OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

# Software Development in Parallax: The ELGDF Language

Hesham El-Rewini
Ted Lewis
Department of Computer Science
Oregon State University
Corvallis, OR 97331

88-60-17

# ELGDF: Design Language for Parallel Programming

Hesham El-Rewini & Ted Lewis

Department of Computer Science
Oregon State University
Corvallis, OR 97331
(503) 754-3273

## ABSTRACT

*ELGDF (Extended Large Grain Data Flow)* is a design language that allows representation of a wide variety of parallel programs. The syntax is graphical and hierarchical to allow construction and viewing of realistically sized programs. ELGDF language facilitates describing parallel programs in a natural way for both shared memory model as well as message passing model. The syntax poses high level structures such as replicators, loops , pipes, branches, and fans, as well as constructs for shared resources. ELGDF resolves arc overloading in current graphical languages by using different symbols and different attributes for different types of arcs.

The ELGDF serves as the foundation of a parallel programming environment under development at Oregon State University. The complete syntax of ELGDF helps the program designers to deal with parallelism in the manner most natural to the problem at hand. It also helps as a way to capture parallel program designs for the purpose of analysis such as scheduling and performance estimating. Thus, the goal of ELGDF is two-fold: 1) a program design notation and computer-aided software engineering tool, and 2) a software description notation for use by automated schedulers and performance analyzers.

## INTRODUCTION

It seems clear that the next generation of computers will be based on the multiprocessor paradigm, but more effort is needed to help software engineers develop programs for parallel computers.

Because humans tend to think sequentially rather than concurrently, program development is most naturally done in a sequential language [19]. Unfortunately sequential programming is incapable of directly making effective use of parallel computers.

If we look at the evolution of sequential programming, we find that sequential programming has evolved in the following way: at the beginning all the programs were written in architecture-specific low level languages. Then high level languages started to appear allowing programs to be written in architecture independent languages so the programmers didn't have to worry about the architectural details. Finally extensions have been made to high level languages to make them more structured and abstract leading to programs that are easier to develop, test, and maintain.

We believe that parallel programming should evolve in the same direction. Developing hand-coded parallel programs is equivalent, in a sense, to programming in a low level sequential language, because hand-coded parallel programs are quite architecture dependent. For example synchronization is done using locks in a shared memory architecture, but synchronization is done via message passing in a distributed memory architecture.

In order to develop hand-coded programs for parallel systems, the programmer has to exploit the potential concurrency of the algorithm, write the parallel program for a given architecture using a language and synchronization constructs suitable for the given architecture, schedule tasks on the available processors using intuitive methods, execute the program, and finally debug the program if it doesn't give the expected results or if it goes into a deadlock situation. Programmers have a great deal of details to worry about at any time which makes parallel programming a very difficult process. In order to make parallel programming easy, we need to get the system to shoulder more of the burden.

Given these facts, it is not surprising that an architecture independent higher abstraction is needed so program designers can express their algorithms in high level structures without having to worry about the details like the synchronization code. High level parallel programs then can be analyzed and translated into schedulable units of computation that fit the target hardware architecture.

In this paper we describe the *ELGDF* design language that allows program designers to express parallel program designs in a graphical and hierarchical syntax. Graphs can be used to represent the potential concurrency in programs [15]. We beleive that the *ELGDF* will help programmer comprehension and will produce parallel program designs in a form appropriate for analysis.

This work is related to a number of other studies [1,2,5,6,11,15]. In our work the program is represented as a large grain data flow network. Our work extends LGDF [2,6,11] to facilitate the following: 1)ELGDF syntax poses high level structures such as replicators, loops, pipes, etc., 2) ELGDF allows the designer to express branches and loops that give more information for scheduling and analysis purposes, 3) ELGDF expresses parameterized constructs compactly,4) ELGDF has arc hierarchy as well as node hierarchy, 5) ELGDF resolves arc overloading by providing different symbols and different attributes for different types of arcs, 6) ELGDF allows the user to express mutual exclusion easily , 7) ELGDF provides synchronized pipelining through repeated arcs, and 8) ELGDF can be easily transformed into a precedence task graph which is useful for task scheduling and processor allocation.

## THE LANGUAGE

ELGDF is a graphical language for designing parallel programs. ELGDF designs can be refined into Pascal, C, FORTRAN, etc. source code programs through simple transformation. However, ELGDF is designed as a program design language, and not a programming language. ELGDF is rich enough to express many of the common structures found in parallel programs (replicators, pipes, branches, fans, etc.). An ELGDF design takes the form of a directed network consisting of nodes, structures, storage constructs, parameterized constructs, and arcs.

## BASIC CONSTRUCTS

### 1-Nodes

A node, as shown in FIG-1 is represented by a "bubble", and can represent either a simple or a compound node. A simple node consists of sequentially executed code and is carried out by at most one processor. A compound node is a decomposable high level abstraction of a subnetwork of the program design network.

A node TASK-1 


FIG-1

### 2- Storage Constructs

A storage construct, as shown in FIG-2 is represented by a rectangle, and can represent either a storage cell or a collection of storage cells. A storage cell represents the data structure to be read or written by a simple node. A node connected to the top of a storage construct has access to it before any node connected to its bottom. Nodes connected to a storage construct on the same side (top/bottom) compete to gain access to that storage construct in any order. A compound node connected to the left or the right sides of a rectangle representing a collection of storage cells means that the compound node will be using the constituents of the storage collection and the details will be given in lower levels description.

A sorage construct
INPUT-1


FIG-2

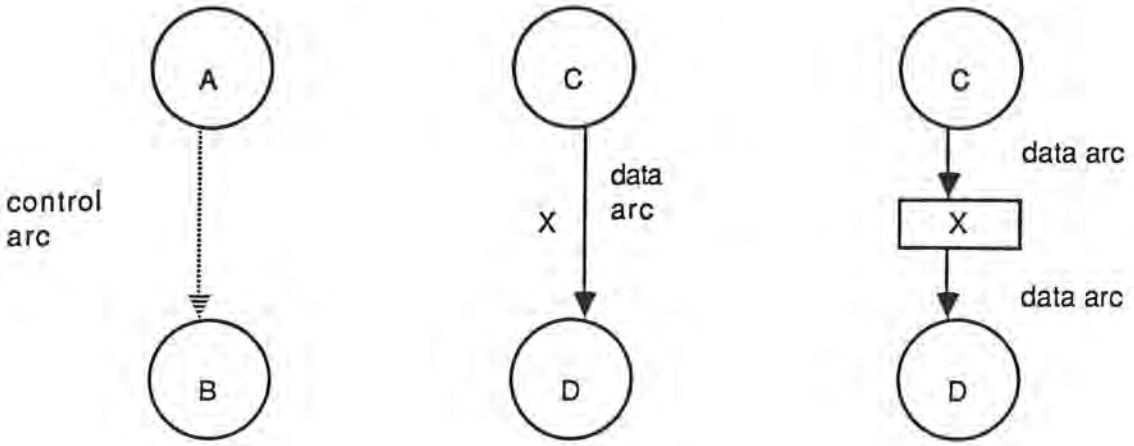
### 3- Arcs

An arc in ELGDF can express either data dependency, sequencing, transfer of control, or read and/or write access to a storage construct. A set of attributes is associated with each arc to provide information about the arc type, data to be passed through the arc, storage access policy, communication strategy, and others.

An arc can be either a simple arc which cannot be decomposed or a compound arc which is decomposable into a set of other simple and/or compound arcs. Simple arcs can be classified into control and data arcs. FIG-6 shows simple arcs classification.

A control arc expresses sequencing and transfer of control among nodes. FIG-3-a shows a control arc connecting node A to node B. Thus, B cannot start execution before A finishes and A should activate B once it is done. A control arc going from compound node C to compound node D is actually a set of control arcs going from the constituents of C to those of D to make sure that every node in C must finish before any node in D can start. Two or more control arcs can meet at a connection point and become one control arc. A control arc leaving a connection point gets activated when all the control arcs coming into the connection point get activated.

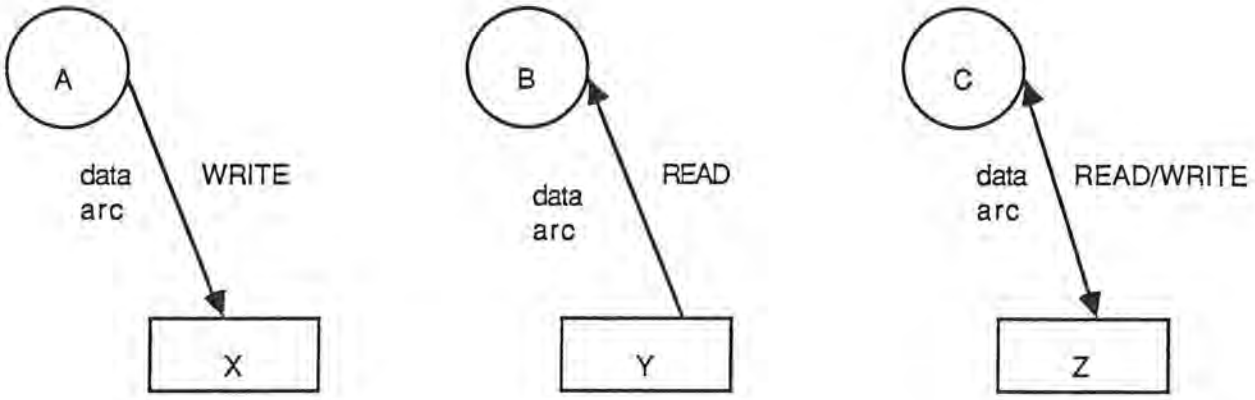
A data arc can carry data from one node to another or can connect a node to a storage construct. FIG-3-b shows a data arc connecting node C to node D which means that node B cannot start execution until it gets the data structure (X) associated with the arc from node A. The data arc connecting nodes C and D can be thought of as a channel carrying (X) in a message passing system. However, it means that C writes to the storage construct (X), then D reads from (X), as shown in FIG-3-c, in a shared memory system. A data arc which is not connected at one end represents a connection.




(a) (b) (c)

FIG-3

A data arc connecting a node and a storage construct can represent READ, WRITE, or READ/WRITE access depending on the direction of the arc. An arc going to a storage construct means WRITE, an arc leaving a storage construct means READ. A bi-directional arc means READ/WRITE. A READ/WRITE compound arc means its constituents can be READ, WRITE, or READ/WRITE arcs. A simple READ/WRITE data arc connecting a simple node to a storage cell means the node reads and then writes from and to the storage cell. See FIG-4.




Simple node A writes into storage cell X

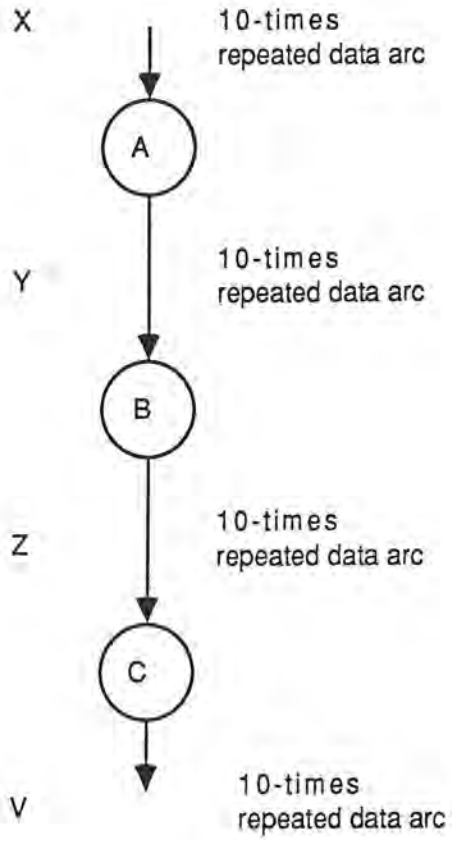
Simple node B reads from storage cell Y

Simple node C reads from storage cell Z and then writes into it

FIG-4

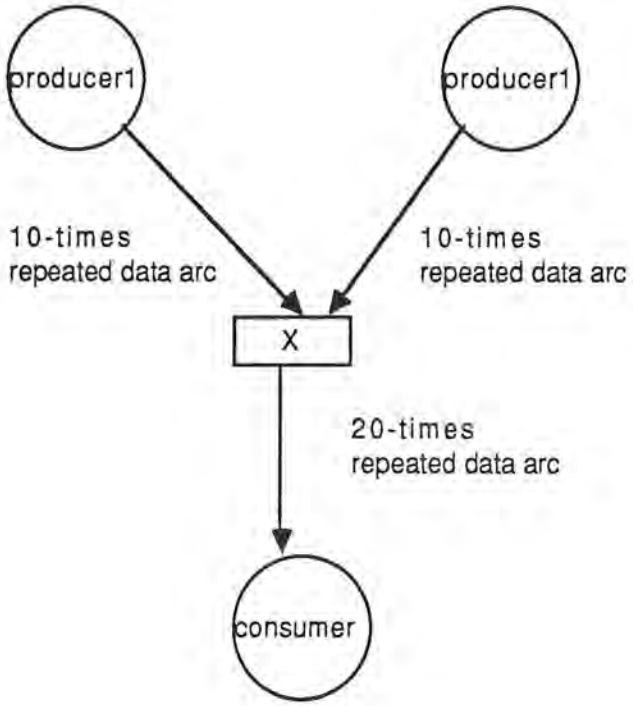
A data arc can be used to carry data once or repeated times per activation. One of the arc's attributes is used to indicate the number of times the data will be passed through. If the value of that attribute is greater than one then the arc is considered a repeated arc.

The repeated arc is used basically in pipelines. It can carry data (repeated times) from a simple node to another in a synchronized fashion. Also it can express synchronized writing and reading to or from a storage cell. FIG-5-a shows three simple nodes A, B, and C forming a pipeline, the three simple nodes can run concurrently as a pipeline. Node A gets X, processes it, sends Y to B, gets another X (now A and B can process their input concurrently) and so on. Using this kind of arc, A cannot send another Y until B consumes the first one and so on. FIG-5-b shows two producers and one consumer problem. The two producers are competing because they are on the same side of X. One producer cannot overwrite the value produced by the other until the consumer consumes it.




A pipeline of 3 stages

(a)




Two producers and one consumer (20 times)

(b)

FIG-5

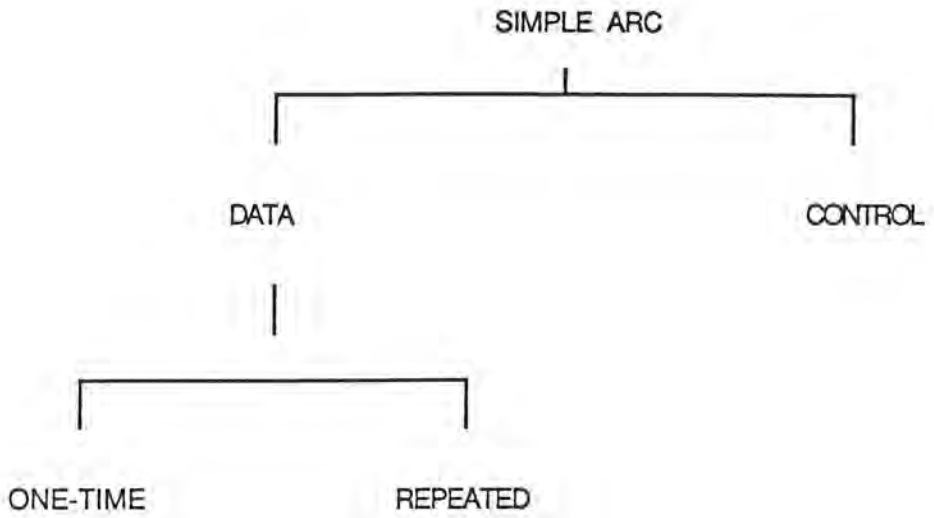
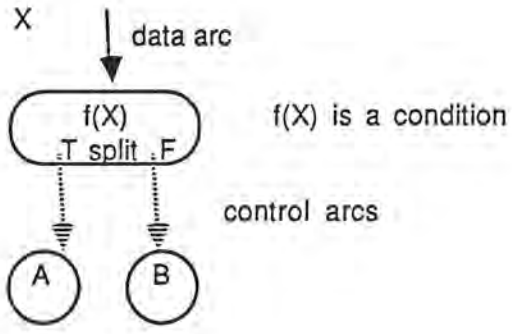



FIG-6 (simple arcs classification)

## 4- Split and Merge

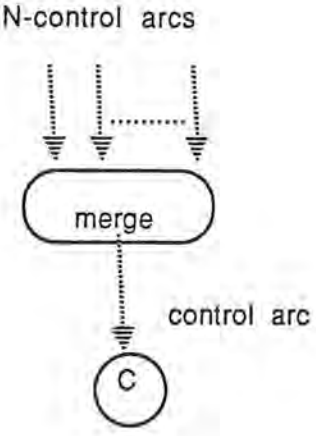
Split and merge are special purpose simple nodes for representing conditional branching. Split has one input data-arc, two output control-arcs; one for T = True, and the other for F = False. According to the truth or the falsehood of the condition associated with the split node one of the two control arcs coming out of it is activated as shown in FIG-7-a. Merge has N input control arcs and one output control arc. Merge activates its output arc when it gets activated by any one of its N inputs. See FIG-7-b.




node A gets activated if f(X) is TRUE

node B gets evaluated if f(X) is FALSE

(a)




node C gets activated when any of the N input control arcs activates merge.

