Data Driven Nets:
A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems

A. L. Davis

UUCS - 78 - 108

July 1978

This work was supported by a grant from Burroughs Corporation.

Data Driven Nets:
A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems

A. L. Davis
Computer Science Department
University of Utah
Salt Lake City, Utah

ABSTRACT - A procedural parallel process representation, known as data-driven nets is described. The sequencing mechanism of the data-driven representation is based on the principle of data dependency. Operations are driven into action by the arrival of the required working set of input operands. Execution of DDN processes is side-effect free, and influence in the net representation is transparent. Data-driven nets have several advantages over many of the existing parallel process representations. These nets are capable of representing parallelism below the statement level, and in addition may be arbitrarily pipelined. Data-driven nets are simpler than other data-flow schema in that no distinction need be made between control and data. A process model for data-driven nets is given and a number of properties of the model are discussed. The operating rules for data-driven nets are completely asynchronous and the nets therefore serve as an excellent low-level process notation for distributed systems.

KEY WORDS - data-driven nets, data-flow schema, asynchronous, distributed control, data dependency.

I. INTRODUCTION

A procedural parallel process representation, known as "data-driven nets" (DDN's) is described. The sequencing mechanism of the data-driven representation is based on the principle of data dependency. Operations are driven into action by the arrival of the required "working set" of operand data. Execution of DDN processes is side-effect free, and the things which influence individual DDN operations are transparent. The intent of the DDN schemata is to provide a low level representation for the study of parallel processes which are to be executed on fully distributed, asynchronous machine systems. Fully distributed systems are defined here to have two principle physical characteristics: 1) at no time can a module of a fully distributed system determine the total system state, and 2) a fully distributed system is incapable of enforcing simultaneity in its distributed modules.

It is not intended that anyone should program directly in the low-level DDN representation, but rather that the actual programming language be translated into DDN form for execution. It is possible to translate well-structured programs in conventional languages into DDN's, but these languages are not well suited to the specification of parallel algorithms. A better approach would be to program in a language such as ID[2], which is both well suited to the description of parallel programs and easily translated into DDN's.

DDN's have several advantages over existing parallel process languages and representations. DDN's are capable of representing concurrency below the statement level, and therefore inherently represent much higher levels of concurrency than statement-oriented languages such as parallel PASCAL[6]. In addition, DDN's may be arbitrarily pipelined (physical resources permitting), while statement languages do not admit so readily to pipelined execution. DDN's are more concrete than the conceptually nice models of Seror[10], and Adams[1]. Another nice model for parallel processing is Kotov's trigger function

---

Note - The DDN representation described here has been implemented as the machine language on a special purpose data-driven machine, DDM1. DDM1[3] was built and programmed by the author and two colleagues from Burroughs Corporation, Karl Boekelheide and L. D. Rogers. DDM1 was completed in July of 1976 and now resides at the University of Utah, where the project continues under Burroughs support.

approach[7]. Trigger functions however require centralized storage and control for execution and are therefore not well suited for distributed control systems. DDN's are similar to the data-flow nets of Dennis [4], and Rodriguez[9], which share the above stated advantages over the non data-flow representations. The advantage of DDN's over the data-flow models of Dennis and Rodriguez is that no distinction need be made in DDN's between information tokens used for control purposes and other types of information. The lack of this distinction yields increased simplicity in DDN processes with no loss of representational power. In addition the DDN primitives, while not being any more numerous than those of the other data-flow languages, are more general.

A process model will be defined for DDN processes and certain properties of these processes will be described. A process call mechanism will then be given which allows hierarchically and recursively defined DDN's to be specified. This permits a clean substitution rule, and allows DDN's to be created and executed in a well-structured, hierarchical manner. Dealing with data structures has traditionally been a major downfall of data-flow schemata. A method will be suggested for dealing with data structures, which marks a distinct departure in thinking from other data-flow groups. Finally, DDN's will be used to represent a variety of situations in order to illustrate the model.

II. The Data Driven Schema

A data-driven net is a bipartite graph, which consists of cells interconnected by directed data paths. A cell may have any number of input and output paths. Information is passed along the data paths in quantum units called items. An item may be a character, number, vector, matrix, literal, etc. A data item is similar to a variable name in conventional program representations. An important distinction is that a named item does not correspond to a storage location in DDN's, but rather to a value which plays a particular role in some computation. The data paths are queues of arbitrary length. The length of these queues may be specified a priori or constrained by an implementation. To avoid a detailed description of the queuing phenomena, it will be assumed here that each data path is a queue of finite but arbitrary length. Implicit in the mechanism for transferring data items between cells on FIFO data paths is an asynchronous request-acknowledge control protocol.

When each member of a set of input data paths (called the firing set) contains at least one item, the cell is said to be fireable. A cell fires at some finite (but unspecified) time after it becomes fireable. When a cell fires, the firing set data items are destroyed, and a set of resultant data items are placed on the output paths. The order in which the output data items appear on the output paths is unknown. The time at which the outputs appear after a cell fires is finite but unspecified, and no assumption can be made about the order or the relation between the times at which the output items appear. This completely asynchronous cell behavior is essential to a schema which is to be easily implemented in a distributed control environment. A cell is said to have fired only after all of the firing set data items have been removed and all output data items have been placed on the output paths.

An example of a cell firing is shown in Figure 1, where a cell performs a simple integer addition. In this case, the firing set is the set of all input data paths.

