVE statements

Mesages can be sent using:

`(SEND` *port message*`)`

Port and *message* must be T expressions which should evaluate to the port to which the communication is directed and the message to be sent respectively.

Messages may be received by:

`(RECEIVE` *port var*`)`

Var must be a settable T expression which is set to the message received. *Port* must be a T expression which evaluates to the port from which the message is to be received.

5.3 The ND-CHOOSE statement

The ND-CHOOSE statement has the following form:

`(ND-CHOOSE` *chosen-mailbox chosen-message port-expression ...*`)`

Chosen-mailbox and *chosen-message* must be settable T expressions, and each *port-expression* must be a mailbox or a pool, or a list or a vector of mailboxes or pools.

This statement Non-deterministically CHOOSEs a port from the specified ports. A message is RECEIVEd from the chosen port, and *chosen-port* and *chosen-message* are set accordingly. Only those ports actually holding a message at the time are considered for the non-deterministic choice. If none of the specified ports holds a message, then the process invoking ND-CHOOSE waits until a message appears on any one of the specified ports. This statement is a generalisation of the Parallel-Or construct of Linda [3], or the Select statement of Ada [6] in that it does not require all the target ports to be textually listed.

The next section contains an example of the use of the ND-CHOOSE statement.

5.4 Modelling Resource Management

The act of communication implies a synchronization between the sender and the receiver. If the message is considered to be a resource, then receiving the message can be regarded as acquiring the resource, while sending the message (with buffering) can be regarded as releasing the resource.

5.4.1 Dining Philosophers

The first resource management problem considered in this section is the Dining Philosophers' problem.

```
(def-box table (nphils)
     (locals
        (shelf (pool:new))
        (forks (v:new pool:new nphils)))
     (boxes
        (phils (v:new (lambda (i) (philosopher:new
                                     shelf
                                     (v:ref forks i)
                                     (v:ref forks (mod (1+ i) nphils))))
                        nphils)))
     (program                        ;;; set the table
        (loop (incr i .in 0 to nphil)
              (do
                 (send (v:ref forks i) 'fork)))
        (loop (incr i .in 1 to nphil)
              (do
                 (send shelf 'plate)))))

(def-box philosopher (shelf lside rside)
     (locals
        (plate nil)
        (lfork nil)
        (rfork nil))
     (program
        (loop (do
                    ... <think> ...

                  (receive shelf plate)
                  (fork ((receive lside lfork))
                        ((receive rside rfork)))

                    ... <eat> ...

                  (fork ((send shelf plate))
                        ((send lside lfork))
                        ((send rside rfork))))))
```

Figure 5-1: Dining Philosophers

The problem is to simulate a system of N philosophers sitting at a round table set with N forks, one between adjacent philosophers. The philosophers alternate between thinking and eating, but to eat, a philosophers must wait till he can pick up the two forks to either side of his. When the philosopher finishes eating, he puts down the forks (allowing his neighbours to use them) and resumes thinking. If, for example, every philosopher picks up the fork on his left, then this leads to deadlock, because now every philosopher must wait till his right neighbour puts down his fork, which can only happen if he finishes eating. The deadlock is avoided in the solution presented in figure 5-1 by introducing an additional resource. Now the philosophers must first contend for *plates*, of which there are only *N-1*. After picking up a plate, from a central *shelf*, the philosophers can contend for forks. Thus all philosophers cannot now pick up one fork and cause a deadlock because there are only *N-1* plates. The plates and the forks are modelled as pools and each philosopher is modelled as a process. Resource acquisition/release is modelled by receiving/sending null messages. The program associated with the table process is only used to initialise the pools.

5.4.2 Routing On A Hypercube

Routing a message on a binary hypercube interconnection network consists of transmitting it across all the dimensions in which its source and destination differ. A given wire can carry only one message at a time, and hence this gives rise to a resource constraint when multiple messages are being routed simultaneously. Consider a routing strategy in which the path of a message is not determined independently of other messages in the system, but is instead computed dynamically as the message moves through the network, depending upon the wires free at that time. Figure 5-2 sketches how a network employing such a strategy may be modelled.

Each message sent on the hypercube is represented as a process which contends for hypercube wires. Each message process initially has a list of dimensions that it needs to cross, and it terminates when the list is emptied. At each intermediate destination the process contends for wires corresponding to yet uncrossed dimensions. When it gets access to a wire the corresponding dimension is removed from its list, and the position of the message is updated. The process then waits for a certain time to simulate transmission delay and then releases the wire. Hypercube wires are represented by two pools, one for each direction. Availability of a wire for transmission in a particular direction is represented by the presence of a null message in the associated pool.

```
(def-box message (list-of-uncrossed-dimensions sourcenode)
    (locals
        (current-location nil)
        (list-of-mailboxes nil)
        (chosen-mailbox nil)
        (token nil)
          .
          .)

    (program
        (set current-location sourcenode)
        (loop (while list-of-uncrossed-dimensions)
                .
                .
              <Compute list-of-mailboxes corresponding to wires
               along dimensions in list-of-uncrossed-dimensions>
                .
                .
              (nd-choose chosen-mailbox token list-of-mailboxes)
                .
                .
              <Determine the dimension along which a free wire was
               found.  Update current-location and remove that
               dimension from list-of-uncrossed-dimensions>
                .
                .
              (wait message-transmission-time)
              (send chosen-mailbox token))      ;; release wire
        ))
```