(b)

FIG-7

## 5- Loops

A loop can represent FOR, WHILE, or REPEAT. In this section we explain how these three structures can be represented in ELGDF. Figures 8, 9, 11 show the loop symbols and their semantics.

i) For Loops

A for loop is one of the parameterized constructs in ELGDF that allows program designers express loops compactly without having cycles in the graph. A set of attributes is associated with the loop such as the control variable, initial value, step, loop bound, and others. A loop over a node for N times, produces a sequence of N instances of that node connected by N-1 control arcs. FIG-8-a shows looping over a node P(i), [i = 1 to 3]. FIG-8-b shows the unrolled loop.

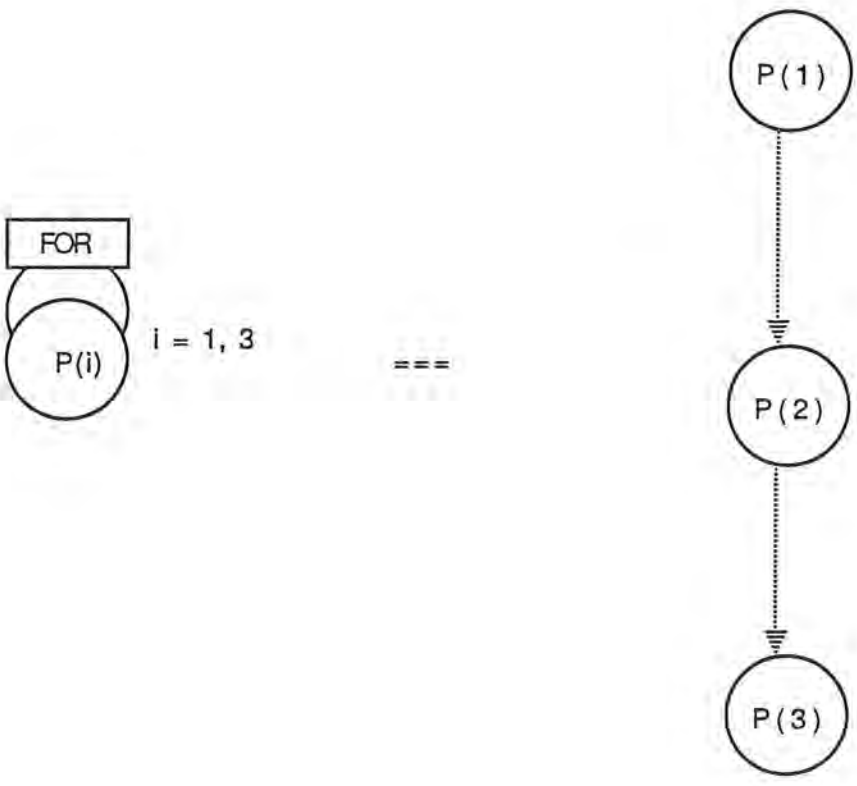



FIG-8

ii) While loop

A while loop structure helps program designers express while loops compactly. A program designer only defines the node representing the body of the while structure and the while predicate. For analysis purposes, the system deals with while structures using branches. As long as the while predicate evaluates to true, a new instance of the node representing the while body is generated. FIG-9 shows the semantics of the while structure ,in terms of split, merge, node and while construct. The graph in FIG-9-a means: while f(X) do P.0; and The graph in FIG-9-b means: if f(X) then { P.O; while f(X) do P.1;};. (P.1 is another instance of P.0). FIG-10-a shows the while construct in FIG-8-a when the condition f(X) evaluates to True twice and then evaluates to False. FIG-10-b shows the same graph in FIG-10-a after replacing the three merge nodes with one merge.

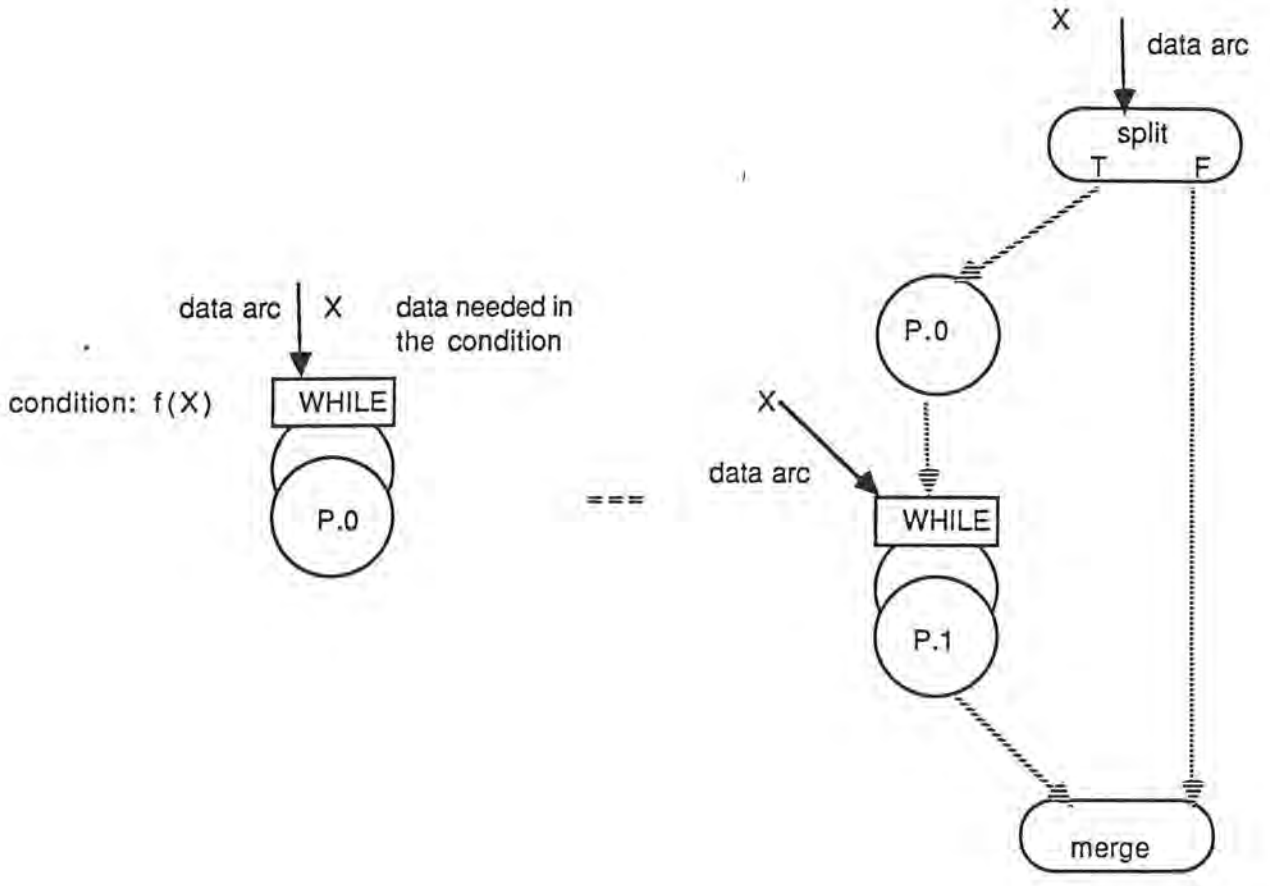



FIG-9

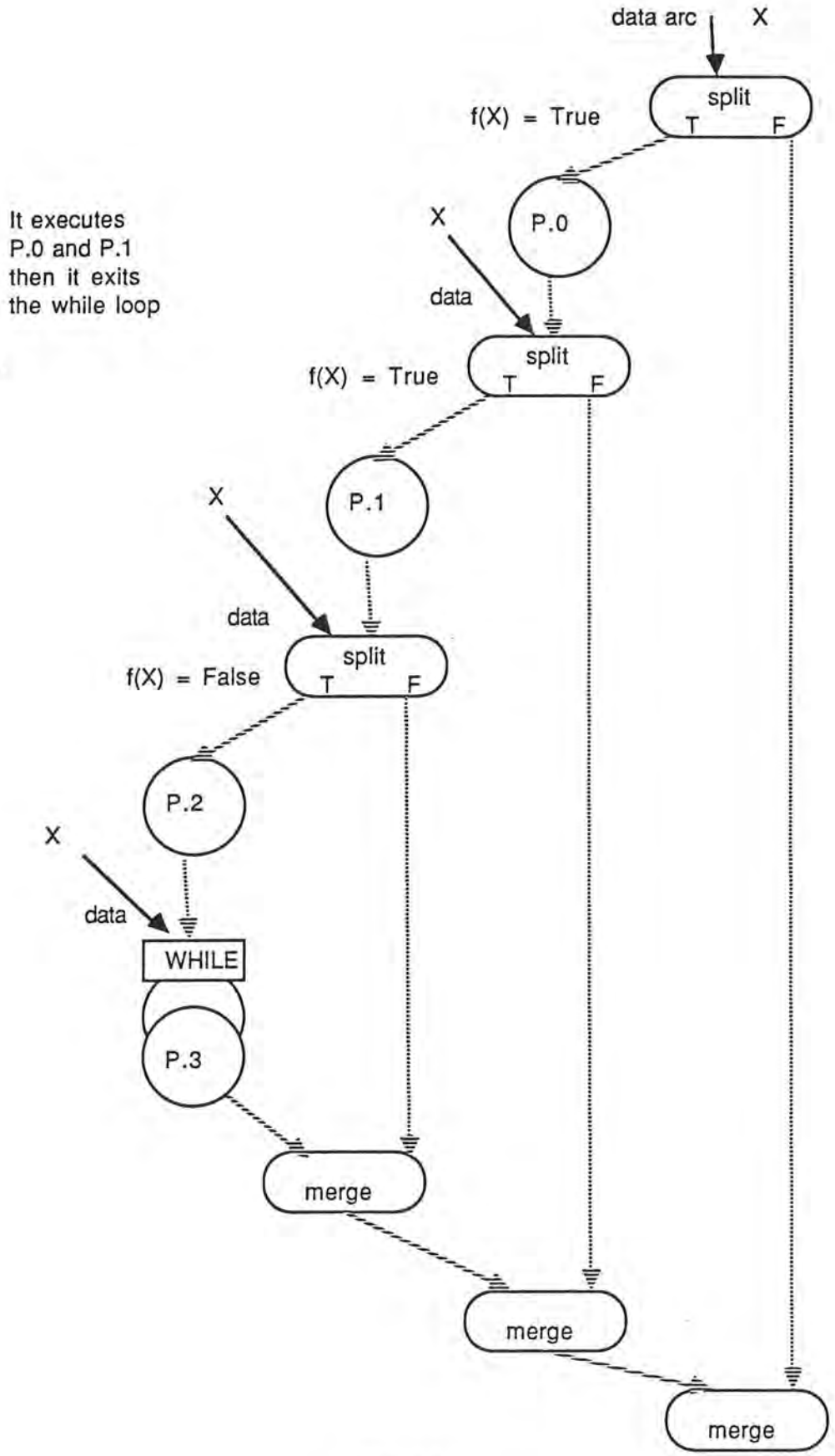

data arc
X
split
T
F
f(X) = True
It executes
P.0 and P.1
then it exits
the while loop
X
P.0
data
split
T
F
f(X) = True
P.1
X
data
split
T
F
f(X) = False
P.2
X
data
WHILE
P.3
merge
merge
merge

FIG-10-a

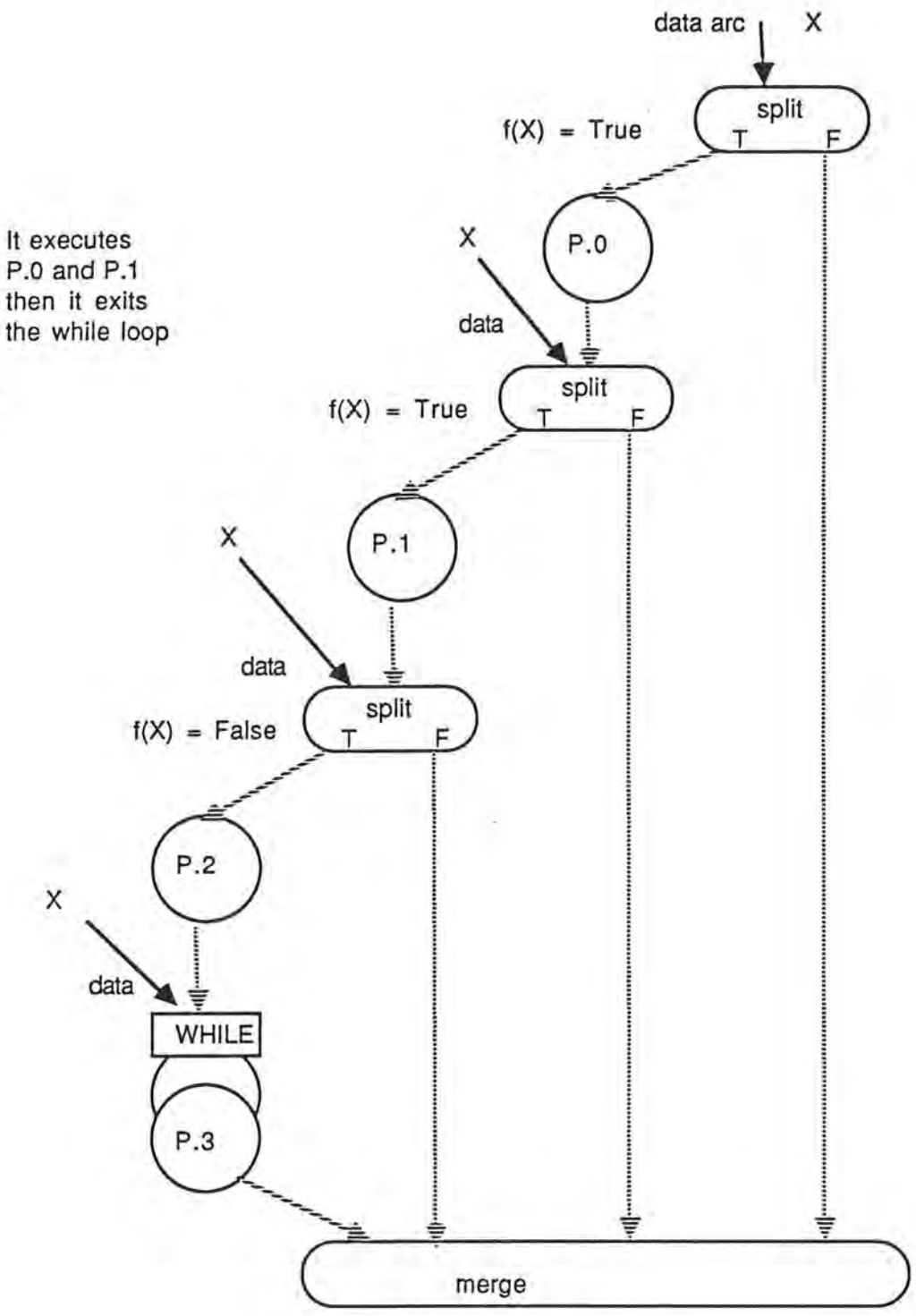

data arc
X
split
T
F
f(X) = True
It executes
P.0 and P.1
then it exits
the while loop
X
P.0
data
split
T
F
f(X) = True
X
P.1
data
split
T
F
f(X) = False
P.2
X
data
WHILE
P.3
merge

FIG-10-b

iii) Repeat-Until loop

Repeat-until structure is similar to while structure explained above except that the predicate comes after the repeat-until body. A new instance of the node is generated until the condition evaluates to true. FIG-11 shows the semantics of the repeat-until structure, in terms of split, merge, a node and repeat until construct. The graph in FIG-11-a means: repeat P.0 until f(X); and The graph in FIG-11-b means: P.0; if f(X) then { repeat P.1 until f(X)}. (P.1 is another instance of P.0). FIG-12-a shows the repeat until construct in FIG-11-a when the condition f(X) evaluates to False once and then evaluates to True. FIG-12-b shows the graph in FIG-12-a with replacing the two merge nodes with one merge.