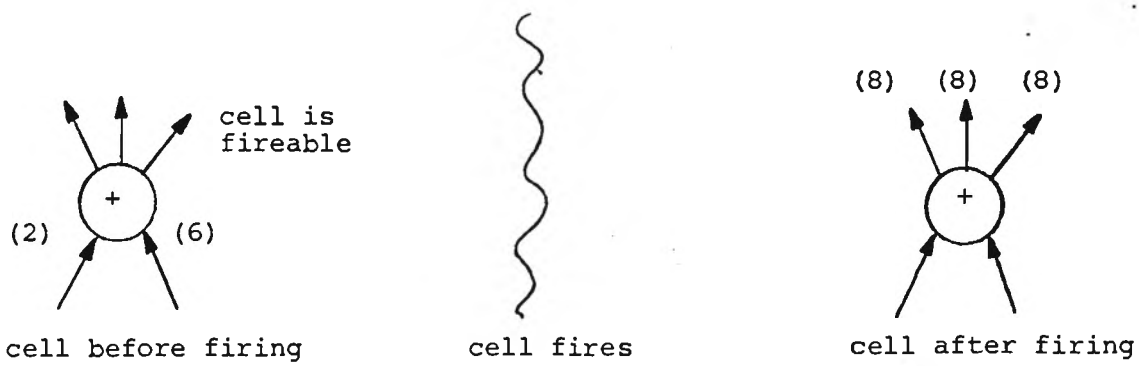


Figure 1: A Sample Cell Firing

A cell generates outputs according to its cell function. The cell function in Figure 1 would be: all output paths receive the sum of the input path items. Figure 2 shows a case where the data paths are treated as queues, and thereby support pipelined execution. In the DDN schemata, cell functions are defined by the cell type (except for the OPERATOR cell, whose cell function is further modified by the specification of a particular operator).

Seven distinct cell types are used in data-driven nets. The choice of cell types is analogous to the choice of op-codes or statement types in sequental programming languages. It is possible to specify primitives at a higher or lower

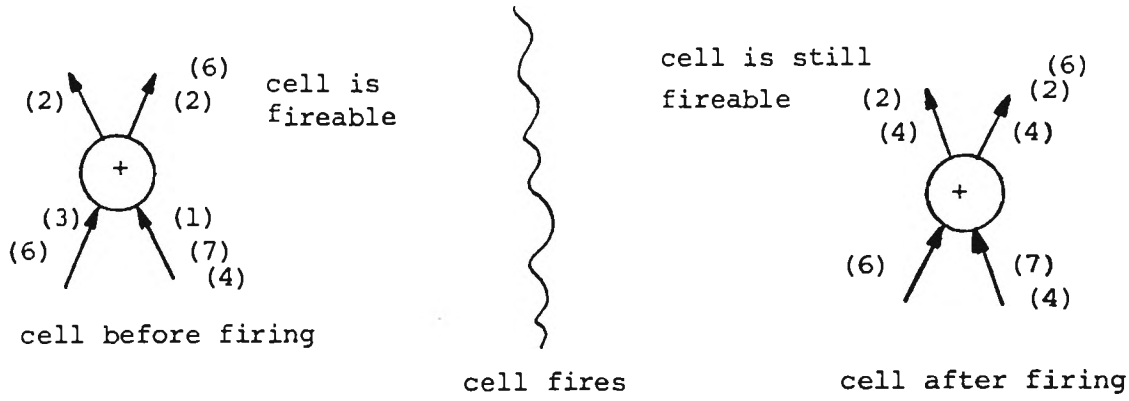
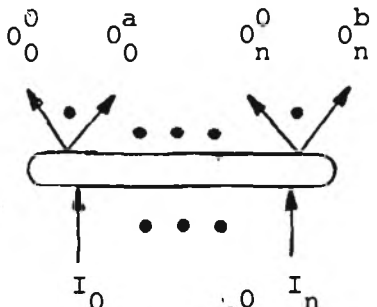


Figure 2: Pipelined execution

level than that of almost any set of primitives in question. There is also the option of selecting between a minimal set and a larger more powerful set. The DDN cells were chosen for simplicity and generality, and each cell type was chosen to clearly characterize a particular type of activity that exists in parallel programs. Each cell type is represented by a unique graphical symbol. Figure 3 shows the cell types, their firing sets, and their cell functions. Each type of data path is named. Subscripts indicate data paths which may receive different valued tokens, while superscripts indicate data paths which will carry identical valued copies of output data items. Since each data item of a firing set is destroyed when the cell fires, any time a data item is to be used in more than one place (due to either pipelining or concurrency requirements), more than one copy of that output item will need to be explicitly produced. This implies that the output destination for any output may be a destination list. If there are n elements of a given destination set, then n copies of the output item will be made and sent to the n respective destinations. Note that input paths will never have superscripted names, but outputs always do, indicating that any output may be multiply copied.

Unlike the other cells, the GATE cell operates on the basis of an internal state. The cell function and firing set depend on this internal state. Due to the asynchronous nature of DDN's and the arbitrary length of the queue data paths, the normal Moore or Mealy state descriptions are insufficient and the normal asynchronous flow tables are unnecessarily complex and mask the actual cell behavior. Therefore the state machine description for the GATE cell shown



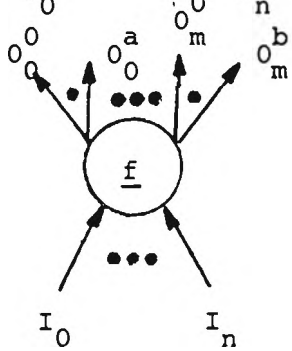
SYNCH CELL

I_j: inputs

O_j^i: outputs

firing set: $(I_0, \ldots, I_n)$

cell function: for every i, j: $O_j^i := I_j$



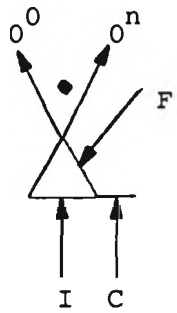
OPERATOR CELL

I_j: inputs

O_j^i: outputs

firing set: $(I_0, \ldots, I_n)$

cell function: for every i, j: $O_j^i := \underline{f}(I_o, \ldots, I_n)$



GATE CELL

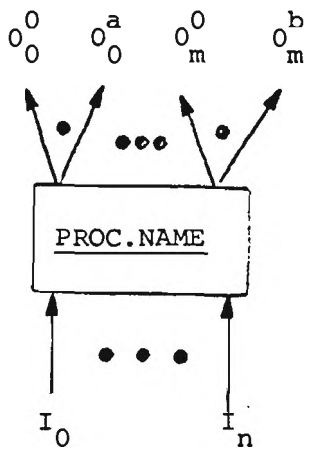
I: initial input

F: feedback input

C: condition input

O^i: outputs

for the cell function and firing set see Fig. 4.