Figure 5-2: Hypercube Routing

6. Modelling Time

In AMPS, a user can specify the time necessary to perform every interesting operation in his program. By varying these times, a wide range of architectures can be modelled.

The time required to perform an operation can be specified in two ways. The operation can be enclosed in a procedure that contains the timing information in addition to the code for performing the operation. Alternatively, T macros can be used. Once this is done and the primitive procedure/macro is defined, it can be used freely in AMPS code.

6.1 Global And Local Clocks

AMPS has the notion of a systemwide *global time* and a *local time* for each thread of control.[2] The state of the local variables of a process is always consistent with its local time, i.e. the local time correctly indicates the time required by the process to reach that state. The local time for a thread need not be the same as the global time. Before a thread communicates with the external world, however, it is forced to wait until the rest of the world catches up, i.e. the global time becomes equal to its local time. This ensures that all communication take place at the right global time, and that a process is not affected by another process ahead of it in time.

Implicit here is the assumption that all processes interact with each other or cause globally observable side-effects only using AMPS communication statements. If this discipline is not observed, and processes share information in any other way (e.g. through T global variables), then it is possible for a process to be influenced by future actions of another process.

The statement

```
(WAIT 0)
```

can be used to make the local time of a process equal to the global time.

The facilities provided for accessing the global and local clocks are discussed in chapter 7.

[2]Note that a process may execute a FORK statement, and cause several concurrently executing threads of control to be created. Hence, for modelling time it is necessary to consider threads of control rather than processes. A process that is not executing a FORK constitutes a single thread of control.

6.2 Modelling computation delay

The following primitive is provided:

(OPDELAY *amount-of-delay*)

Amount-of-delay is a T expression which should evaluate to the delay required. This statement may be inserted in any procedure. Every execution of the statement causes the concerned thread of control to be delayed by the amount specified. As an example consider the following procedure which returns the quotient of two numbers as well as simulates the computation delay.

```
(define (./ a b)
        (opdelay *DIVISION-DELAY*)
        (/ a b))
```

6.3 Modelling communication delay

Communication delay can be modelled using T macros:

```
(define-syntax (.receive port var)
        `(block (receive ,port ,var) (opdelay receive-delay-expression)))
```

Or using AMPS subroutines

```
(subroutine (.receive port)
            (declare ((msg nil))
                     (receive port msg)
                     (opdelay receive-delay-expression)
                     (return msg)))
```

Unlike the macro implementation, this would have to be used only through a CALL statement. Notice that for the subroutine, as well as the macro implementation, the expression *receive-delay-expression* is evaluated at runtime. Thus it is possible to simulate delays equal to the message length etc.

6.4 Predefined delay modelling functions

The four arithmetic operators have already been transformed in the manner described above. The resulting functions are:

```
.+,  .-,  .*  and ./
```

If these are used in the PROGRAM section of a box, they not only perform the indicated operation, but also simulate a computation delay of respectively:

```
*ADDITION-DELAY*,
*SUBTRACTION-DELAY*,
*MULTIPLICATION-DELAY* and
*DIVISION-DELAY*
```

These variables can be set as desired.

7. Simulation control

This chapter describes what a simulation session looks like and the facilities provided for controlling a simulation.

7.1 General simulation procedure

A simulator session begins with loading in the simulator. After this the various user defined box-types/utilities may be loaded. The simulator is then reset, and the top level box instantiated. After this the boxes in the system may be commanded to run, and controlled, using commands described below. More than one architecture may be simulated in a single session. Box-type definitions may be loaded at any time, but the simulator needs to be reset before starting on a new architecture.

The appendices contain definitions of various architectures and also show how the definitions can be used in an actual session.

7.2 Commands

`(SIM:CLEAR)`

This command resets the simulator. All old objects are deleted. The global clock is reset to zero. Various simulator internal variables are reinitialised, and the simulator is ready to instantiate a new box hierarchy.

`(SIM:INIT` *top-level-box*`)`

This command starts the simulation for the box hierarchy created. The whole hierarchy must be contained in *top-level-box* which must be already created. The processes created execute to termination or until all of them have to wait for an external event. Then the control returns to the user.

`(SIM:TICK)`

This causes the global time to increment by 1 unit, and causes all processes waiting for the new time to resume.