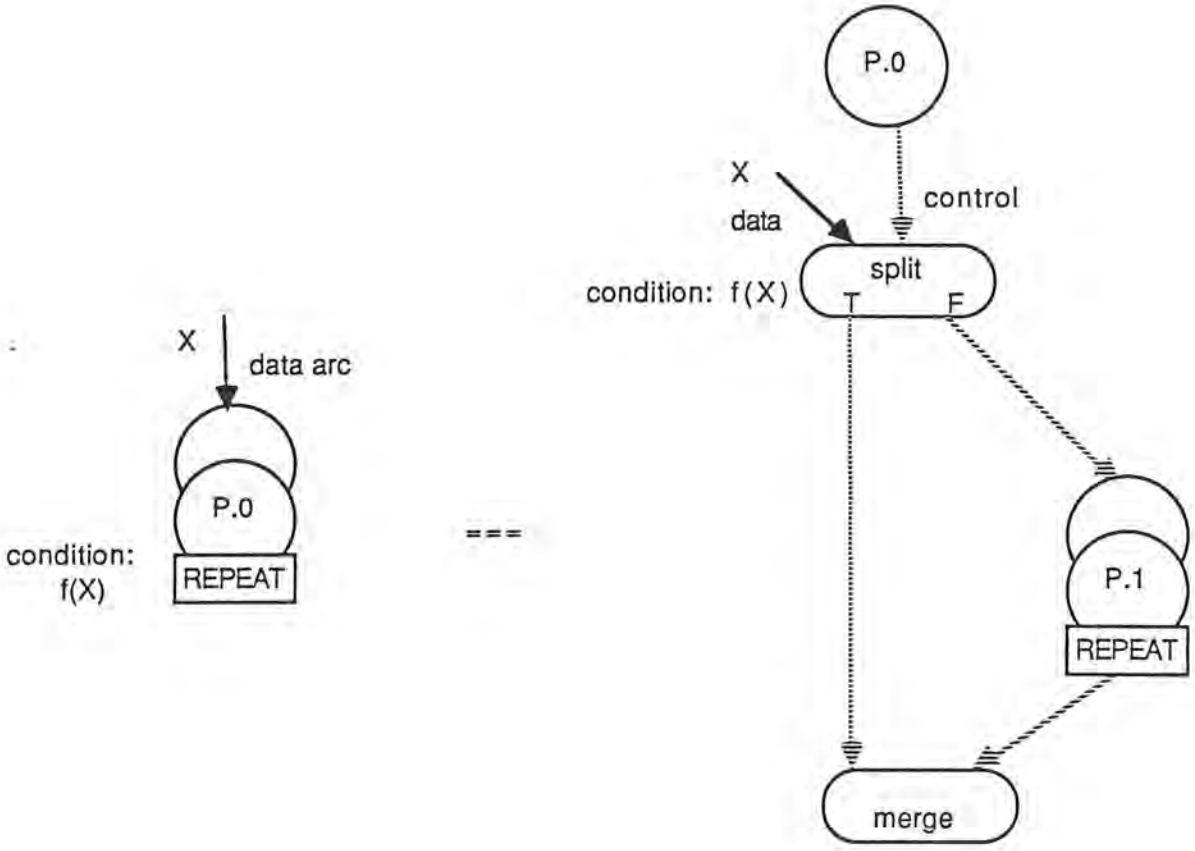



FIG-11

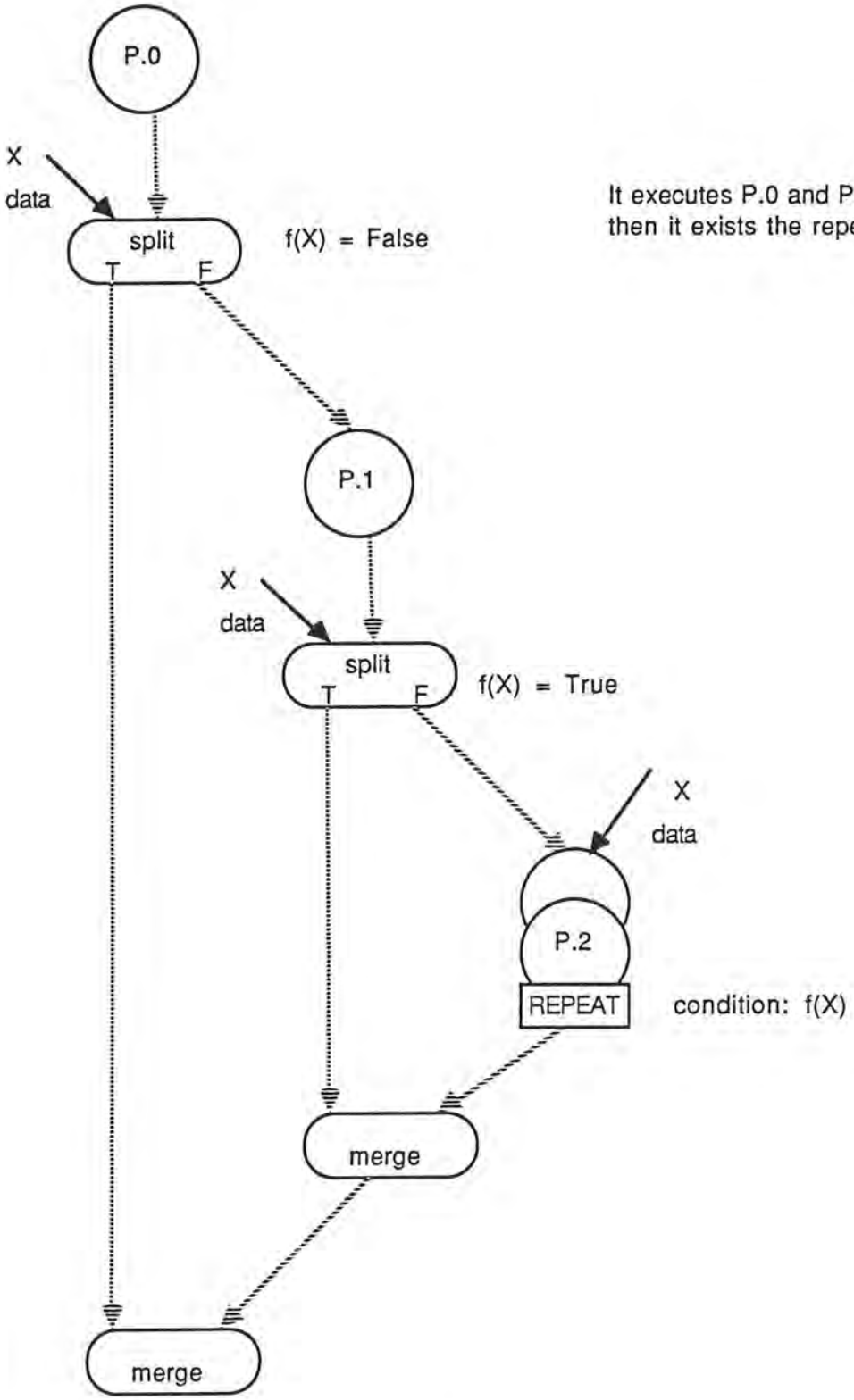

P.0
X
data
split
T
F
f(X) = False
It executes P.0 and P.1
then it exists the repeat loop
P.1
X
data
split
T
F
f(X) = True
X
data
P.2
REPEAT
condition: f(X)
merge
merge

FIG-12-a

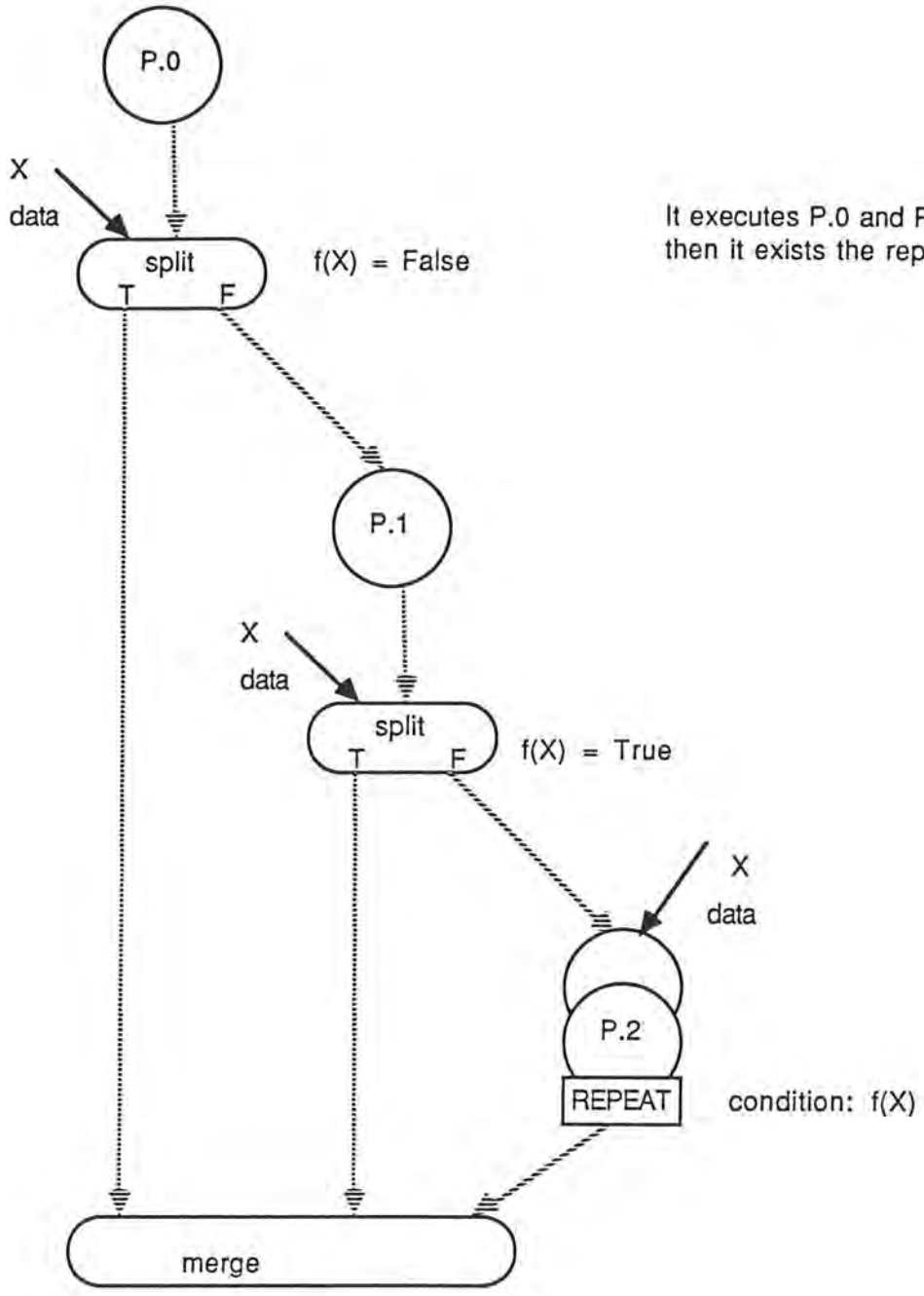

P.0
X
data
split
T
F
f(X) = False
It executes P.0 and P.1
then it exists the repeat loop
P.1
X
data
split
T
F
f(X) = True
X
data
P.2
REPEAT
condition: f(X)
merge

FIG-12-b

### 6) Replicators

A replicator is one of the parameterized constructs in ELGDF that allows program designers to represent concurrent loop iterations compactly. A set of attributes is associated with the replicator such as the control variable, initial value, step, replicator bound, and others. Replication of a node N times , produces N concurrent instances of that node. FIG-13-a shows a replicator over a node P(i) (i = 1 to 3) and its expansion. An arc connected to a replicator is expanded as a set of identical arcs each of which is connected to one of the replicated instances. See FIG-13-b.


FIG-13-a

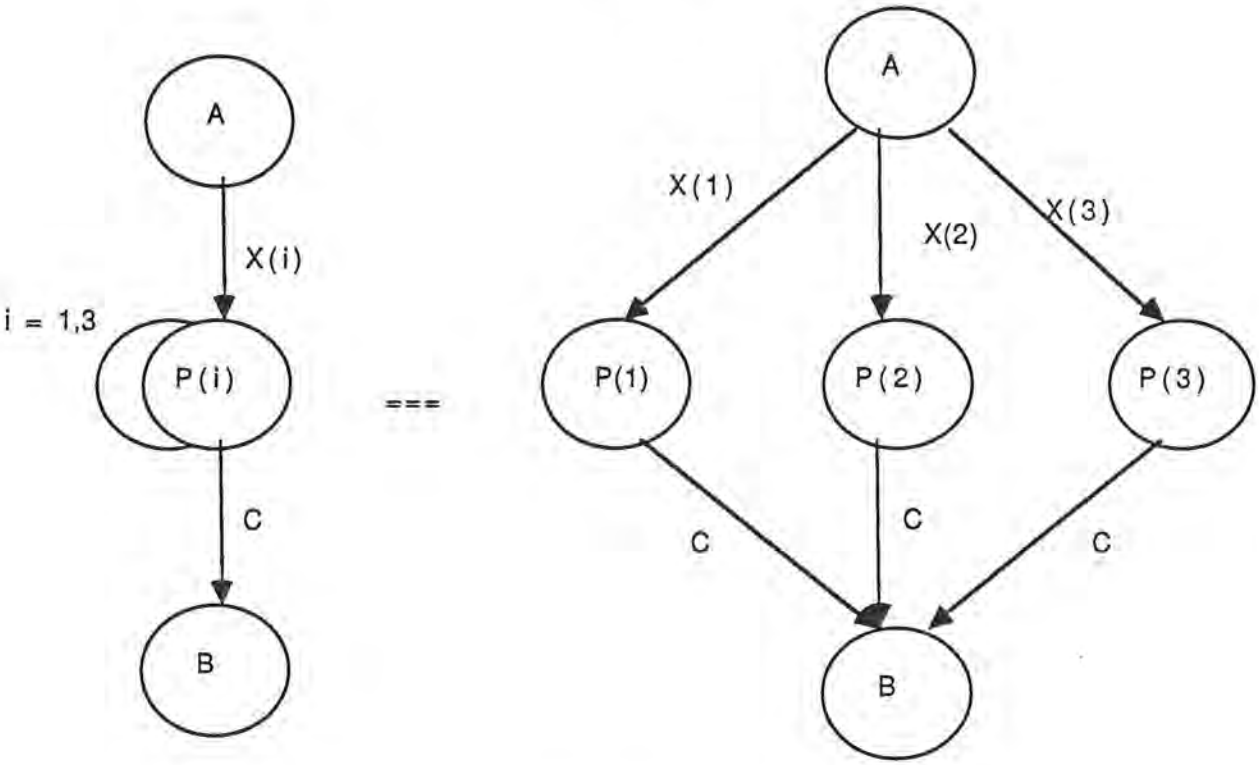



FIG-13-b

### 7) Homogeneous Pipes

A homogeneous pipe is a high level abstraction that allows program designers to represent a set of N nodes forming a pipeline compactly. The pipe consists of N simple nodes and N-1 m-repeated arcs. The nodes forming the pipeline are replications of the same simple node. A pipe has several attributes associated with it such as number of stages in the pipeline (N), number of times the data will be passed through repeated arcs in the pipe (m) and others. A pipe of N stages of node p(i) is shown in FIG-14-a and its expansion is given in FIG-14-b.