CALL CELL

I_i: inputs

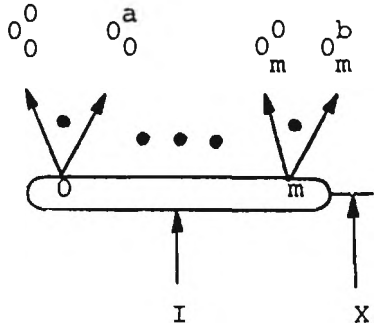
O_j^k: outputs

firing set: $(I_0, \ldots, I_n)$

cell function: for every a, b

$$O_b^a := \underline{\text{PROC.NAME}}\ (I_o, \ldots, I_n)$$

Figure 3: DDN Cell Types



DISTRIBUTE CELL

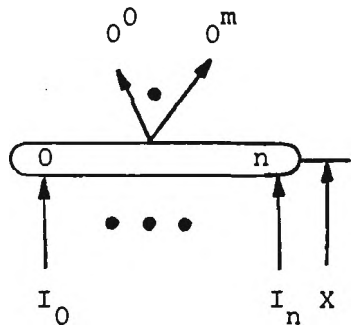
I: input

O_j^i: outputs

X: index

firing set: (I, X)

cell function: $O_x^i := I$ for all i and where x is the value of X



SELECT CELL

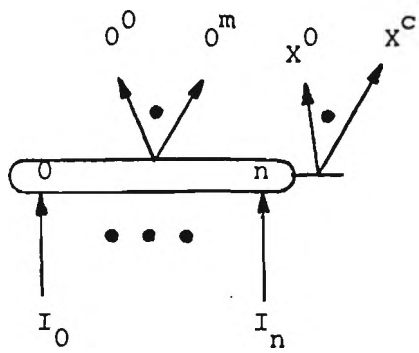
I_j: inputs

O^i: outputs

X: index

firing set: (I_x, X) where x is the value of X

cell function: $O^i := I_x$ for all i



ARBITER CELL

I_j: inputs

O^i: outputs

X^k: index outputs

firing set: at least one input: I_j

cell function: $O^i :=$ first I_j; $X^a := j$ for all i, a

(Note: in case of a tie any input I_j which is present is chosen).

Figure 3 (Cont'd): DDN Cell Types

in Figure 4 should be interpreted as a normal Moore machine except:

1. When a data item arrives at the GATE cell that is not of the type labeling an exit from the current state, that data item is queued in the normal manner and no change in the current state occurs.
2. When in a given state, and any data item of the type labeling an exit from that state exists (at the head of the appropriate queue) then that exit can be taken and the corresponding state change can be made.
3. When more than one next state is possible, any one may be taken.

While it is possible to define the GATE cell to operate in a more concurrent manner, the increased complexity of the resulting state table would be considerable. This model of the GATE cell assumes that data items will be taken in the following order: (I, (Ct, F)* , Cf)* where ('sequence')* denotes zero or more instances of the 'sequence'.

The basic function of the GATE cell is to perform a controlled merge operation on inputs I and F, as specified by the input C. Initially the gate is set "open" to pass a single I data item, then the gate "closes" to inhibit further I's and allow F's to pass. After a Cf input arrives the gate again opens. The GATE cell is used to control iterative situations, and will be clarified in the examples.

The OPERATOR, SYNCH, DISTRIBUTE and CALL cells exhibit conjunctive firing rules, i.e. all input paths must contain at least one data item for the cell to be fireable. The GATE, SELECT, and ARBITER cells have disjunctive firing rules, in that only a certain subset of input data paths are required to contain at least one data item before the cell becomes fireable. Which subset is determined by:

a) Arrival order of the ARBITER cell inputs.
b) Value of the internal state for the GATE cell.
c) Value of data item C for the SELECT cell.

The situation where several data paths terminate at a single destination is not allowed. This would imply that non-deterministic merging could occur at such a junction. The pragmatic approach is taken here, and non-determinacy is not viewed as something to be sought after, but rather something that should be explicitly avoided. Merging of data paths is allowed in well-controlled instances as provided by the GATE, SELECTION, and ARBITER cells.

inputs/next state

State ID	I	F	C_t	C_f	Function
1. initial	2	7	6	6	wait
2. I sent	-	-	3	4	send & destroy I
3. iterate	-	5	-	-	destroy C
4. reinit.	2	7	6	6	destroy C
5. F sent	-	-	3	4	send & destroy F
6. ERROR	8	7	6	6	destroy C
7. ERROR	8	7	6	6	destroy F and send NULL
8. ERROR	8	7	6	6	destroy I and send NULL

where: C_t is a true condition input

C_f is a false condition input

F is the feedback input

I is the initial input

NULL is a special data item (see section VI)

Figure 4: GATE Cell State Table

III. Some Basic DDN Examples

Figure 5 illustrates the two types of concurrency that can be obtained using the DDN representation (pipelining and independent operations). Under pipelining all five cells of the net may be concurrently active, but with no pipelining at most two cells may fire in parallel. All of the data items in this example are simple integers. The execution sequence is not unique, but barring any new inputs the final configuration will be the same for any execution sequence. This output functional property of DDN's is aided by the persistence property and the FIFO path discipline. Persistence implies that once a data item has been placed in a destination queue, it can only be removed by the appropriate firing of the respective destination cell. Ordering of data items within a given data path is preserved by the queues. Persistence and data item ordering are necessary but not sufficient conditions for output functionality. If some cell function specified that the cell was to pick any input and place it on any output, then a DDN containing such a cell would not be output functional. All DDN cell functions are output functional, and therefore it is possible to determine by topological inspection whether or not a given DDN is output functional.