`(SIM:KEEP-TICKING)`

Causes the global clock to keep ticking till there are no more processes waiting.

`(SIM:TIME)`

Returns the current global time.

`(SIM:LOCAL-TIME)`

Returns the local time for the thread of control calling it. Unpredictable if called from outside an AMPS program.

8. Implementation Notes

This chapter describes in brief the main ideas in the implementation of the AMPS programminng language and also how new communication primitives can be developed.

8.1 Continuation Passing

The basic requirement for simulating multiple processes is a mechanism for process suspension and resumption. As noted in [9], the CATCH operator performs this admirably. However, in the absence of a fully general CATCH operator implementation in T, other means must be used. The approach adopted in AMPS is to use continuations [8] to encapsulate program state.

It is not, however, necessary to completely convert a program into continuation passing style. The real requirement is that continuations be available wherever it might be necessary to suspend execution. Suspension is acheived simply by saving the continuation. A suspended process can be resumed by retreiving the saved continuation and calling it. It is also possible to transfer information to the process by calling the continuation with the appropriate arguments.

8.1.1 *CALL* statement Transformation

As an example, figure 8-1 shows how a CALL statement is converted to continuation passing style.[3] After the transformation, the procedure call contains an extra parameter, the second parameter[4] now being the continuation. The SUBROUTINE statement used to define procedure *abc* suitably transforms the definition so that it has the appropriate number of formal parameters. The RETURN statement in the definition simply calls the second parameter with the result of the procedure.

```
;;; source program              ;;; after transformation

(call (abc a b c)               (abc a (lambda (arg1 arg2 arg3)
      (d e f))                                 (set d arg1)
   .                                           (set e arg2)
   .                                           (set f arg3)
   rest                                        .
   .                                           .
   .                                           rest
                                               .
                                               .) b c)
```

Figure 8-1: CALL statement transformation

[3]Note that all AMPS statements only cause side-effects, this simplifies the conversion.

[4]The call might be an operation on an object, in which case T requires that the object be the first parameter.

8.1.2 Concurrent operations

User defined operations on boxes that have the keyword CONCURRENT in their definition are called concurrent operations. Concurrent operation definitions are transformed in a manner similar to SUBROUTINEs. In either case, the continuation with which the operation/subroutine gets called is bound to RETURN. Thus the RETURN statement actually calls this continuation. It is not necessary, however, that the continuation always be called. If it is saved and called later, the effect is process suspension and later resumption. This is used to implement the communication primitives.

8.2 The communication interface

Internally, ports are simply instances of predefined box-types. The communication statements are concurrent operations on ports, with some syntactic sugar.

```
;;; Pools.
;;;=====================================================================
;;; Note that this does not implement the ND-CHOSE operation.

(def-box pool ()
 (locals
    (messgq nil)
    (recvq  nil))
 (CONCURRENT ..receive ()
      (wait 0)
      (if messgq
          (return (pop messgq))
          (push recvq return)) )
 (CONCURRENT ..send (message)
      (wait 0)
      (if recvq
          ((pop recvq) message)
          (push messgq message))
      (return))
)

;;; Syntactic sugar.
;;;=====================================================================

(define-syntax (send mbx msg)
               `(call (..send ,mbx ,msg) ()))
(define-syntax (receive mbx msg)
               `(call (..receive ,mbx) (,msg)))
```

Figure 8-2: Pool Implementation

Figure 8-2 shows an implementation of pools that can support the SEND and RECEIVE primitives of chapter 5. In a similar manner, it is possible to implement port-types having different queuing strategies (say First In First Out), or fixed amount of buffering etc. More operations on ports could also be defined, e.g. an operation that finds the number of unsent messages.

8.3 Alternate Communication Interfaces

The ideas discussed above allow one to define communication interfaces conveniently.

```
;;; Tuple-space
;;;============================================================================
;;; Note that this does not implement the READ operation.

(def-box tuple-space ()
   (locals
        (selected-message  nil)
        (selected-receiver nil)
        (messageq nil)
        (receiverq  nil))
   (CONCURRENT ..in (predicate)
        (wait 0)
        (set selected-message (any? predicate messageq))
        (if selected-message
            (block (delq selected-message messageq)
                   (return selected-message))
            (push recvq (list predicate return))))
   (CONCURRENT ..out (message)
        (wait 0)
        (set selected-receiver
             (any? (lambda (predicate-continuation-pair)
                           ((car predicate-continuation-pair)
                            message))
                   receiverq))
        (if selected-receiver
            (block (delq selected-receiver receiverq)
                   ((cadr selected-receiver) message))
            (push messageq message))
        (return))
)

;;; Syntactic sugar.
;;;============================================================================

(define-syntax (in tspace predicate dest-var)
               `(call (..in ,tspace ,predicate) (,dest-var)))
(define-syntax (out tspace tuple)
               `(call (..out ,tspace ,tuple)))
```