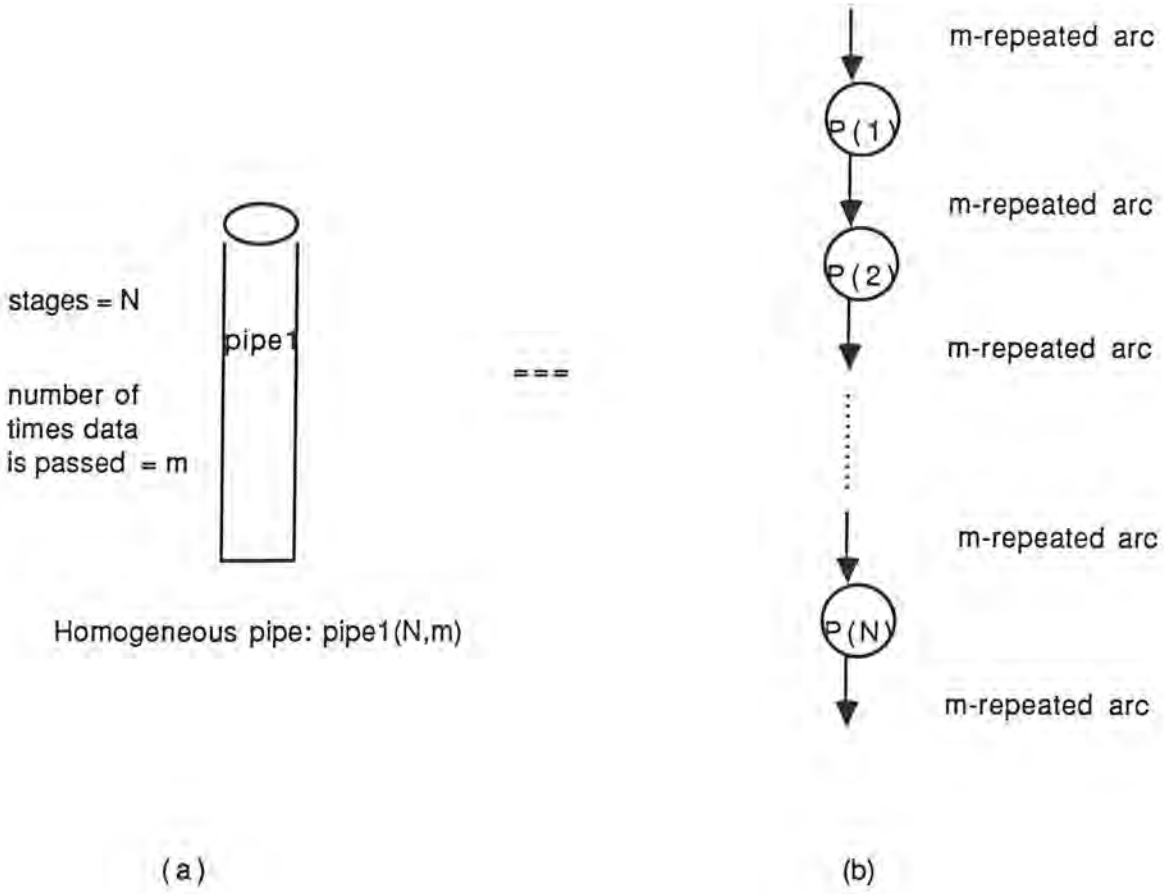



FIG-14 (pipe pipe1 (N,m))

## CONVENIENT STRUCTURES

### i) IF-THEN-ELSE

IF-THAN-ELSE is one of the convenient structures in ELGDF, in which the designer uses the symbol given in FIG-15-a and then he/she can define the two alternatives and the condition later. That will reduce drawing time and will help design readability. An IF-THEN-ELSE structure is composed of a split node connected to the two alternatives of the IF-THEN-ELSE by control arcs. Then those two alternatives are connected to a merge node by control arcs at the end point of the IF. Compound arcs that are connected to the body of IF-THEN-ELSE carries data to or from the two alternatives within the structure. FIG-15-b shows the decomposition of an IF-THEN-ELSE structure with two alternatives: node A if the condition evaluates to True and node B if the condition evaluates to False.

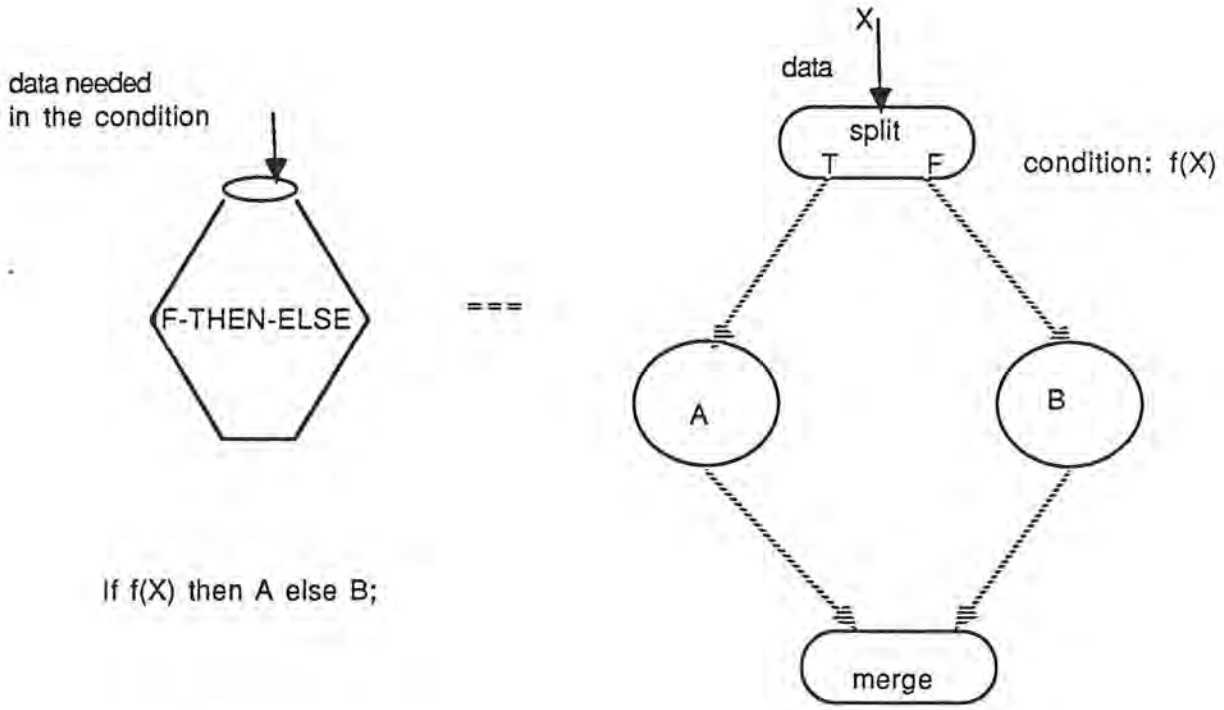
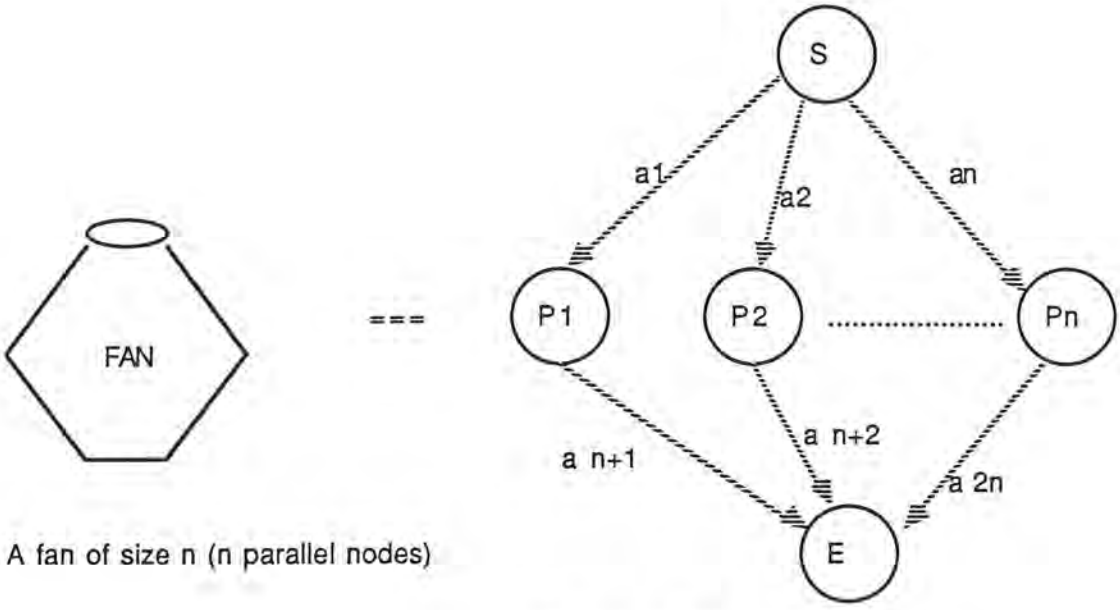



FIG-15

ii) Fan

Fan is another form of convenient structures in ELGDF, in which the designer can use the skeleton prepared as fan and then define its constituents later. That will help design readability and will reduce drawing time. A fan of size N is composed of a start node (S), N parallel nodes $P_i$ ,i = [1..N], 2N control arcs $a_j$, j = [1..2N], and an end node (E). Arc $a_j$ is connecting S to $P_j$, j = [1..N]. Arc $a_k$ is connecting $P_{k-N}$ to E, k = [N+1..2N]. The start node activates the parallel nodes and when they all finish E gets activated. Compound arcs that are connected to the body of Fan carries data to or from its constituents. FIG-16 shows a fan of size n and its expansion.




A fan of size n (n parallel nodes)

ai , i = [1..2n] are control arcs

S activates P1..Pn

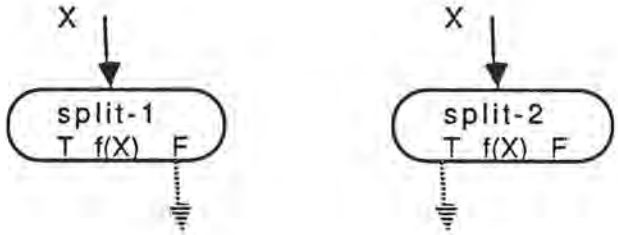
E start execution after (P1..Pn) finish

FIG-16

DEFINITIONS

**Semi-Sink Split:**

A split is called semi-sink if it has one of its two connection points (T & F) not connected. See FIG-17.




semi-sink split

'T' connection point is unconnected

semi-sink split

'F' connection point is unconnected

FIG-17

**Source node:**

A node is called a source if it does not have any predecessor nodes. See FIG-18.

**Local source node (merge):**

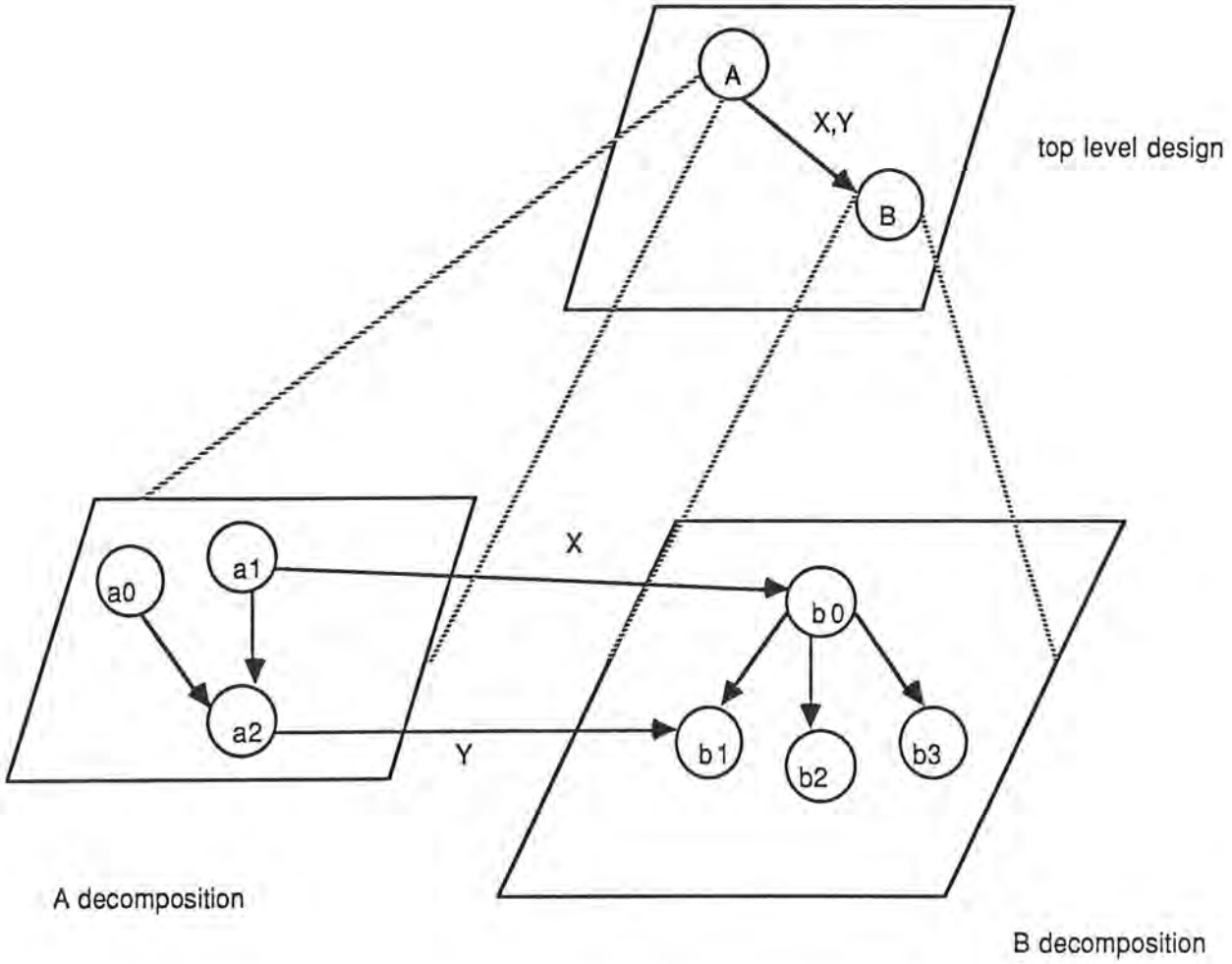
A node (merge) in a compound node is called a local source if it does not have any predecessor nodes within that compound node. See FIG-18

**Sink node (merge):**

A node (merge) is called a sink if it does not have any successor nodes. See FIG-18

**Local sink node:**

A node in a compound node is called a local sink if it does not have any successor nodes within that compound node. See FIG-18




**Source nodes : A, a0,a1**

**Local source nodes : b0**

**Sink nodes: B, b1, b2, b3**

**Local sink nodes : a2**

FIG-18

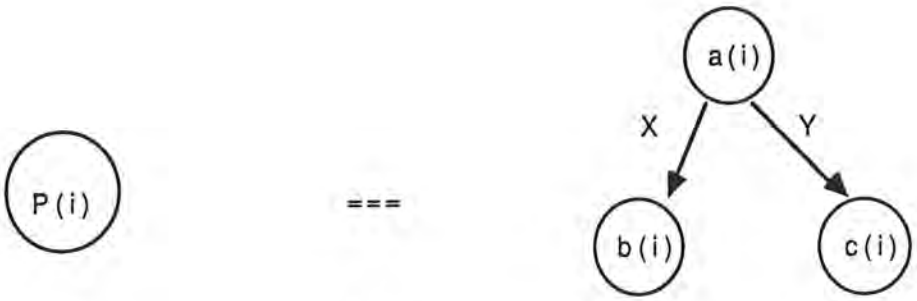
## COMPOUND NODES CONNECTION

i) Using compound arcs

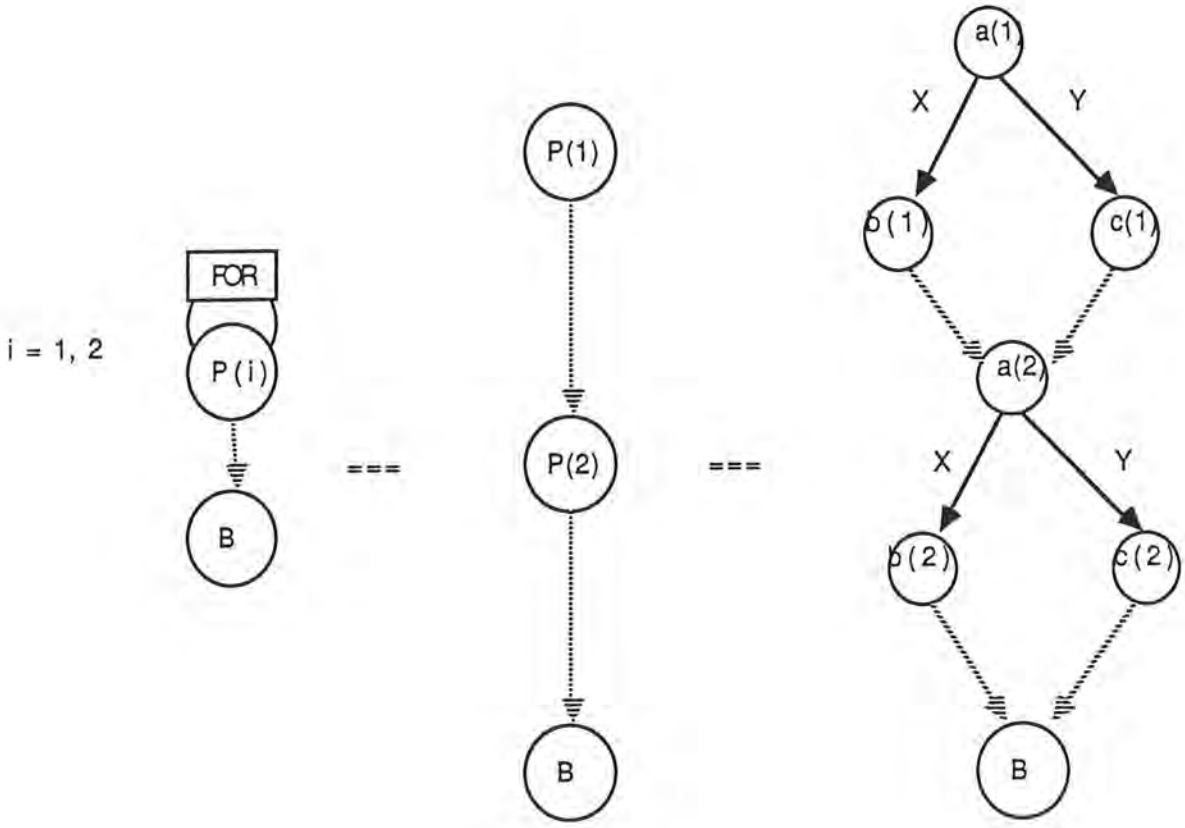
FIG-18 shows a compound arc going from compound node A to compound node B having X, Y as data structure associated with it. Thus, some constituents in B are data dependent on some constituents in A and the data involved are X, Y. In the lower level decomposition of A and B, node b0 needs X from node a0 and node b1 needs Y from node a2.

ii) Using control arc

A control arc going from compound node A to compound node B means that all the constituents of A must finish before any of B's constituents can start execution. In the lower decomposition of A and B, there must be a control arc going from any sink, local sink, or the unconnected connection point of semi-sink split nodes in A to every source, or local source nodes in B. For example FIG-19-a shows a compound node P(i) and its decomposition. FIG-19-b shows a network that has a loop over the compound node P(i) and its expansion. Notice the control arcs from b(1) (sink) to a(2) (source), c(1) (sink) to a(2) (source), b(2) (sink) to B and c(2)(sink) to B.