Several common conditional situations are depicted in Figure 6. Note that conditional control (IF or CASE expressions in traditional sequential control languages) corresponds to conditional routing of the data items in DDN's.

Iteration in DDN's corresponds to a directed circuit in the net. In general an iterative net is represented as: 1) a net or process to be iterated, 2) a set of initial data paths, 3) a set of feedback data paths, and 4) a set of output data paths. This is illustrated in Figure 7.

Proper sequencing for such an iteration would be:

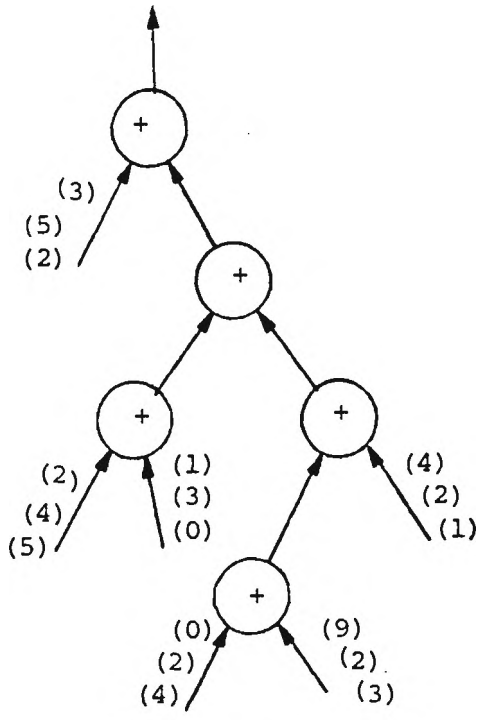
a) When each initial data path has an item, the net fires.

b) When the net has fired, output items are placed on the feedback paths, and the net is then primed to fire again.

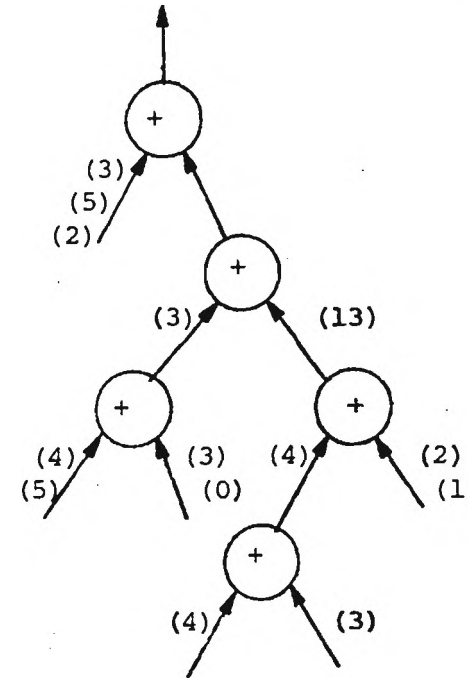
c) Step b is repeated until the iteration started by the first set of initial inputs terminates and the sequence is then restarted.

Iterative DDN's present several problems:

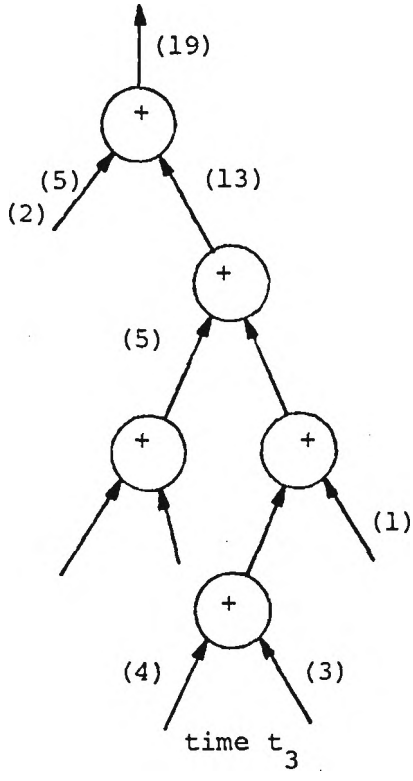
a) How to terminate an iteration.