Figure 8-3: Tuple-space Implementation

Figure 8-3 shows an implementation of primitives similar to those in Linda [3]. The basic communication device here is a tuple space, which can be shared amongst all processes. A tuple space holds tuples of data and responds to three operations: IN, OUT and READ. Figure 8-3 only shows the implementation of IN and OUT, READ can be similary implemented. The OUT primitive allows one to deposit a tuple into a tuple space, while the IN primitive allows one to remove a tuple satisfying the specified predicate from a tuple space. The process executing an IN statement is made to suspend until a tuple of the kind it demands is available. Arbitrary predicates may be specified in the IN statement, and in this sense this is a generalisation of the primitives of Linda. Unlike Linda, more than one tuple space can be created. The code shown in the figure uses lists for implementing queues. This is only for reasons of expositional clarity, and obviously, other data structures like hash tables may be used.

8.4 Use Of T Macros

The code examples in the previous sections contained several instances of T macro definitions and calls. Indeed, T macros can be freely used in AMPS code. The program that transforms AMPS primitives into T code expands T macros whenever necessary.

T macros may be used to define more convenient syntax as in figure 8-2 or figure 8-3, or could even be used to compose more primitives. Another example of the use of macros was in section 6.2, where a macro was used to define a receive statement that models communication delays.

I. A Shuffle-nearest-neighbour Architecture

The following file defines a shuffle-nearest-neighbour architecture for finding the partial sums of a vector.

```
(herald shuffle-nn-network)

;;; Processors in a shuffle nearest neighbour interconnect
;;;============================================================================
;;; The problem:
;;;  To find the partial sums of an array of size netsize.
;;; About the algorithm:
;;;  ith processor contains ith element.  Standard algorithm using
;;;  log(netsize) steps.  Towards the end only a few of the processors
;;;  remain active,  i.e. though they participate in moving the data
;;;  around, they do not perform any additions.  This is controlled
;;;  by shifting out the activity? program variable.
;;;============================================================================
;;; Processor parameters:
;;;  myid       :   Processor id.  The ith processor holds the ith element
;;;                 of the array.
;;;  elem       :   The array element
;;;  netdesc    :   A network descriptor.  Holds network parameters like
;;;                 network size, defines the shuffle and unshuffle
;;;                 operations, also the operations lshif (which depends
;;;                 upon the network size).
;;; nn+, nn-    :   The nearest neighbour mailboxes.
;;; shf+, shf-  :   The shuffle and unshuffle mailboxes.
;;;
(def-box proc (myid elem netdesc nn+ nn- shf+ shf-)
 (locals
  (inmsg nil)
  (outmsg nil))
 (program
  (set outmsg elem)
  (loop (incr iter from 1 to (netdesc:lognsize netdesc))
        (initial (activity? myid))
        (do
          (fork
            ((receive  nn- inmsg ))
            ((send     nn+ outmsg)))
          (if (>0? activity?)
                  (set outmsg (.+ outmsg inmsg)))
          (fork
            ((receive shf+ inmsg  ))
            ((send    shf- outmsg )))
          (set outmsg inmsg))
    (next (activity? (lshif netdesc activity?))))

  (format t "~% My Id: ~a    Partial sum: ~a" myid outmsg)))

;;; Top level box for shuffle nearest neighbour interrconnect
;;;============================================================================
;;; Parameterised on the size of the network.
;;; Locals:
;;; nn          :   Mailbox array for nearest neighbour communication.
;;; shf         :   Mailbox array for shuffle/unshuffle communication.
;;; Boxes:
;;; Netdesc     :   A box that holds network size related informations, e.g.
;;;                 the various functions representing the interconnections.
;;;
;;; Procs       :   The processor array.
```

```
(def-box net (netsize)
 (locals
  (nn      (v:new mailbox:new netsize))
  (shf     (v:new mailbox:new netsize)))
 (boxes
  (netdesc (netdesc:new netsize))
  (procs (v:newsp
          (lambda (i) (proc:new i i netdesc
                            (v:ref nn i)                   ;;; nn+
                            (v:ref nn (prev netdesc i))    ;;; nn-
                            (v:ref shf i)                  ;;; shf+
                            (v:ref shf (ushfl netdesc i))));;; shf-
          netsize))))

;;;*********************** END OF FILE *****************************************
```