(a)



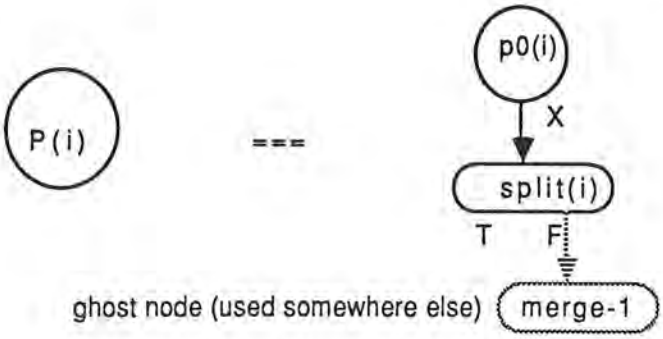

(b)

FIG-19

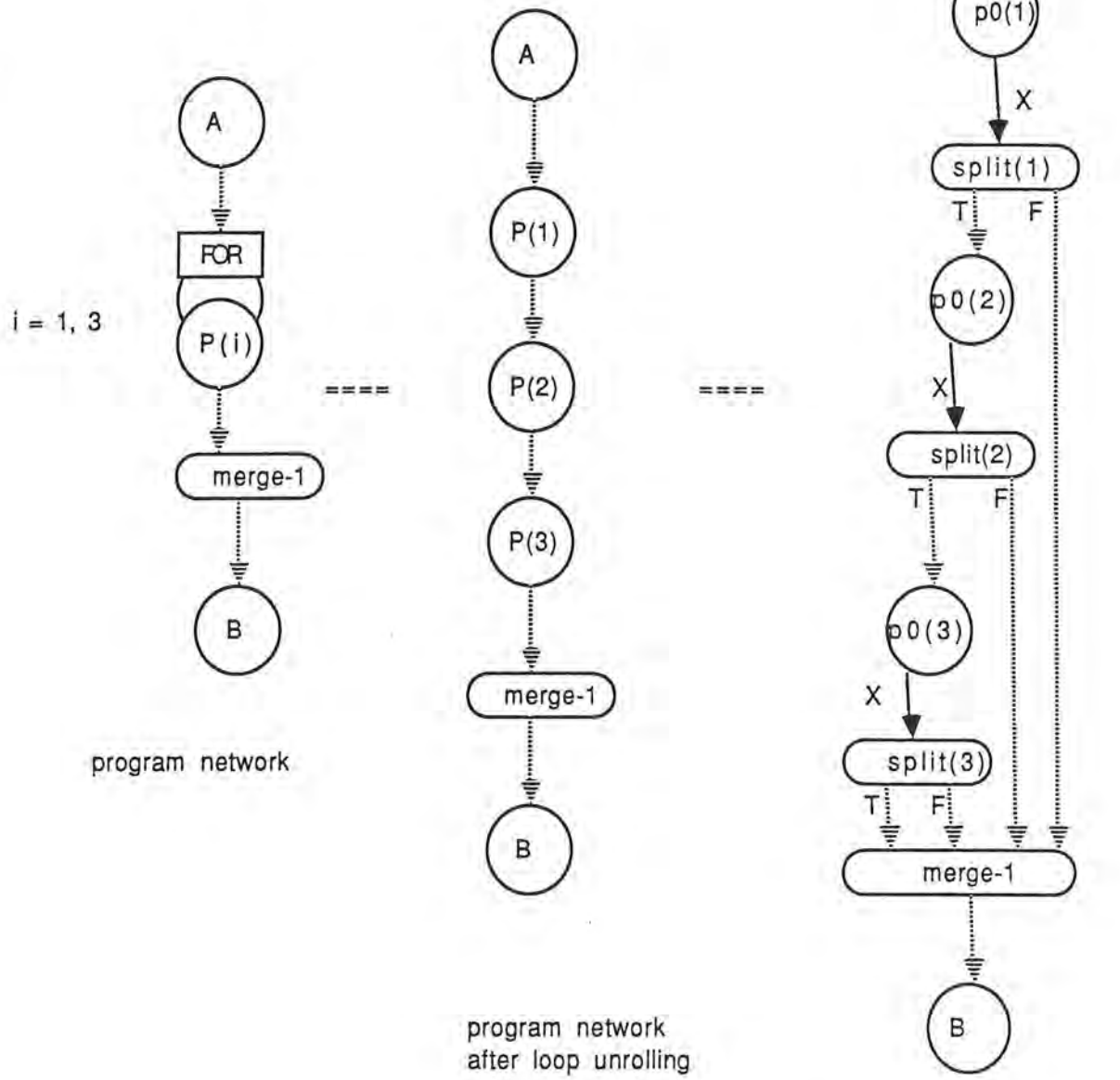
## TRANSFER OF CONTROL AND USING GHOST MERGE

A ghost merge is a merge that is used else where in the network and we use it to express transfer of control such as exit from a loop. FIG-20-a shows a compound node P(i) and its expansion that includes a ghost merge (merge-1 which is already used in the program network). FIG-20-b shows the program network which has a loop over P(i) and its expansion. The loop expansion shows the four different paths in the program network according to the truth or the falsehood of the condition associated with the split node at each loop iteration.

FIG-21 shows another example of transfer of control. The top level design has three nodes B, C, and D. Nodes B and D are compound nodes and C is a simple one. Two control arcs are going from B to C and from C to D because at that level we know that B must finish before C and C must finish before D. The lower level expansion of B shows a transfer of control to D if the condition associated with the split evaluates to False. The description of A should have a ghost merge to show transfer of control to merge-d.




(a)




program network

program network
after loop unrolling

low level expansion of
the program network

(b)

FIG-20

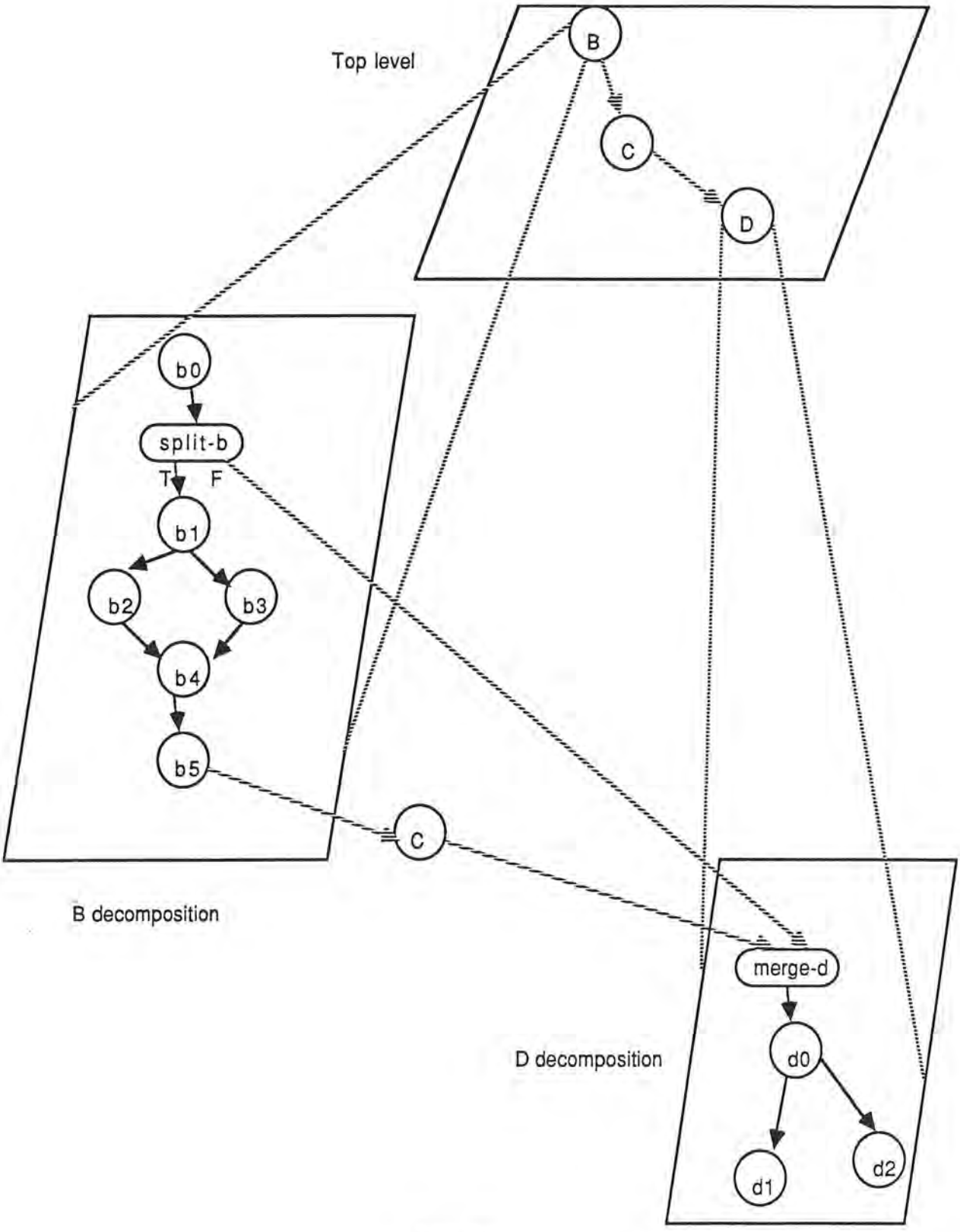

Top level
B
C
D
b0
split-b
T
F
b1
b2
b3
b4
b5
C
B decomposition
merge-d
d0
d1
d2
D decomposition

FIG-21

## MUTUAL EXCLUSION IN ELGDF

ELGDF helps users to easily express mutual exclusive access to shared variables by having an attribute associated with each arc connecting a node to a storage construct. If the exclusion attribute is set to ON , then mutual exclusion will be guaranteed. FIG-22 shows three simple nodes A, B, and C and a storge cell X forming an ELGDF network. Nodes A, B, and C share the variable X, yet A and B will have access to X before C because A and B are connected to the top of X and C is connected to the bottom. A and B can access X in any order since they are both in the same side of X. Both A and B want to update X through a READ/WRITE arc and that might produce incorrect result unless we set the exc (mutual exclusion) attribute associated with those READ/WRITE arcs to guarantee mutual exclusive access to X as shown in the figure.

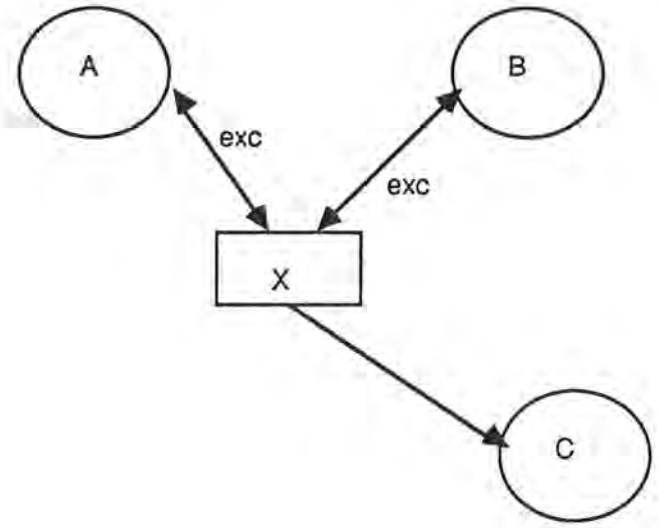
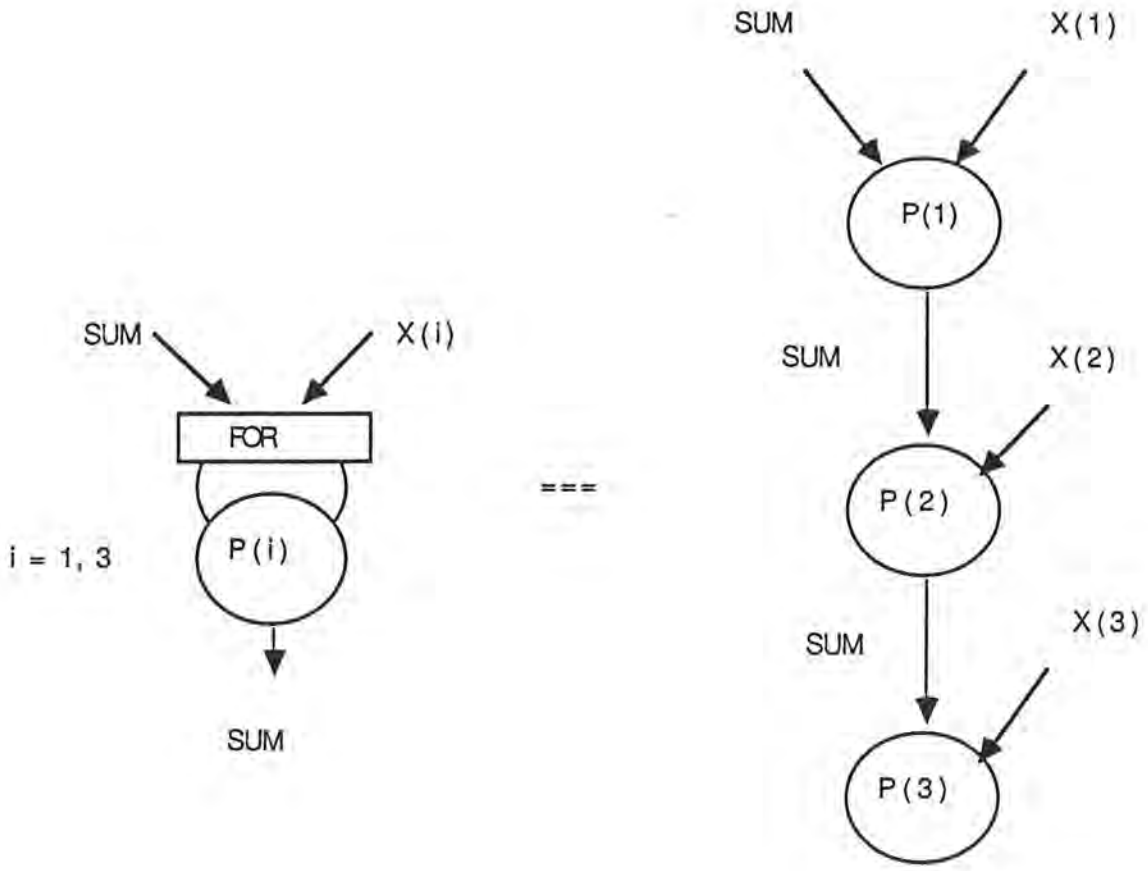



FIG-22

## LOOP UNROLLING WITH DATA FLOW IN ELGDF

ELGDF helps designers express their loops compactly for analysis purposes. FIG-23-a shows a process P(i) that takes two inputs SUM and X(i) and produces one output SUM to be run in a loop with loop bound = 3. FIG-23-b shows the unfolding of the loop and how an iteration of the loop can use the result of the previous iteration.