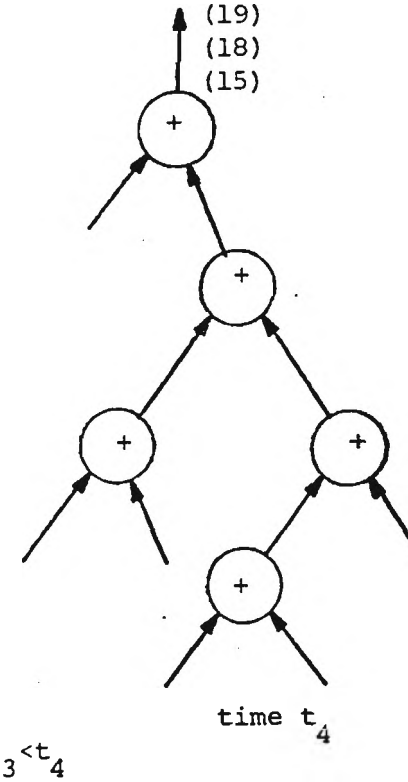
time t_1



time t_2



time t_3



time t_4

where $t_1 < t_2 < t_3 < t_4$

Figure 5: Two types of DDN Concurrency

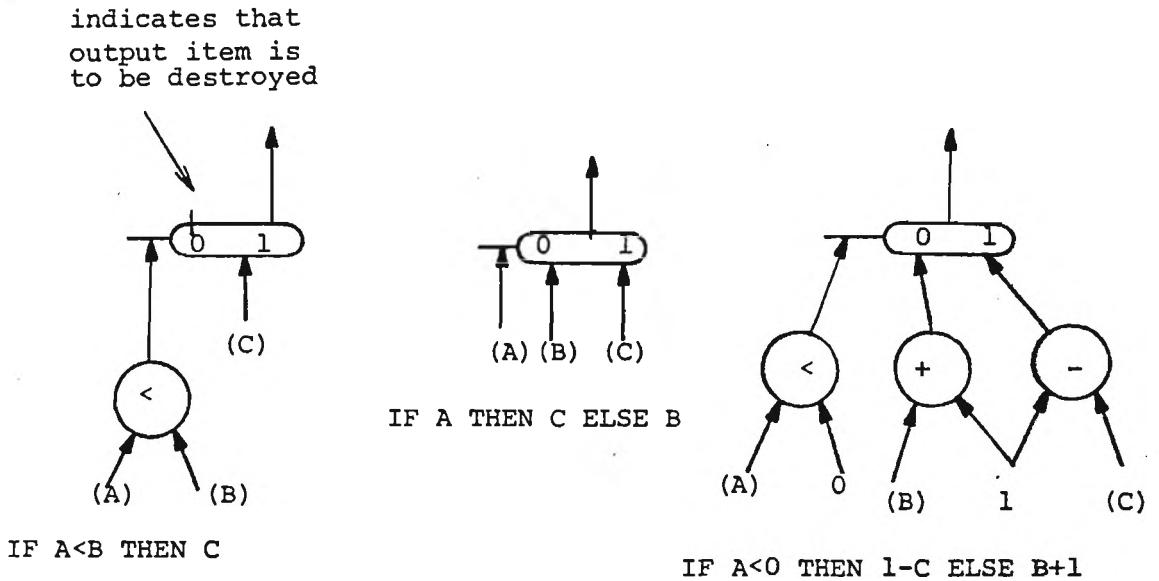


Figure 6: Conditionals

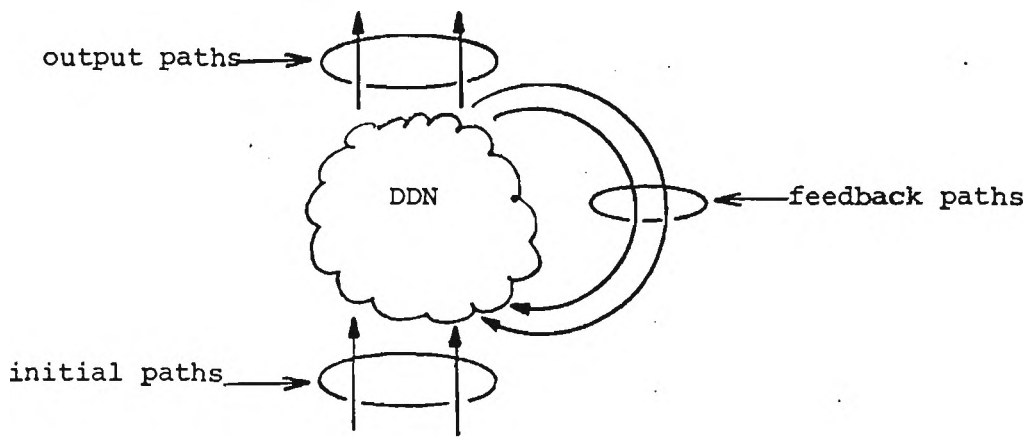


Figure 7: Data-Driven Iteration

b) How to separate possible pipelined items on the initial data paths from the feedback items.

The GATE cell is used to prevent non-deterministic merging of data paths in iterative situations. Halting of the iteration is implemented by the joint use of DISTRIBUTE, and OPERATOR cells. A sample iteration is shown in Figure 8, which increments a value iteratively until it becomes 3, and outputs it. Data items which are not delimited by parentheses are of type CONSTANT and are therefore not destroyed by the firing of a cell. In this manner constants are treated as part of the cell function rather than as a special token type, which

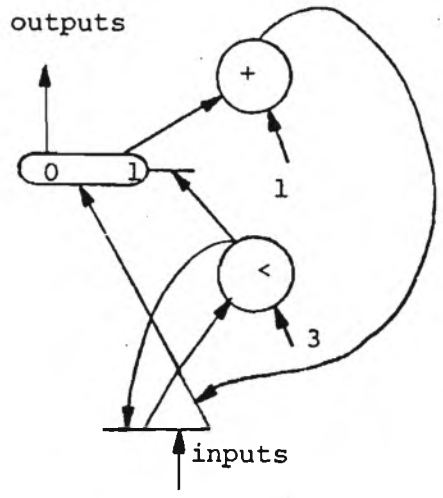


Figure 8: A Simple Iterative Net

is transmitted over the data paths.

A sample execution sequence for the net of Figure 8 is shown in Figure 9. Note the operation of the GATE and DISTRIBUTE cells with respect to the pipelined initial inputs. Every step of the sequence is not detailed, but the basic progression is shown.

These examples have illustrated the basic uses of the OPERATOR, GATE, SELECT, and DISTRIBUTE cells. Other cell use examples will appear later.

IV. The DDN Process Model and Call Mechanisms

There are many DDN structures which exhibit meaningless or erroneous behaviors. One reason is that a DDN can be considered to be any collection of cells connected by directed data paths which satisfies the following:

1) No directed data path may originate from a cell input.
2) No directed data path may terminate in a cell output.
3) No two distinct data paths may terminate at the same cell input.
4) Every cell of a DDN must have at least one data path connecting it with at least one other cell of the DDN.
5) Every data path must originate at a cell output, or terminate at a cell input, or both.
6) Every cell must have at least one output path.
7) Every cell must have at least one input path.