The following file contains the utilities needed:

```
(herald shfnnutils)
;;; Network descriptor for a shuffle nearest neighbour array.
;;;=============================================================================
;;; Locals:
;;;  Just so that they may be computed once for all.
;;; Operations:
;;;  Shfl   :  (Shuffle) One bit circular right shift.
;;;  Ushfl  :  (Unshuffle) One bit circular left shift.
;;;  lshif  :  One bit simple left shift (msb drops out).
;;;  prev   :  Previous element in the nearest neighbour scheme
;;;            i.e. num - 1 mod netsize.


(def-box netdesc (netsize)
 (locals
  (lognsize     (log2 netsize))
  (lnsize-1     (-1+ lognsize)))
 (shfl (num)
  (set-bit-field (set-bit-field 0 1 lnsize-1 (bit-field num 0 lnsize-1))
                 0
                 1
                 (bit-field num lnsize-1 1)))
 (ushfl (num)
  (set-bit-field (set-bit-field 0 0 lnsize-1 (bit-field num 1 lnsize-1))
                 lnsize-1
                 1
                 (bit-field num 0 1)))
 (lshif (num)
  (set-bit-field 0 1 lnsize-1 (bit-field num 0 lnsize-1)))
 (prev  (num)
  (mod (-1+ num) netsize)))


;;; A function needed above:

(define (log2 num)
        (loop (initial (pwr 1) (res 0))
              (while (< pwr num))
              (next (pwr (* pwr 2)) (res (1+ res)))
              (result res)) )

;;;*********************** END OF FILE *****************************************
```

The following may be used to instantiate and simulate a shuffle exchange network of n processors.

```
(herald test shuffle-nn)
;;;============================================================================
;;; The following controls the simulation.
;;;============================================================================
;;; The simulator is first reset.
;;; A network of size 8 is then instantiated.
;;; The processes created are started up.
;;; The clock is made to tick until the system settles down.
;;;============================================================================

(sim:clear)
(set net (net:new 8))
(sim:init net)
(sim:keep-ticking)

;;;*********************** END OF FILE *****************************************
```

II. A Tree Architecture

The following defines a tree of processors for performing cyclic reduction.

```
(herald tree:main-process-declarations)

;;; Description of a single processor in a TREE OF PROCESSORS
;;;========================================================================
;;; Program:
;;;  Solving tridiagonal systems by doing cyclic reduction.
;;; Algorithm:
;;;  See Lennart Johnsson, "Odd-even cyclic reduction on ensemble
;;;      architectures and the solution of tridiagonal systems of
;;;      equations", Dept. of Comp. Sc., Yale University, RR-339,
;;;      October 1984.
;;;========================================================================
;;; Parameters:
;;;  my-id        : serial number in an inorder numbering.
;;;  kind-of-node
;;;               : root, inner, or leaf.
;;;  parent       : parent mailbox.
;;;  lson, rson   : left and right son mailboxes.
;;; Locals:
;;;  x            : Will hold my-id th element of the solution vector.
;;;  state        : This is set to a structure during intialisation.
;;;                 The structure has components a,b,c,y corresponding to
;;;                 the 3 diagonal elements of the my-idth row and the myidth
;;;                 rhs element.
;;;  Others       : Intermediate values.
;;; Operations:
;;;  Set-state    : Initialises the processor.
;;;  show         : Prints the value of x.

(def-box proc (my-id kind-of-node parent lson rson)
 (locals
  (x      'uninitialised)
  (state nil)
  (lmsg   nil)
  (rmsg   nil)
  (msg    nil))
 (set-state (new-state)
  (set state new-state))
 (show ()
  (format t "~% X ( ~a )   =   ~a " my-id x))
 (program
  (loop (initial (newid my-id))
        (until (even? newid))
        (do
          (receive lson lmsg)
          (receive rson rmsg)
          (if (neq? kind-of-node '*root*)
          (send parent (new-msg state lmsg rmsg)))
          (set state (next-state state lmsg rmsg)))
        (next (newid (div newid 2))))
  (if (neq? kind-of-node '*root*)
        (send parent (last-msg state)))

  (format t "~% End of phase1 for ~a, state: ~a ~a ~a ~a" my-id
                                                          (state-a state)
                                                          (state-b state)
                                                          (state-c state)
                                                          (state-y state))
 (case kind-of-node
        ((*root*)    (set x (./ (state-y state) (state-b state)))
                     (send lson (msg:new 0 x))
                     (send rson (msg:new x 0)))
        ((*inner*)   (receive parent msg)
                     (set x (./ (.- (state-y state)
                                      (+ (.* (state-a state) (msg-first msg))
                                         (.* (state-c state) (msg-second msg))))
                                   (state-b state)))
                     (send lson (msg:new (msg-first  msg) x))
                     (send rson (msg:new x (msg-second msg))))
        ((*leaf*)    (receive parent msg)
                     (set x (./ (.- (state-y state)
                                      (+ (.* (state-a state) (msg-first msg))
                                         (.* (state-c state) (msg-second msg))))
                                   (state-b state))) )

        (else   (error "~% Illegal processor kind.")))))

;;; Top level box to configure the tree
;;;==============================================================================
;;; Parameterised on the size of the tree.
;;; Locals:
;;;  mbs        : array of mailboxes.  Mailbox i will become the link
;;;               between processor i and its parent.
;;; Boxes:
;;;  procs      : vector of processors.  The vector is built up so that
;;;               the ith element has number i in an inorder numbering of the
;;;               tree.  This has a straightforward recursive formulation:
;;;               The id of the root and its sons is known, hence it can
;;;               be directly configured.  But knowing that, the id of the
;;;               sons of the sons can be known and so on.  Thus a single
;;;               recursive procedure build-node is used.  This receives
;;;               two parameters, the id of the processor to be configured,
;;;               and 'sep' (the seperation between the parent and the sons);
;;;               this allows the ids of the left and right sons to be
;;;               computed.  Thus the current processor can be configured
;;;               by passing it the appropriate id, type and its
;;;               mailboxes.   Its sons can be recursively configured,
;;;               because the seperation  between the parent and the sons
;;;               halves as one goes down the tree.
;;; Operations:
;;;  Show       : Simply mapped onto the individual processors.
;;;  Init-tree  : Initialises the processors using elements from a list.
;;;               a,b,c refer to the three diagonal elements, y refers to
;;;               the element of the  rhs in every processor.

(def-box tree (size)
 (locals
  (index-of-root (div (-1+ size) 2))
  (mbs   (v:new mailbox:new size)))
 (boxes
  (procs (v:serial-build (v:new empty-elt size)
        (labels (
                 (build-node
                  (lambda (id sep)
                    (let* ((new-sep (div sep 2))
                           (lson    (- id new-sep))
                           (rson    (+ id new-sep)))

                      (cond ((eq? id index-of-root)
                             (v:make-elt id (proc:new id '*root*
                                                      (v:ref mbs id  )
                                                      (v:ref mbs lson)
```

```
                                                   (v:ref mbs rson)))
                              (build-node lson new-sep)
                              (build-node rson new-sep))
                             ((>0? new-sep)
                              (v:make-elt id (proc:new id '*inner*
                                                   (v:ref mbs id  )
                                                   (v:ref mbs lson)
                                                   (v:ref mbs rson)))
                              (build-node lson new-sep)
                              (build-node rson new-sep))
                             (t
                              (v:make-elt id (proc:new id '*leaf*
                                                   (v:ref mbs id  )
                                                   nil
                                                   nil))))))))
                   (build-node index-of-root (div (1+ size) 2))))))

 (show ()
  (v:map show procs))
 (init-tree (init-list)
  (loop (initial (i 0))
        (while   (< i size))
        (do
           (let* ((a (pop init-list))
                  (b (pop init-list))
                  (c (pop init-list))
                  (y (pop init-list)))
                 (set-state (v:ref procs i) (state:new a b c y))))
        (next (i (1+ i))))))

;;;****************** END OF FILE **********************************************
```