(a) (b)

FIG-23

## PRECEDENCE GRAPH

A precedence graph of a parallel program can easily be obtained from its ELGDF design at different levels of granularity for use by automated schedulers. A precedence graph is a directed graph G (V, S, M, E) where V is a set of vertices, S is a set of split nodes, M is a set of merge nodes, and E is a set of edges. The set of vertices at certain level consists of all the ordinary nodes in the ELGDF network at that level. The set of split nodes at certain level consists of all the split nodes in the ELGDF network at that level. Similarly the set of merge nodes at certain level is the set of all the merge nodes in the ELGDF network at that level. The set of edges consists of all the control arcs connecting split and merge nodes to ordinary nodes and all edges Vi->Vj, if any of the following conditions hold in the ELGDF program network:

o There is a control arc going from node Vi to node Vj.
o There is a data arc going from node Vi to node Vj.
o There exist a storage construct (Q) such that Vi is connected to the top of Q and Vj is connected to the bottom of it.

A precedence graph can be represented compactly using parameterized nodes (replicators and loops).

FIG-24 shows an ELGDF design network (the top level) and the decomposition of some of its constituents. FIG-25 (a through c) shows the precedence graphs of the program at different levels of granularity.

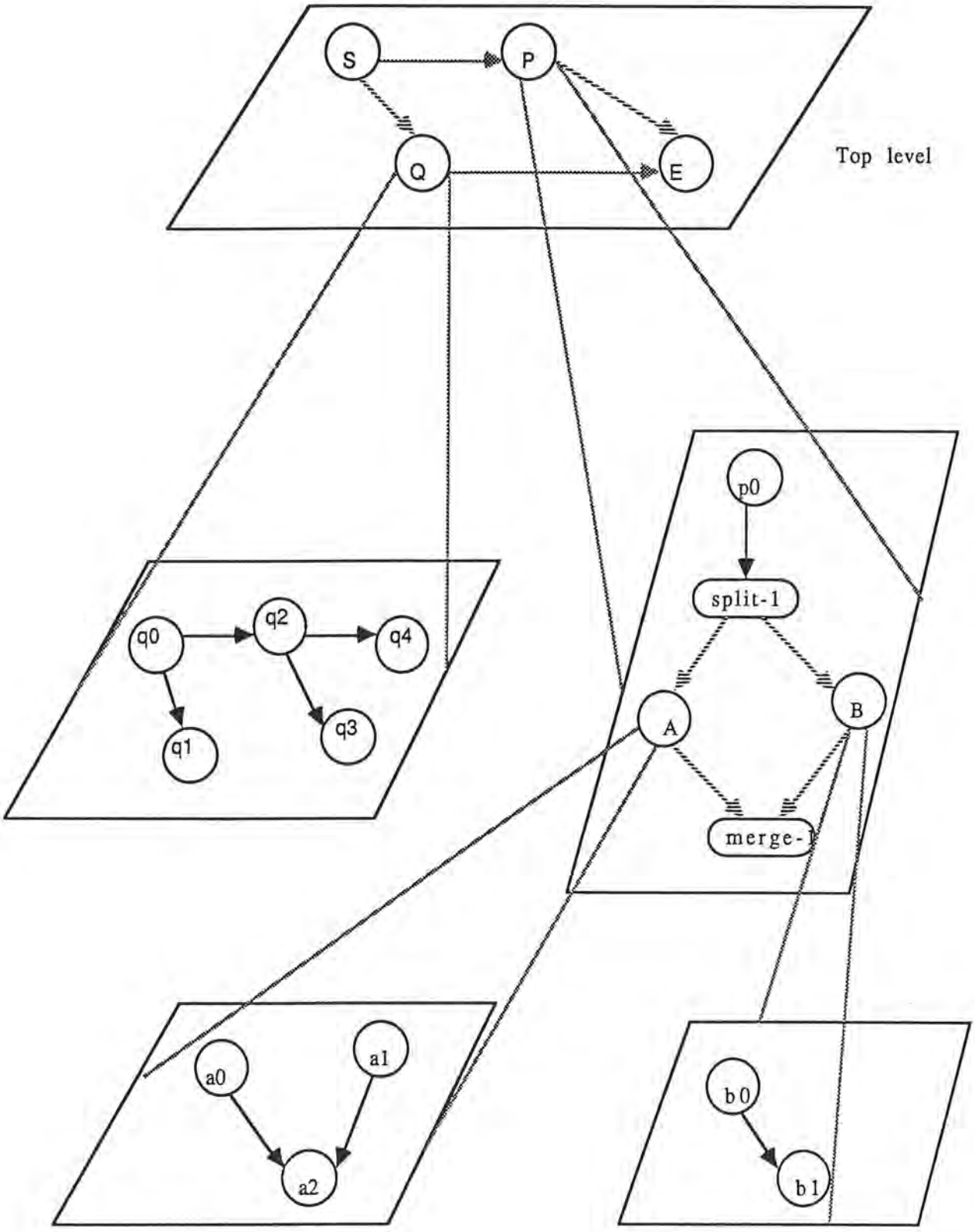



FIG-24

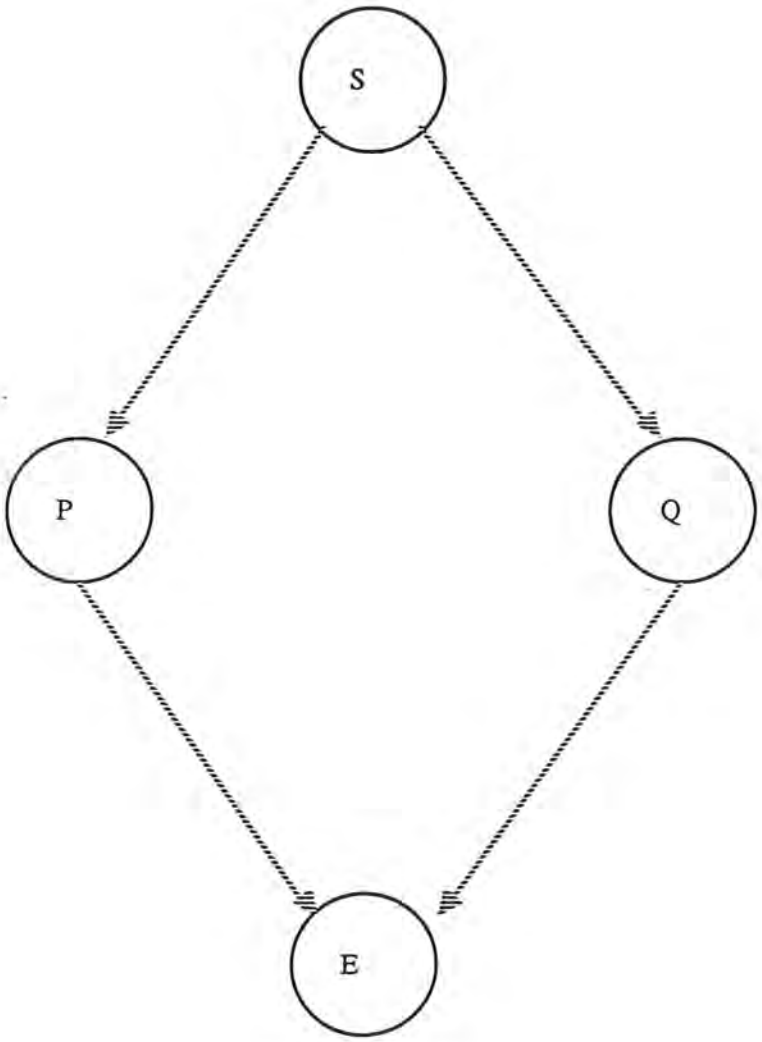



FIG-25-a

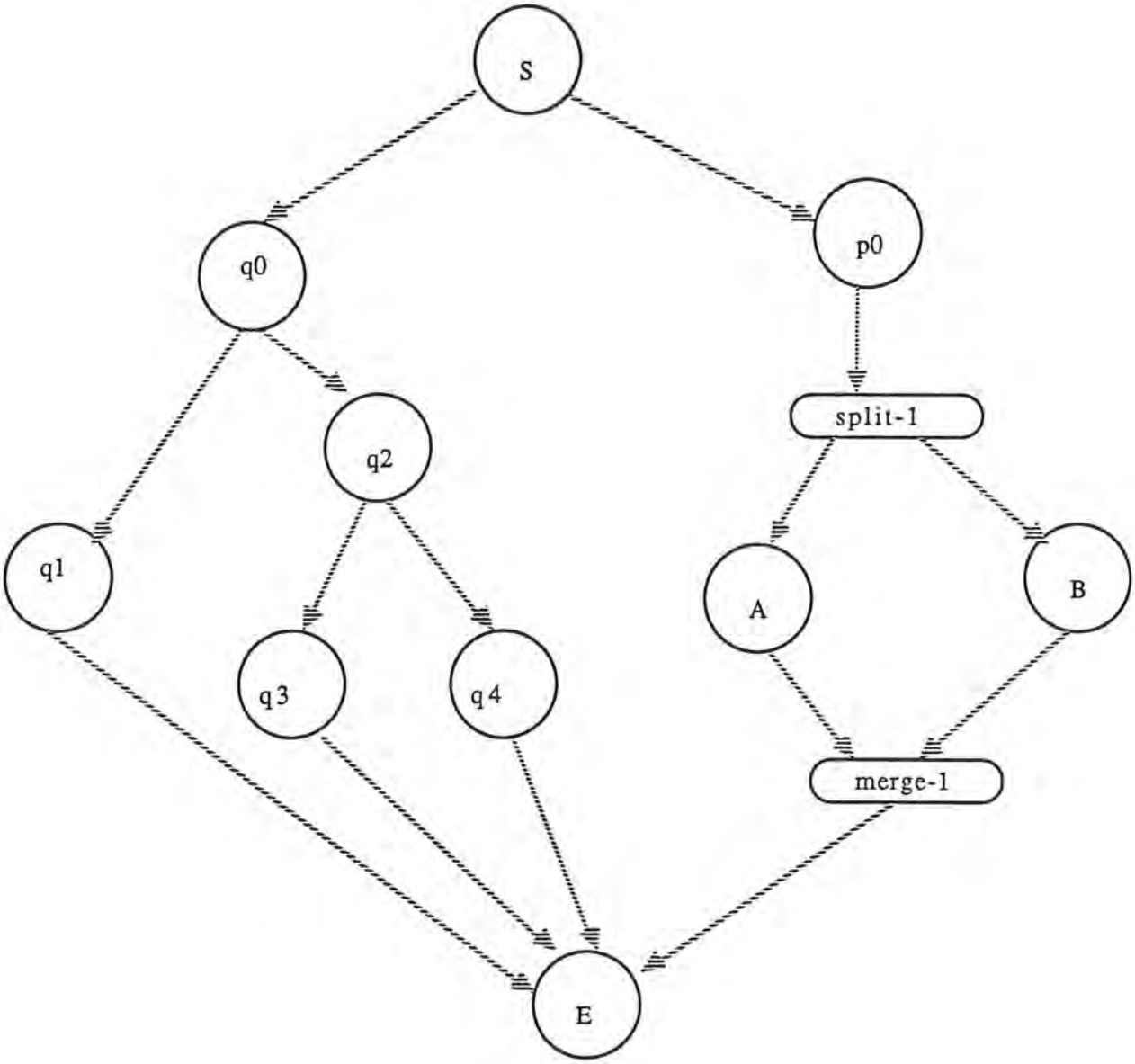



FIG-25-b

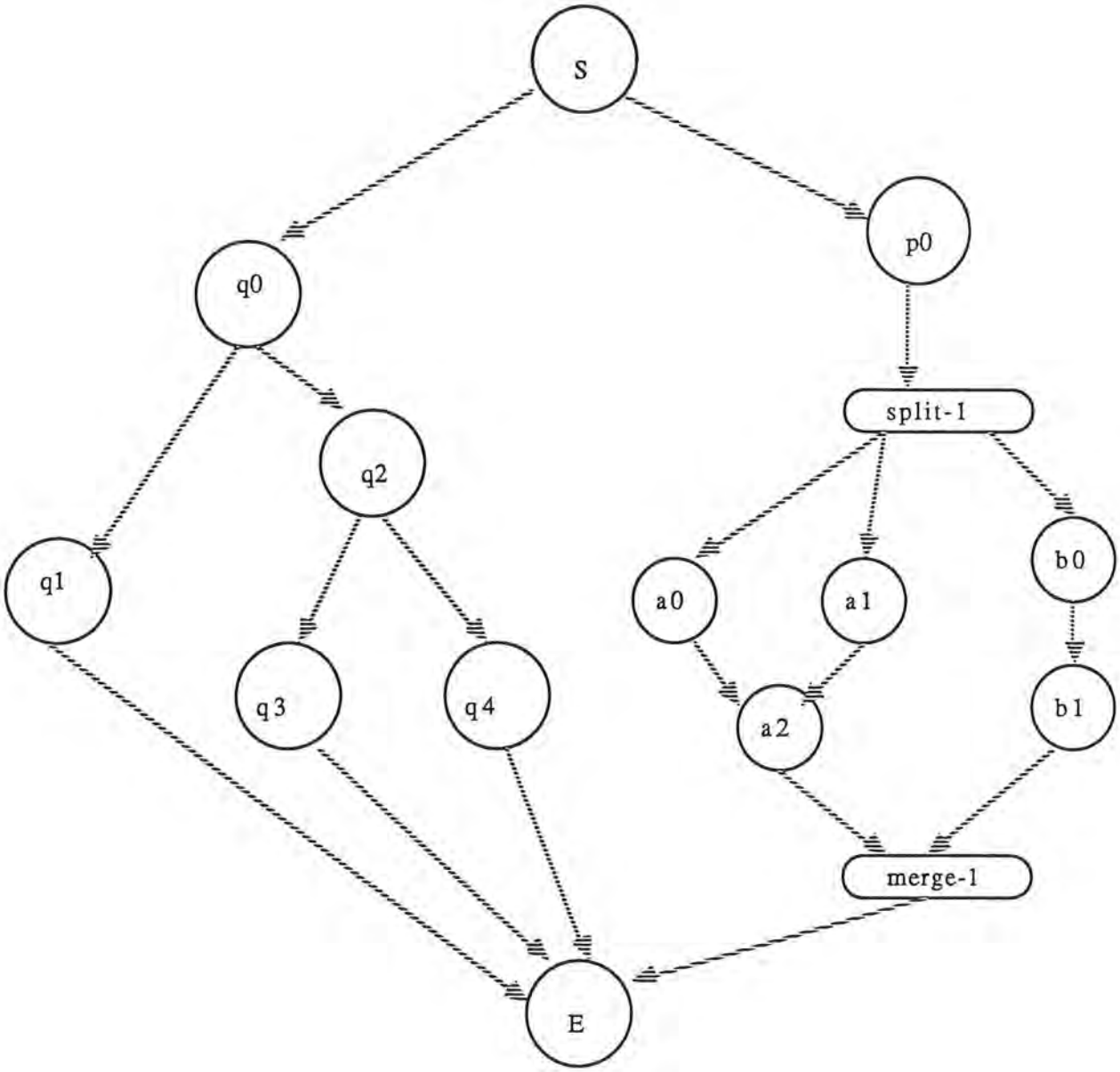



FIG-25-c

**EXAMPLE-1**

**The solution of lower triangular matrix AX = B**

A :- N * N lower triangular matrix.

X :- N vector

B :- N vector.