Rules 1 and 2 prohibit data items from flowing in the wrong direction; Rule 3 prevents the nondeterministic merging of data paths; and Rules 4 and 5 prevent the occurrence of isolated cells, subnets, or data paths. An isolated data path

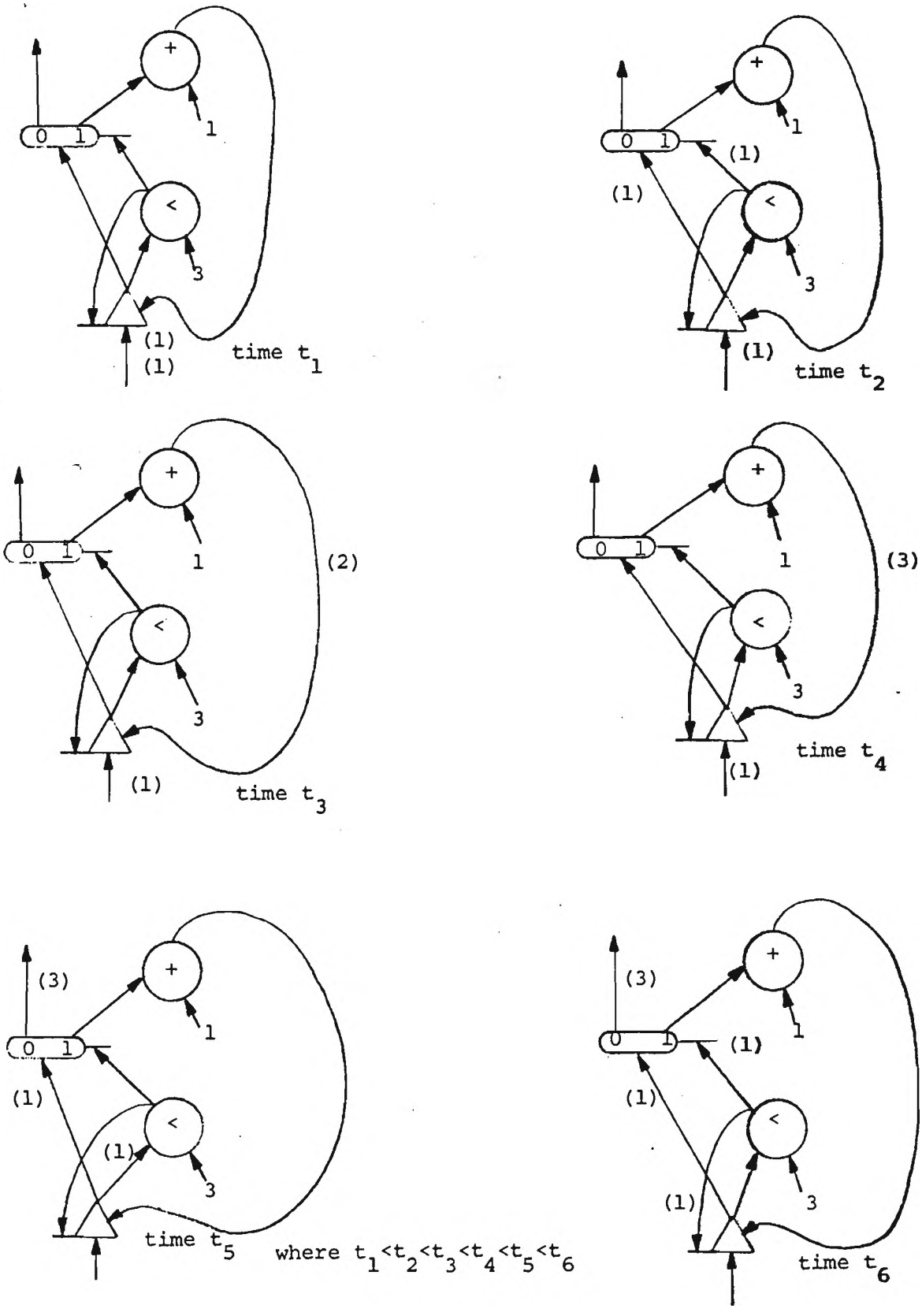


Figure 9: An Iterative Pipelined Sequence

is meaningless, and isolated cells or subnets are considered to be separate DDN's. A data path with no source can be used to indicate that the data items to be placed on it may be marked as constants or supplied by name from some environment. Data paths with no destination cell may indicate that data items are to be delivered to the environment by name, or that they are to be destroyed as in Figure 6. A cell with no inputs can never fire and is therefore useless, as is any cell with no outputs. Rules 6 and 7 prevent these last two possibilities. Yet certain DDN's allowed under these rules are meaningless, as shown in Figure 10.

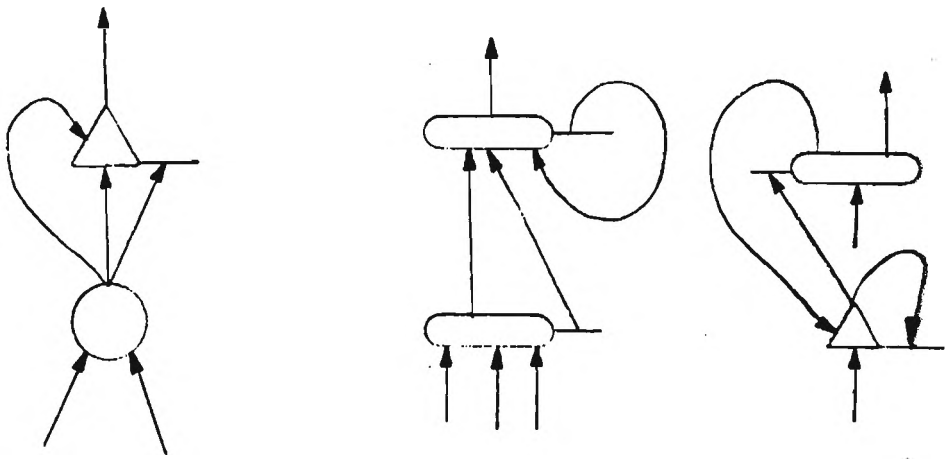


Figure 10: Meaningless DDN topologies

Extending the set of rules to eliminate all other forms of unwanted behavior would result in an unnecessarily long and complex set of rules, and might prohibit some useful net structures. It is therefore important to verify correct DDN behavior. This can be accomplished by the following sequence of actions:

a) Defining a process form which can be used as an abstraction aid, and thereby delimit the DDN which is to be analyzed.

b) Define a set of properties which guarantee that if a DDN has these properties then it will behave in a certain manner.

c) Produce a method by which DDN processes can be analyzed to determine whether these properties hold or not.

d) Prove that these properties guarantee the desired behavior.