This uses the following utility functions to manage the communication and computation in a structured manner.

```
(herald tree:utilities)


(define-structure-type state a b c y)
(define (state:new a b c y)
        (let ((state (make-state)))
             (set (state-a  state) a)
             (set (state-b  state) b)
             (set (state-c  state) c)
             (set (state-y  state) y)
             state))

(define-structure-type msg   first second)
(define (msg:new f s)
        (let ((msg (make-msg)))
             (set (msg-first  msg) f)
             (set (msg-second msg) s)
             msg))


(define (next-state state lmsg rmsg)
   (let* ((left (msg-second lmsg))
          (right (msg-first rmsg))
          (ai    (state-a   state))
          (bi    (state-b   state))
          (ci    (state-c   state))
          (yi    (state-y   state))
          (alpha (./ (.- 0 ai) (state-b left)))
          (gamma (./ (.- 0 ci) (state-b right))))
         (state:new (.* alpha (state-a left))
                    (.+ bi (.* alpha (state-c left)) (.* gamma (state-a right)))
                    (.* gamma (state-c right))
                    (.+ yi (.* alpha (state-y left)) (.* gamma (state-y right))))
))


(define (new-msg state lmsg rmsg)
        (msg:new (msg-first lmsg) (msg-second rmsg)))


(define (last-msg state)
        (msg:new state state))

;;;****************** END OF FILE **********************************************
```

The tree may be configured and simulated using:

```
(herald test-tree)

(sim:clear)
(set tree (tree:new 7))
(init-tree tree '(0 1 1 1
                  0 1 1 2
                  0 1 1 3
                  0 1 1 4
                  0 1 1 5
                  0 1 1 6
                  0 1 1 7))

(sim:init tree)
(show tree)

(sim:keep-ticking)
(show tree)

;;;****************** END OF FILE **********************************************
```