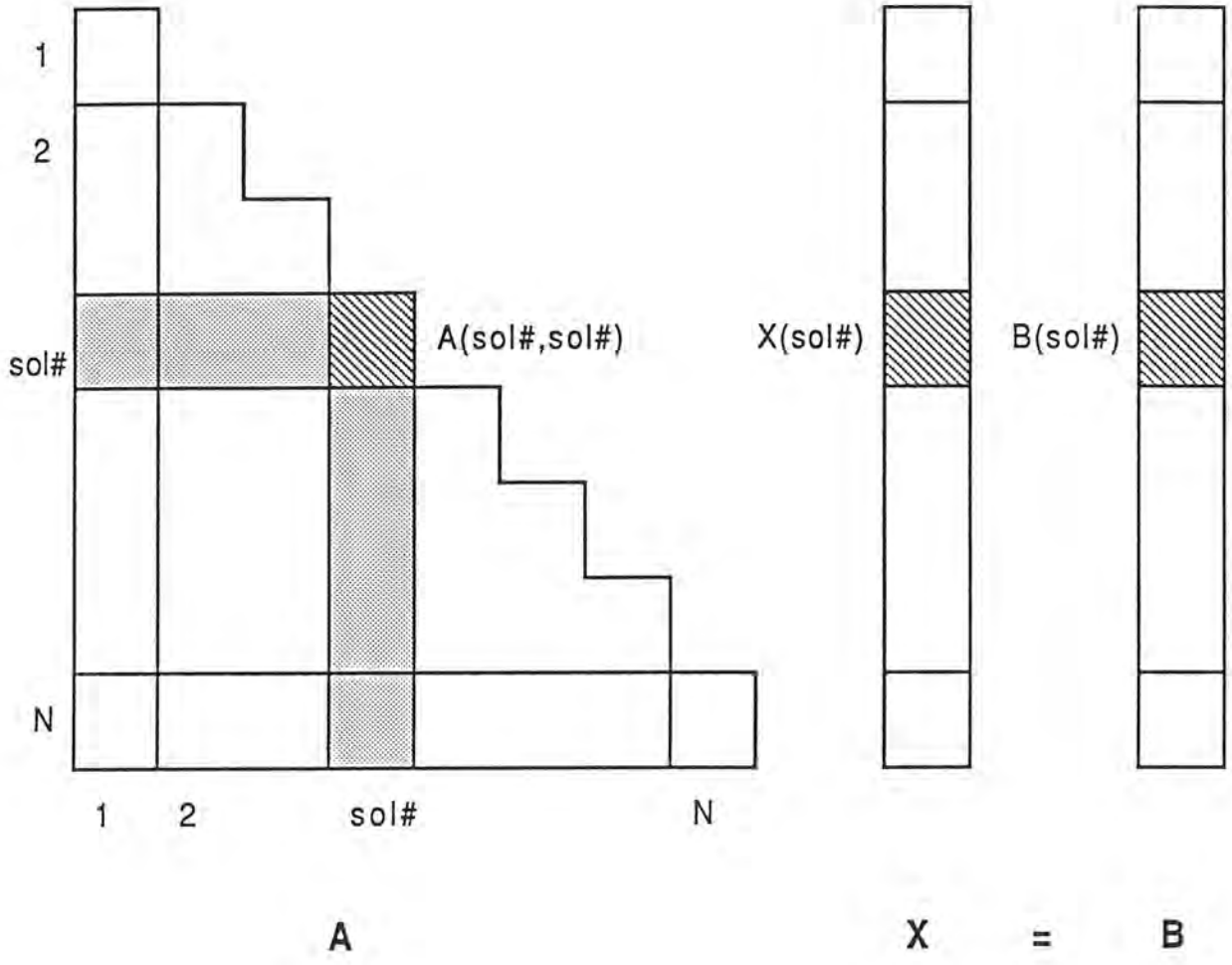



FIG-26

There are two types of tasks in the solution:

Task S(sol#)

Computes X(sol#) = B(sol#)/A(sol#,sol#) for a given sol#

Task T(i,j)

Computes B(i) = B(i) - A(i,j)*X(j) for a given i, j

FIG-27 shows the top level design which consists of a compound node (AX=B) and a storage construct of the data structure to be used in the program. FIG-28 shows the decomposition of the compound node (AX=B) into the compound node solve(1), a replicator over a compound node solve(sol#) for sol# = 2, N, and two strorage constructs.

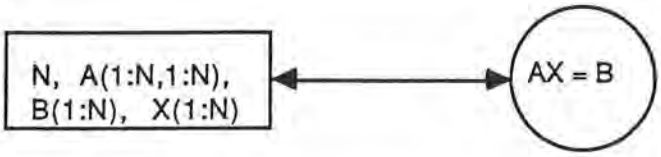
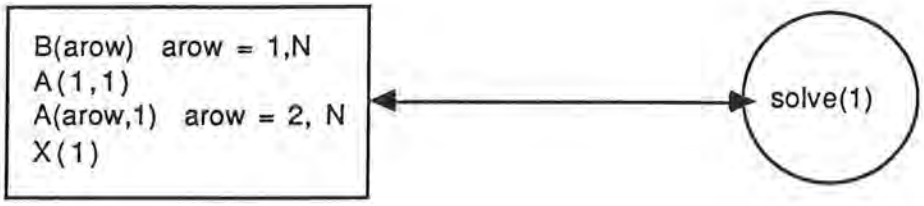
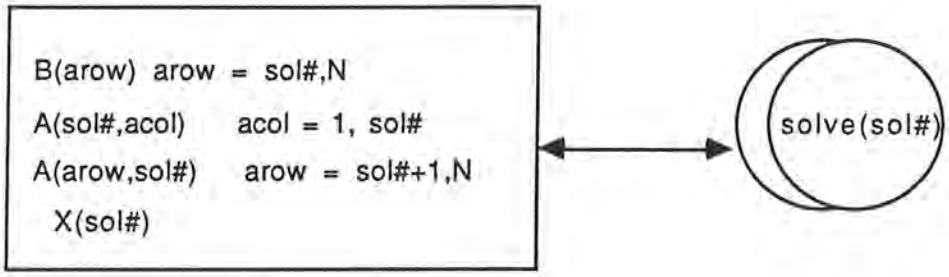



FIG-27 (The top level design)




sol# = 2, N

FIG-28 (AX=B decomposition)

FIG-29 shows the decomposition of the compound node solve(1) and FIG-30 shows the decomposition of the compound node solve(sol#). FIG-31 shows the FORTRAN code associated with simple nodes s(i) for a given i and T(i,j) for a given i and j. The program designer now has finished the program description and the system has to do the rest of the work such as generating the ELGDF program network for a given N, generating the precedence graph for scheduling analysis, and producing synchronization code for certain arcitecture. FIG-32 shows the expanded ELGDF network when N = 3. A precedence graph in compcat form is shown in FIG-33 and an expanded precedence graph when N = 3 is shown in FIG-34. The final FORTRAN code of the program might look like the program given in FIG-35.

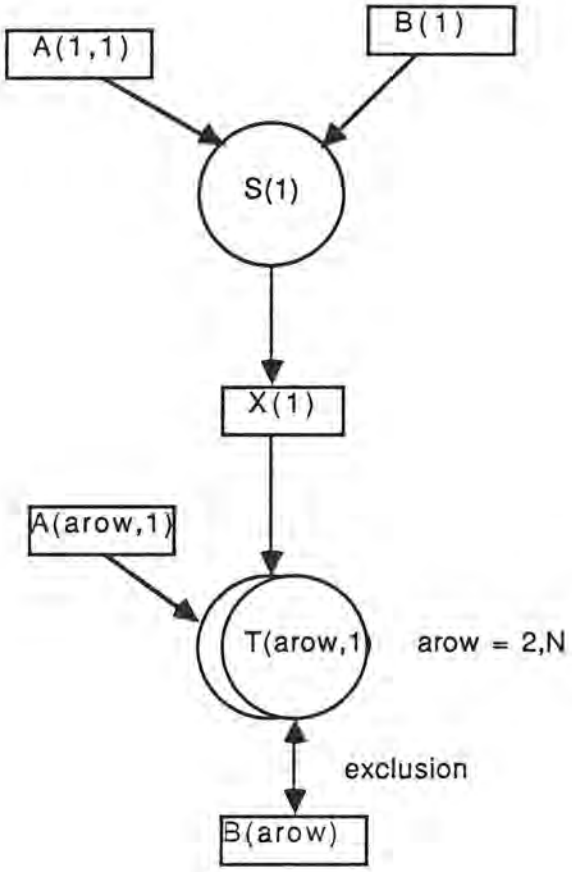



FIG-29

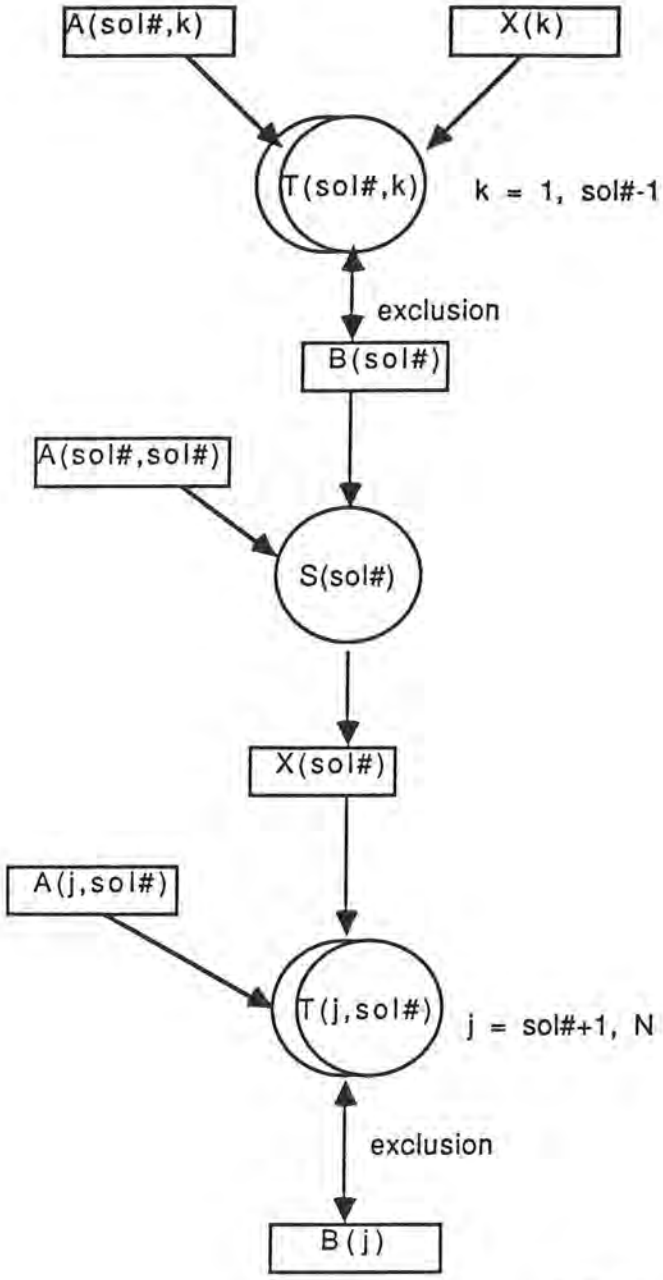



FIG-30

**The FORTRAN code at simple process s(i):**

```
X(i) = B(i)/A(i,i)
```

**The FORTRAN code at simple process T(i,j):**

```
B(i) = B(i) - A(i,j)*X(j)
```

FIG-31

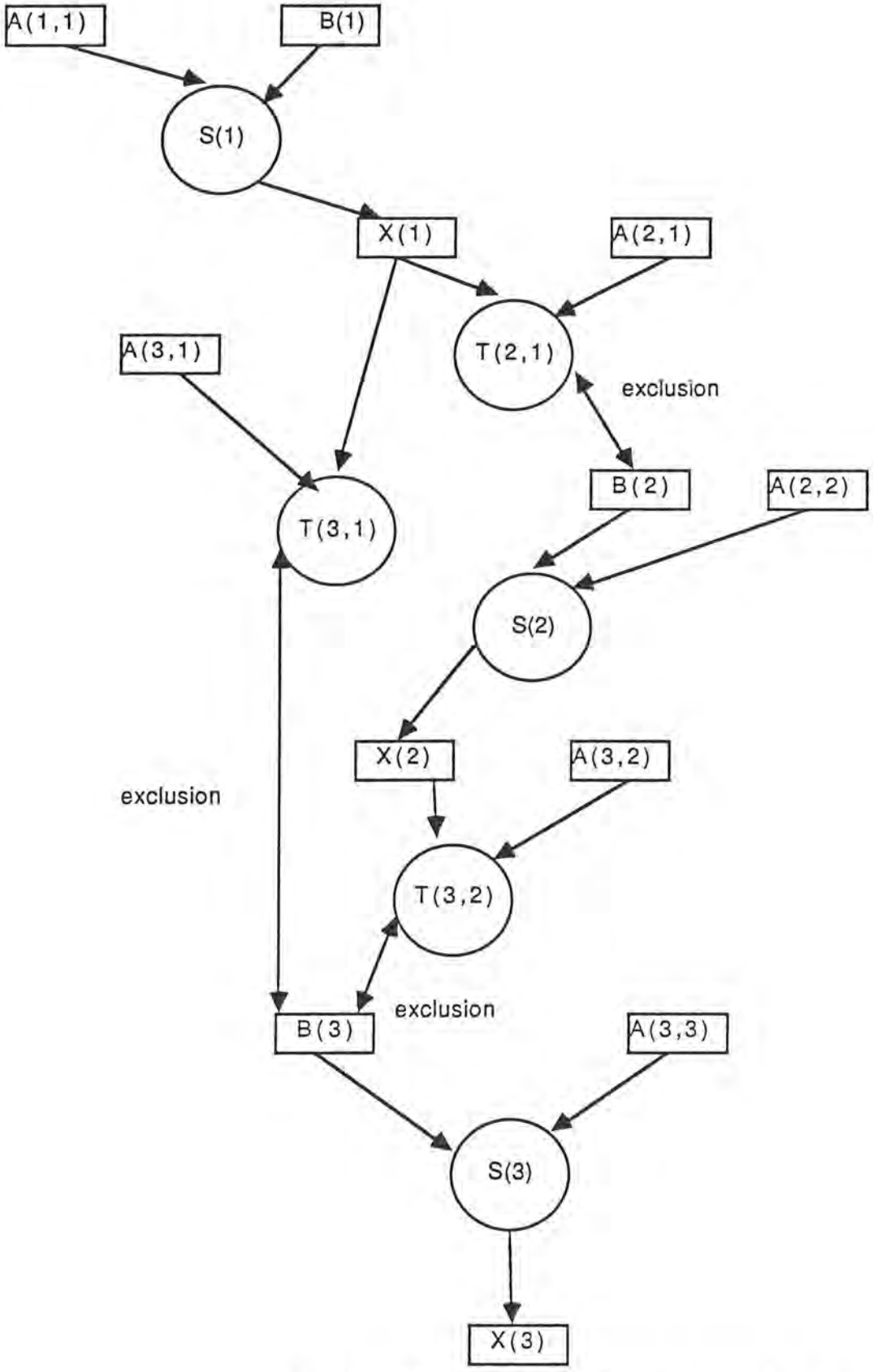



FIG-32 (the expanded ELGDF program when N =3)

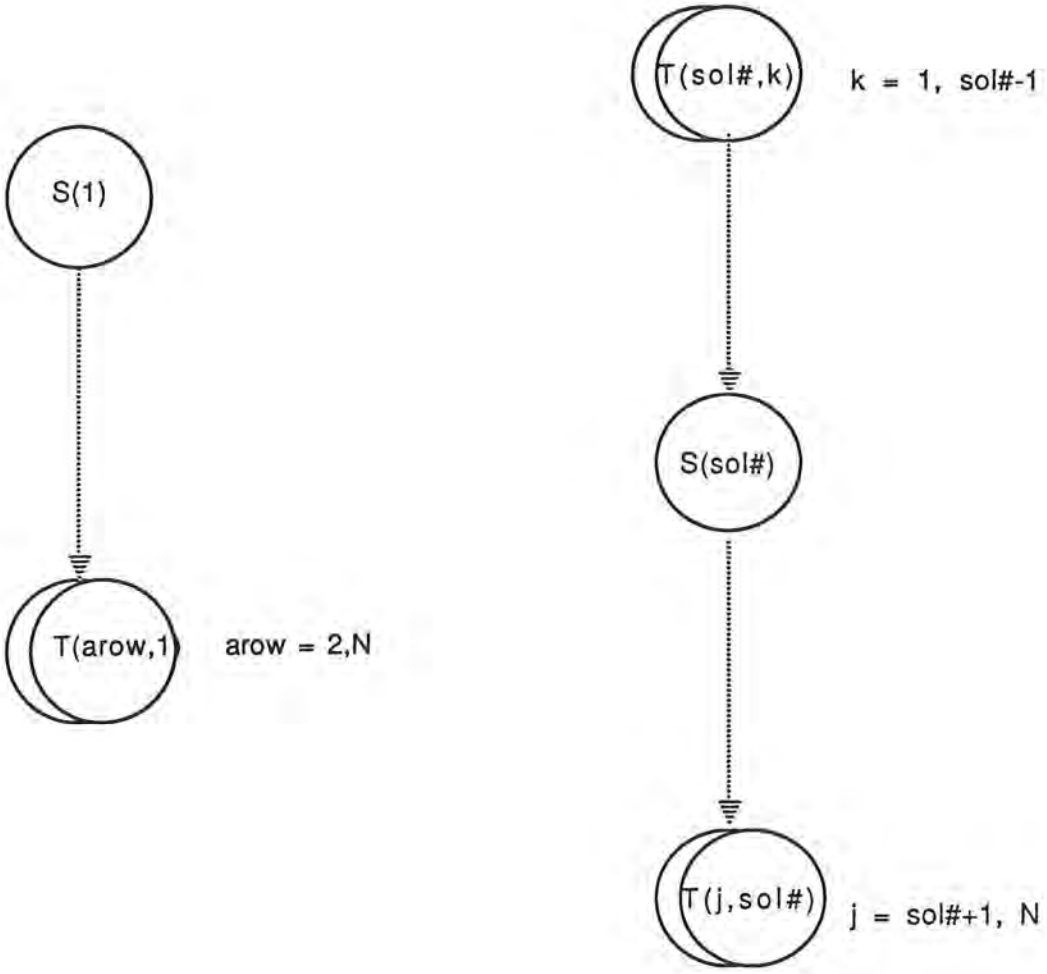



FIG-33 (compact precedence graph)