To avoid undue complexity here, only a and b of this sequence will be covered.

DEFINITION: A data-driven process (DDP) is a triplet of the form {DDN, INPUT SYNCH CELL, OUTPUT SYNCH CELL}.

This form is illustrated in Figure 11.

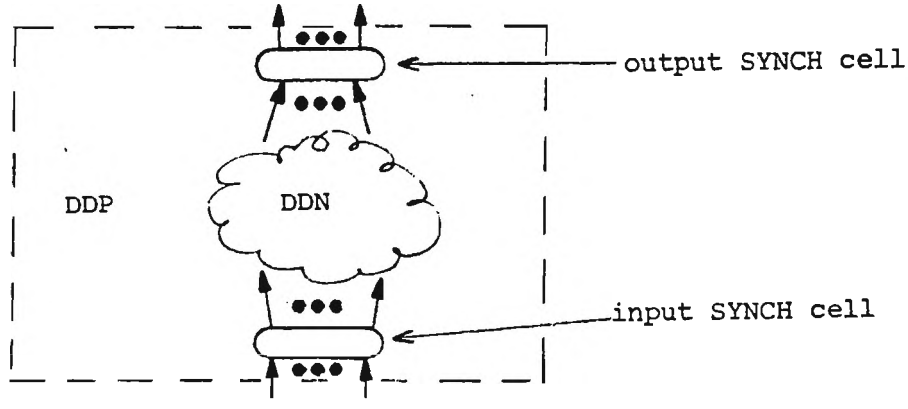


Figure 11: Data Driven Process (DDP)

This simple DDP model has many advantages. It is difficult to say when a DDN has started or terminated, since DDN's may receive inputs and produce outputs in any number of places. The two-terminal process form simplifies things considerably. The single input SYNCH cell acts as a collector for the DDP's working set, and the single output SYNCH cell collects the DDP results. This makes it possible to say where DDP's terminate and where they begin. In addition, DDP's have exactly the same firing characteristics as simple OPERATOR cells. When the input SYNCH cell becomes fireable, then the DDP is considered to be fireable. When the output SYNCH cell has fired, and when no cell in the DDP (i.e. within the enclosed DDN) is fireable, then the DDP is said to have terminated. During the time between firing and termination, the DDP is said to be active. Before and after this time the DDP is inactive.

Under pipelining, the definitions of termination, active, and inactive must be modified. Pipelined operation inherently implies that another set of input data may arrive at any time, and in this sense pipelined DDP's may never really terminate. Certain pragmatic considerations such as resource allocation require that some sort of termination definition be made. Since DDP's are output

functional, an instance of firing can be defined to be the firing of the input SYNCH cell, and an instance of termination may be defined as the firing of the output SYNCH cell. A more general definition could be: A DDP is said to have terminated whenever the number of input SYNCH cell firings equals the number of output SYNCH cell firings, and when no fireable cells remain in the DDP. All of these definitions are stronger than they might be if certain error situations were not taken into account. For instance, defining termination to exist whenever no fireable cells exist is valid except when the net hangs. This occurs when no fireable cell exists and the output SYNCH cell has not fired. A DDP which does not produce outputs when fired is considered to be in error. The converse possibility is that, due to the completely asynchronous nature of DDP's, the output SYNCH cell may fire, with some remaining data items left in the DDP that have yet to be "cleaned up" by some subsequent cell firings. The property of clean termination need not be required, but in general is an important property for pipelined situations (discussed in section VII).

Since a DDP exhibits the same behavior as a simple OPERATOR cell, a clean substitution rule can be formulated: within any DDN, a DDP which performs a function F may be substituted for any OPERATOR cell performing F without changing the functional behavior of the original DDN. This substitution rule allows a call mechanism to be defined (the CALL cell), which allows for recursive and/or hierarchically defined DDN's and DDP's. The CALL cell is used in DDN's to call DDP's. The name of the called DDP is indicated inside the CALL cell box.

There are two ways to implement non-recursive calls in DDN's: 1) open call (macro substitution type), and 2) closed call (pass parameter list type). With the open call, the firing of the CALL cell corresponds to substitution of the called DDP for the CALL cell in the net. This expanded net can then be executed in the normal manner. After the output SYNCH cell of the called DDP has fired, the inserted net can be removed and the net may contract to its original form. This removal of the inserted DDP is not necessary and its usefulness depends strictly on the pragmatic considerations of storage management and pipelining possibilities. Substitution of the CALL cell is easily accomplished by substitution of the DDP's output SYNCH cell destinations for the CALL cell's output destinations, and similarly for the input paths. The advantage of the closed call is that it allows commonly called DDP's to be shared. The closed

call mechanism can be implemented as follows:

a) When the CALL cell is fireable a message is formed containing the location of the CALL cell, the name of the called DDP, and the firing set of the CALL cell.

b) This message is then sent to the set of physical resources which will execute the called DDP.

c) A copy of the DDP named in the message is brought to the executing resource (if it is not already there) and the DDP is executed.

d) When the output SYNCH cell of the called DDP fires, a message is formed containing the firing set of the output SYNCH cell. This message is then sent back to the calling net's CALL cell, the location of which was sent in the previous message.