III. A Triangular Array Of Processors

The following file defines a two dimensional triangular processor array for doing LU decomposition of a matrix.

```
(herald givens-array)
;;;=============================================================================
;;; Definition of each processor of the Givens array
;;;=============================================================================
;;; Algorithm:
;;;  See J.M. Delosme, M. Morf, "Scattering Arrays For Matrix Computations"
;;;      SPIE Vol. 298 Real-time Signal Processing IV(1981) pp 74-83.
;;;=============================================================================
;;; The input parameters are:
;;; mode        :   0 for linear, 1 for Euclidean norm
;;; rowno, colno:   coordinates of the processor in the array
;;; north, east, south, west
;;;             :   mailboxes for communication in the four directions
;;; size        :   size of the input matrix
;;;

(def-box proc (mode rowno colno north east south west size)
 (locals
  (scratch nil)
  (x       nil)
  (y       nil)
  (scx     nil)
  (scy     nil)
  (v       nil)
  (theta   nil)
  (sintheta   nil)
  (costheta   nil)
  (inn     0.0)
  (ine     0.0)
  (diag? (eq? rowno colno)))
 (initialise ()
  (if (eq? mode 0)
      (block
       (set v (./ scy scx)))
      (block
       (set theta (atan (./ scy scx)))
       (set sintheta (sin theta))
       (set costheta (cos theta)) )))
 (transform ()
  (if (eq? mode 0)
      (block
       (set x scx)
       (set y (.- scy (.* v scx))))
      (block
       (set x (.+ (.* scx costheta) (.* scy sintheta)))
       (set y (.- (.* scy costheta) (.* scx sintheta))))))

 (program
  (if diag?
      (loop
            (do
              (fork
                 ((receive west ine))
                 ((send south scratch)))
              (wait 10)
              (set scratch ine))))
    (block
      (loop (decr i from (+ rowno colno colno) to 0)
            (do
              (set x scx)
              (set y scy)
              (fork ((receive west scy))
                    ((receive north scx))
                    ((send south x))
                    ((send east y)))
              (format t "~%rrr(~a,~a)rrr x: ~a    y: ~a" rowno colno scx scy)
              (wait 10)))
        (initialise self)
        (transform  self)
        (format t "~%***(~a,~a)*** x: ~a    y: ~a" rowno colno x y)
        (loop (do
              (fork ((receive west scy))
                    ((receive north scx))
                        ((send south x))
                        ((send east y)))
              (transform self)
              (wait 10))) )))

;;; The top level box for the Givens array
;;;=========================================================================
;;; Contains a column of processors for feeding inputs, a lower triangular
;;; givens array, and a row of processors to accumulate the results.
;;; Parameters:
;;; netsize :   size of the input matrix, and hence the processor arrays.
;;; mode    :   0 for linear, 1 for Euclidean norm
;;;
;;; Locals
;;; mbv     :   A triangular array of mailboxes used for communication in
;;;             the vertical direction.  Thus mailbox (i,j) becomes the
;;;             south mailbox of processor (i,j) of the Givens array,
;;;             and the north mailbox for the processor (i+1,j)
;;; mbh     :   A triangular array of mailboxes used for communication in
;;;             the horizontal direction.  Thus mailbox (i,j) becomes the
;;;             west mailbox of processor (i,j) of the Givens array,
;;;             and the east mailbox for the processor (i+1,j)
;;;
;;; Note that the diagonal processors do not have east and north  mailboxes.
;;;
;;; Boxes
;;; procs   :   The triangular array  of processors.
;;;             The array is built up element by element, after each element
;;;             is initialised to empty-elt.  The appropriate mailboxes
;;;             are passed to each processor as it is created.
;;; westcol :   This is a single process which holds the input
;;;             matrix etc. and streams it in one column at a time into the
;;;             givens array.  It passes data to the (*,0) column of the
;;;             Givens array.  The triangular mbh array is a vector of vectors,
;;;             with the first subscript indicating the number of vectors at the
;;;             top level.  Thus the 0th element of every top level vector
;;;             is obtained by mapping the function
;;;                 (lambda (v) (v:ref v 0))
;;;             onto the mbh array.
;;; southrow:   This is a single process which accumulates results
;;;             as they are generated by the Givens array.  It
;;;             communicates with the lowest row of processors in the
;;;             Givens array through their southward connections i.e.
;;;             the lowest row of the mbv array.  The final row of the
;;;             mbv array is passed as a parameter.
;;;
;;;                 These processes write/read elements sequentially to the
;;;             Givens array, but this does not affect the algorithm nor
;;;             the time because writing/reading mailboxes has no overhead.
```

```
(def-box net (netsize mode)
 (locals
  (mbv   (triangle:new netsize mailbox:new))
  (mbh   (triangle:new netsize mailbox:new)))
 (boxes
  (procs (v2:serial-build (triangle:new netsize empty-elt)
          (loop (incr i .in 0 to netsize)
                (do
                   (loop (incr j .in. 0 to i)
                         (do
                            (v2:make-elt i j
                              (proc:new
                                mode
                                i j
                                (if (eq? i j)
                                    ()
                                        (v2:ref mbv (-1+ i) j))
                                (if (eq? i j)
                                    ()
                                    (v2:ref mbh i (1+ j)))
                                (v2:ref mbv i j)
                    (v2:ref mbh i j)
                                    netsize)))))))

   (westcol   (wcol:new netsize (v:map (lambda (v) (v:ref v 0))
                                       mbh)))
   (southrow  (srow:new netsize (v:ref mbv (-1+ netsize)))) ))

;;; Processor to feed the givens array.
;;;============================================================================
;;; Has a data area to hold the input matrix.
;;; Has operations defined for initialising and displaying the
;;; data area

(def-box  wcol (size mbs)
  (locals
    (data  (v2:new empty-elt (+ size size size) size )))
  (set-state (lst)
    (loop (incr i .in 0 to (+ size size size))
          (do
             (loop (incr j .in 0 to size)
                   (do
                      (set (v2:ref data i j) (pop lst))))))
  (show ()
    (v2:map (lambda (v) (format t "~% ~a" v)) data))
  (program
    (loop (incr i .in 0 to (+ size size size))
          (do
            (loop (incr j .in 0 to size)
                  (do
                    (send (v:ref mbs j) (v2:ref data i j))))))))

;;; Processor to receive results from the givens array.
;;;============================================================================
;;; Has a data area to hold results.
;;; Has operations defined for initialising and displaying the
;;; data area

(def-box  srow (size mbs)
  (locals
    (data  (v2:new empty-elt (+ size size size) size)))
  (set-state (lst)
    (loop (incr i .in 0 to (+ size size size))
          (do
                (loop (incr j .in 0 to size)
                      (do
                         (set (v2:ref data i j) (pop lst))))))
     (show ()
       (v2:map (lambda (v) (format t "~% ~a" v)) data))
     (program
       (loop (incr i .in 0 to (+ size size size))
             (do
               (loop (incr j .in 0 to size)
                     (do
                       (receive  (v:ref mbs j)   (v2:ref data i j))))))))

;;;************************ END OF FILE ****************************************
```