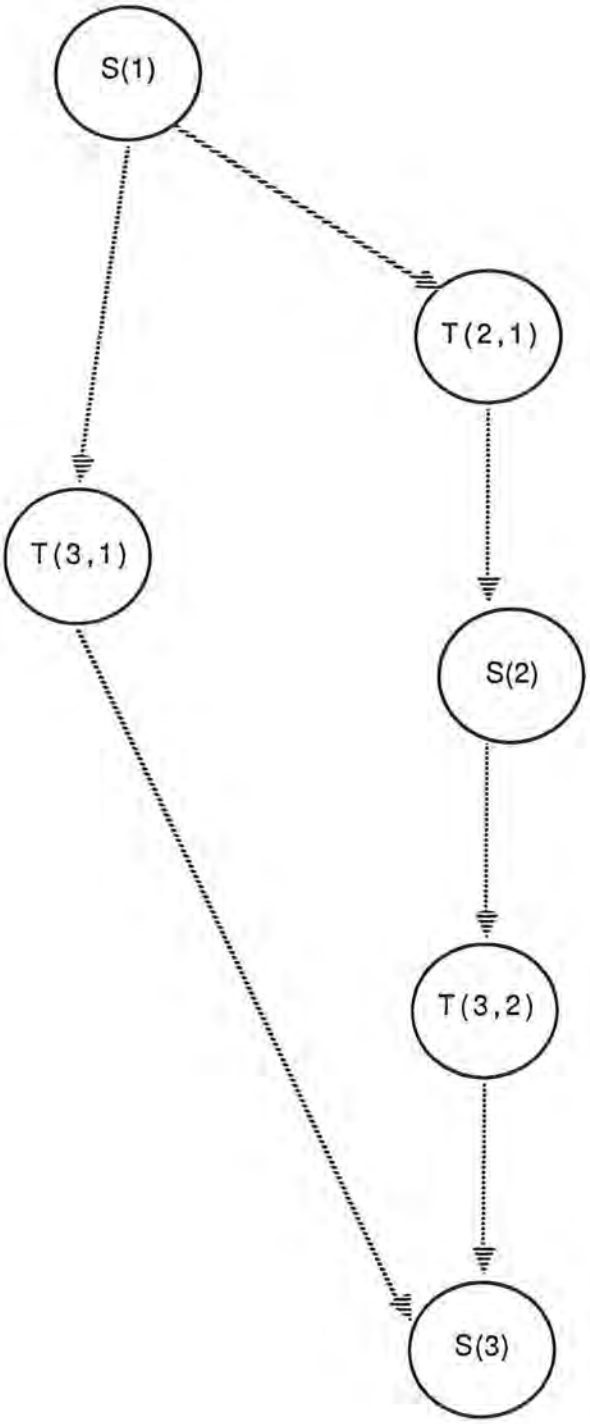



FIG-34 (expanded precedence graph when N=3)

```
C       MAIN
        COMMON A(N,N), B(N), X(N), COUNT(N), N
C       Do loop for parallel execution
        PARDO 77 I = 1, N
 77     SOLVE(I)
        END
        SUBROUTINE SOLVE(I)
        COMMON A(N,N), B(N), X(N), N, COUNT(N)
        INTEGER I, K
C       Synchronization function: the process waits till the condition is true
        WAIT-FOR (COUNT(I) = I-1)
        CALL S(I)
C       Do loop for parallel execution
        PARDO 88 K = I+1, N
        CALL T(K,I)
        M-LOCK(COUNT(I))
        COUNT(I) = COUNT(I) +1
        M-UNLOCK(COUNT(I))
 88     CONTINUE
 99     CONTINUE
        RETURN
        END
        SUBROUTINE S(I)
        COMMON A(N,N), B(N), X(N), N
        INTEGER I
        X(I) = B(I)/A(I,I)
        RETURN
        END
        SUBROUTINE T(I,J)
        COMMON A(N,N), B(N), X(N), N
        INTEGER I, J
        B(I) = B(I) - A(I,J)*X(J)
        RETURN
        END
```

FIG-35

## EXAMPLE-2

### A pipelined sort program

The basic principle of the program will be that a stream of unsorted numbers is passed into a pipeline which has as many parallel processes as there are numbers. Each replicated process has local variables called highest and next. As a number enters a new process it will be compared with the value in highest. If it is not larger than highest, then it will be passed straight on to the next process in the pipe. Otherwise it will be put in highest and the previous value of highest will be sent on to the next process.

FIG-36 shows the top level design which consists of two storage cells (X & Y) and a pipe named pipe1. two repeated arcs (A1 & A2) are connecting X to the top of the pipe and its bottom to Y. FIG-37 shows the attributes associated with the pipe and the two repeated arcs. The expanded ELGDF program network when N = m = 4 is shown in FIG-38. The program designer can use the predefined routines GET and PUT in the FORTRAN code at stage #i. GET(i, VALUE) receives VALUE from the previous stage (i-1) in the pipe or from outside if i = 1. PUT(i, VALUE) sends VALUE to the next stage (i+1) in the pipe or to outside if i = N. The progrm designer does not have to worry about writing the synchronization code for sending and receiving to and from the channels in the pipe or even taking care of the special cases at the two ends of the pipe (stages 1, N). The FORTRAN code associated with stage #i is given in FIG-39 and the FORTRAN code generated by the system is shown in FIG-40 (a & b).

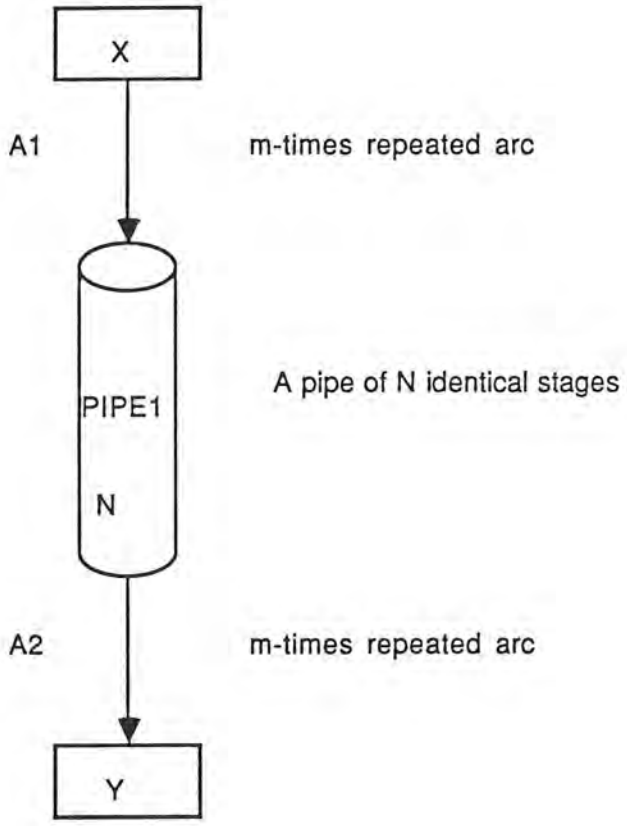

X
A1
m-times repeated arc
PIPE1
N
A pipe of N identical stages
A2
m-times repeated arc
Y

FIG-36

PIPE ATTRIBUTES

NAME : PIPE1
NUMBER OF STAGES : N
NUMBER OF ITERATIONS : m
DATA TYPE : INTEGER
CHANNELS PROTOCOL (SH/CC)* : SH
STAGE # i : COMP(i)

* SH/CC : shared memory / communication channels

REPEATED ARCS ATTRIBUTES

NAME : A1
NUMBER OF ITERATIONS : N
DATA TYPE : INTEGER
SOURCE : X
DESTINATION : PIPE1

NAME : A2
NUMBER OF ITERATIONS : N
DATA TYPE : INTEGER
SOURCE : PIPE1
DESTINATION : Y

FIG-37

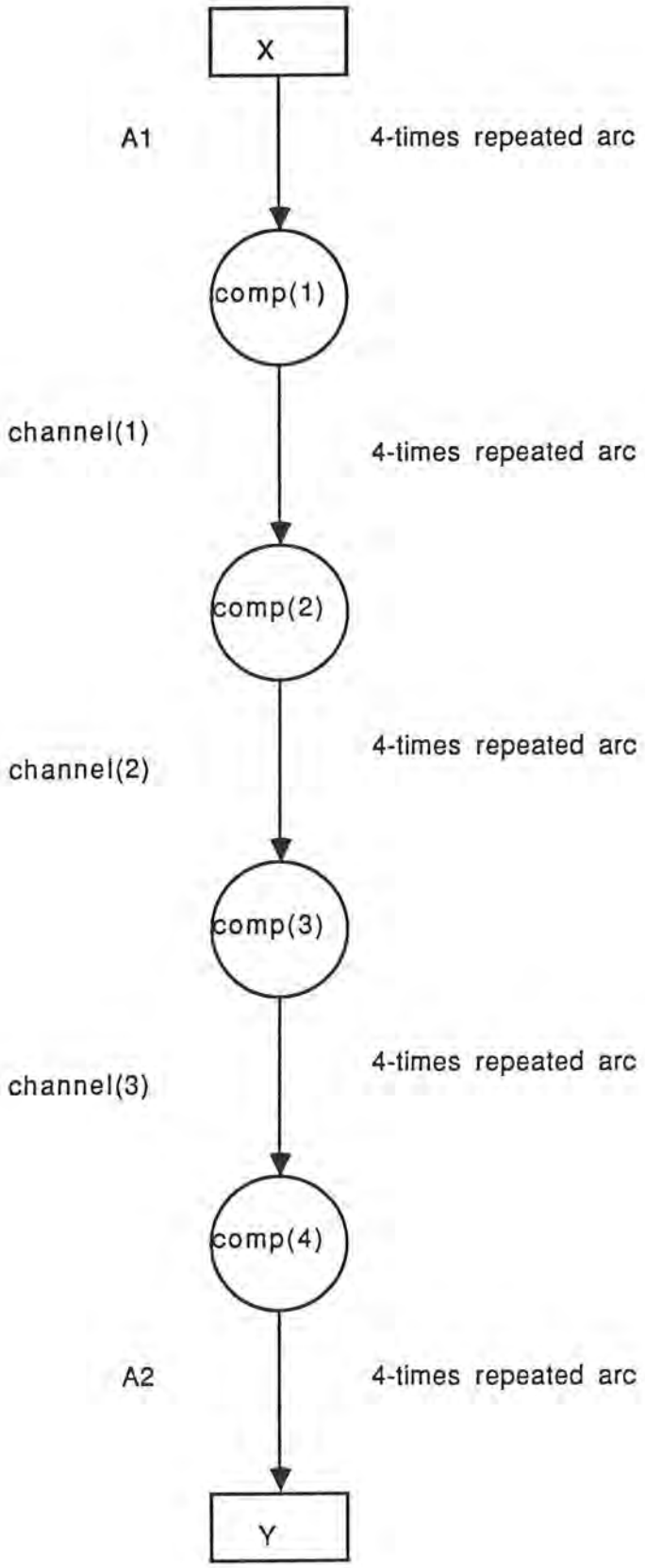



FIG-38 (expanded program when N = m = 4)

**The FORTRAN code associated with comp(i)**

```
      SUBROUTINE COMP(I)
      COMMON m
      INTEGER I, HIGHEST, NEXT,J
      CALL GET (I,HIGHEST)
      DO 99 J = 2, m
      CALL GET(I,NEXT)
      IF (NEXT. LE. HIGHEST) THEN
        CALL PUT(I,NEXT)
      ELSE
        CALL PUT(I,HIGHEST)
        HIGHEST = NEXT
      ENDIF
99    CONTINUE
      CALL PUT(I, HIGHEST)
      RETURN
      END
```

FIG-39

```
C       MAIN
C       R(N-1) , W(N-1) are two arrays of semaphores
        COMMON  N,  m,  CHANNEL(N-1),  R(N-1),W(N-1)
        INTEGER I
C       initialization
        DO 33 I = 1, N-1
C       P & V (semaphore operations)
 33     P(R(I))
C       Do loop for parallel execution
        PARDO 88 I = 1 , N
        CALL COMP(I)
 88     CONTINUE
        END

      SUBROUTINE GET(INDEX, VALUE)
       INTEGER INDEX, VALUE
       IF (INDEX .EQ. 1) THEN
         VALUE = X
       ELSE
        VALUE =  GET-FROM-CH(INDEX-1)
       RETURN
       END

       SUBROUTINE PUT(INDEX,VALUE)
       INTEGER INDEX, VALUE
       IF (INDEX .EQ. N) THEN
         Y = VALUE
       ELSE
       CALL PUT-INTO-CH(INDEX, VALUE)
       RETURN
       END
```

FIG-40-a

```
      FUNCTION GET-FROM-CH(INDEX)
      COMMON N, CHANNEL(N-1), R(N-1), W(N-1)
      INTEGER INDEX
C     P & V semaphore operations
      P(R(INDEX))
      GET-FROM-CH = CHANNEL(INDEX)
      V(W(INDEX))
      RETURN
      END

      SUBROUTINE PUT-INTO-CH(INDEX,VALUE)
      COMMON N, CHANNEL(N-1), R(N-1), W(N-1)
      INTEGER INDEX, VALUE
C     P & V semaphore operations
      P(W(INDEX))
      CHANNEL(INDEX) = VALUE
      V(R(INDEX))
      RETURN
      END
```

FIG-40-b

## REFERENCES

1. P. David Stotts, "The PFG Environment: Parallel programming with Petri net semantics," proceedings of the HICSS conference, 1988.

2. R. G. Babb, "parallel processing with large-grain data flow techniques," IEEE Computer, pp. 55-61, July 1984.

3. Lawrence Snyder, "Parallel programming and the poker programming environment," IEEE Computer, pp. 27-36, July 1984.

4. Lawrence Snyder and David Socha, "Poker on the cosmic cube: The first retargetable parallel programming language and environment," proceedings of ICPP, 1986 IEEE.

5. A. K. Adiga and J. C. Browne, "A graph model for parallel computations expressed in the computation structures language," proceedings of ICPP, 1986 IEEE.

6. David C. DiNucci and R. C. Babb, "Practical support for parallel programming," proceedings of the HICSS conference, 1988.

7. T. D. Kimura, "Visual programming by transaction network," proceedings of the HICSS conference, 1988.

8. J. C. Browne, "Formulation and programming of parallel computations: a unified approach," proceedings of ICPP, 1985 IEEE.

9. G. Raeder, "A survey of current graphical programming techniques," IEEE Computer, pp. 11-25, August 1985.

10. E. P. Glinert and S. L. Tanimoto, "Pict: An interactive graphical programming environment," IEEE Computer. pp.7-25, November 1984.

11. R. G. Babb and D. C. DiNucci, "Design and implementation of parallel programs with large-grain data flow," in Characteristics of parallel algorithms, Cambridge, MA: MIT Press,

1987, pp. 335-349.

12. T. Agerwala and Arvind, "Data flow systems: Guest editor's introduction," IEEE Computer, pp. 10-13, February 1982.

13. G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," ACM computing surveys 15, 1(March 1983), 3-43.

14. W. B. Ackerman, "Data flow languages," IEEE Computer, pp. 15-25, February 1982.

15. A. L. Davis and R. M. Keller, "Data flow program graphs," IEEE Computer, pp. 26-41, February 1982.

16. J. W. Choi and T. D. Kimura, "A compiled picture language on Macintosh," ACM, pp. 72-77, 1986.

17. R. B. Grafton and T. Ichikawa, "Visual Programming: Guest editor's introduction," IEEE Computer, pp. 6-9, August 1985.

18. S. Ahuja, N.Carriero, and D. Gelernter, " Linda and Friends," IEEE Computer, pp. 26-34, August 1986.

19. John R. Allen, and Ken Kennedy, " A Parallel Programming Environment," IEEE Software, July 1985.