The following code creates a givens array to solve a three by three system. It initialises it with the input matrix and the rhs vector and the proceeds to run the array.

```
(herald test-givens)
;;; The input matrix is skewed as it enters in the array, and hence
;;; some of the initial elements entering the array are ignored by the
;;; processors.  This is ensured by placing non numeric quantities (a,b,c)
;;; in their place.  If they indeed were not ignored, and were used
;;; in computation, it would cause an error message.

(sim:clear)
(set net (net:new 3 0))
(set-state (net:westcol net) '( 1.0  a    b
                                4.0  2.0  d
                                7.0  5.0  2.0

                               12.0  8.0  6.0
                                0.0 15.0  9.0
                                0.0  0.0 17.0

                                0.0  0.0  0.0
                                0.0  0.0  0.0
                                0.0  0.0  0.0 ))
(show (net:westcol net))
(sim:init net)
(sim:keep-ticking)

(show (net:southrow net))

;;;****************** END OF FILE **********************************************
```

Acknowledgements

The implementation of AMPS has been very greatly influenced by that of the Yale Digital Simulator [1]. Whenever possible, I have freely stolen ideas and code from that implementation. I am grateful to its authors for helping me in doing this.

I would like to thank Lennart Johnsson and Martin Schultz for using AMPS in a parallel computation course taught by them during the Spring semester, 1985. This was instrumental in exposing several bugs in the program and an earlier version of this document. For this, I am very grateful to Leslie Greengard, Venkatesh Krishnaswamy, Faisal Saied and others.

I am very grateful to Nick Carriero, Jim Philbin, Doug Baldwin and Lennart Johnsson for their careful reading of earlier drafts of this document. I feel that their detailed comments have been very useful in improving the organization and the readability.

References

[1] Doug Baldwin, Richard Kelsey, John Ruttenberg, Joseph Fisher,John Ellis.
Design and Use of the Yale Digital Simulator.
Research Report, Yale University, May, 1983.

[2] Eugene Charniak, Christopher Riesbeck, Drew McDermott.
Artificial Intelligence Programming.
Lawrence Earlbaum Associates, Publishers, 1980.

[3] D. Gelernter.
Generative Communication In Linda.
ACM TOPLAS 7-1:80-112, January, 1985.

[4] P.B. Hansen.
Distributed Processes: A Concurrent Programming Concept.
CACM 21-11:934-941, November, 1978.

[5] C.A.R. Hoare.
Communicating Sequential Processes.
CACM 21-8:666-677, August, 1978.

[6] Henry Ledgard.
Ada, An Introduction.
Springer-Verlag, 1983.

[7] Jonathan Rees, Norman Adams, James Meehan.
The T Manual
Yale University, 1983.

[8] Guy Steele.
RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization).
Technical Report, MIT, May, 1978.

[9] Mitchell Wand.
Continuation-based Multiprocessing.
In *Proc. 1980 Lisp Conference*, pages 19-28. 1980.