There are also several choices for implementing recursive calls. One method is to implement a recursive call by repeated insertions of the called net as described for open calls. Another method is as described for closed calls, where each recursive call initiates a closed call to a new copy of the called net. Both of these methods require a new copy of the DDP to be made for each level of the recursion. This would require a large amount of storage, and therefore these two methods are considered unsuitable. The third method is to use an approach similar to the use of "colored tokens" in Petri Nets [12].

The contents of the DDN data path queues can be considered to be the net marking. The colored token type of call may be implemented by allowing such markings to be stacked as follows:

1. The current marking is pushed down leaving a new current marking at the top of the stack.
2. The new current marking places the inputs of the recursive CALL cell to the inputs of the input SYNCH cell of the DDP.
3. The resulting DDP execution is performed in the normal manner.

On return from a recursively called DDP:

1. The marking stack is popped.
2. The outputs of the output SYNCH cell of the called DDP are sent as outputs of the recursive CALL cell.
3. Normal processing resumes.

Using this method, each data path appears as a stack of queues. The top queue element of the stack will be active during the execution of the DDP at the current level of recursion. If the recursion goes deeper, then that level will be pushed down and become dormant, and a new top level will become active. On a return all of the top element queues will be popped. If any popped queue is not

empty (except for the output SYNCH cell's queues) then an unclean termination of that recursive call has occurred, and the DDP is considered to be in ERROR.

The bottom queue element of each stack may also be active due to pipelining. In this respect the actual data structure which exists on a data path is a deque of elements, each of which is a queue. If special hardware (such as that provided in DDM1[3]) does not exist, then this data structure may prove to be too difficult and inefficient to maintain. A method which greatly simplifies this difficulty is to restrict pipelining to not occur in recursively called DDP's. This can be done as shown in Fig 12. A non-constant data item is initially placed on the feedback path. This item is consumed on every entry to the recursive DDP and thereby prevents further pipelined firing sets from passing until the recursion terminates and produces another item on the feedback path.

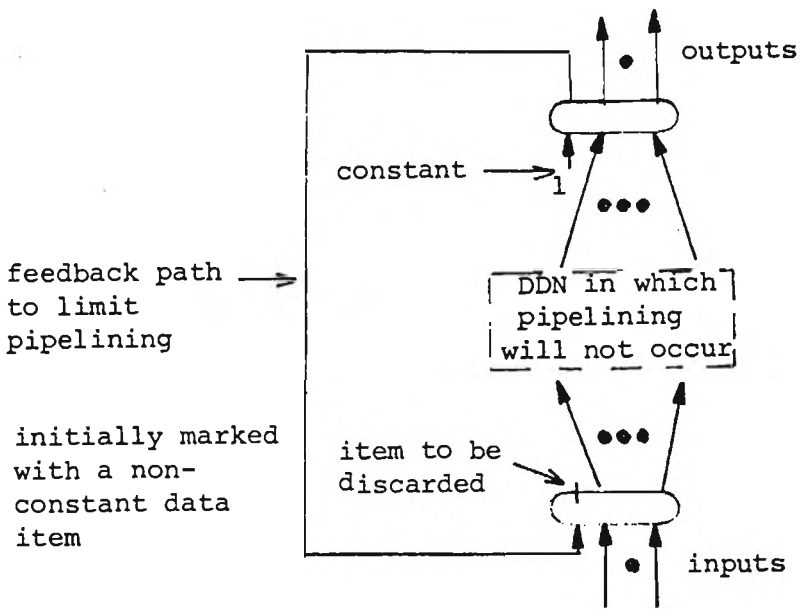


Figure 12: Limiting pipelining to DDP's

While the DDP model has some very nice properties with respect to abstraction, analysis, substitution, and hierarchical structure, it is more limited in what it can represent than the more general DDN's. Consider the egg boxing factory of Figure 13. The factory has two inputs: 1) a conveyor belt of eggs, and 2) a conveyor belt of egg cartons. The single output is a belt carrying cartons of one dozen eggs each. Figure 14 shows the DDN representation of this factory process.

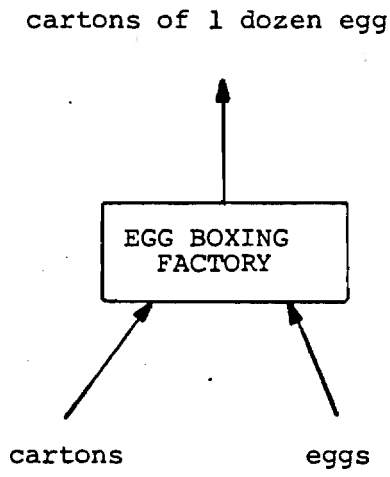


Figure 13: Egg boxing factory

While the egg boxing factory can be modeled in a very straightforward manner by a DDN, it cannot be represented by a DDP. The fact that 12 of the "EGGS" input data item and one of the "CARTONS" inputs are required to produce a single output item makes it impossible to find a DDP representation for this situation. One general problem with DDP's is that for every set of input items consumed, a set of output items is produced. The sets need not be of equal size and since individual data items may have substructure, this restriction is usually not a problem. However, in the case of the egg boxing factory, it is not appropriate to require that each "EGGS" input contain exactly 12 eggs. Even though they are less general than DDN's, DDP's provide a convenient vehicle for discussion of some important properties for computational models of this type.

V. DDP Examples

Examples are given in this section which illustrate the use of the data-driven cells, process definition methods and various forms of net behavior. The readers are encouraged to actually "play" the nets by physically moving their favorite form of token around the data paths.

In order to illustrate the representational advantages of DDN's over other data-flow representations, an example given by Dennis and Misunas in [5] is shown in Figure 15 and compared with the simpler but fuctionally equivalent DDN of Figure 16. Both nets distribute incoming data items uniformly onto 8 output paths. The DDN version is simpler, allows increased concurrency under pipelined

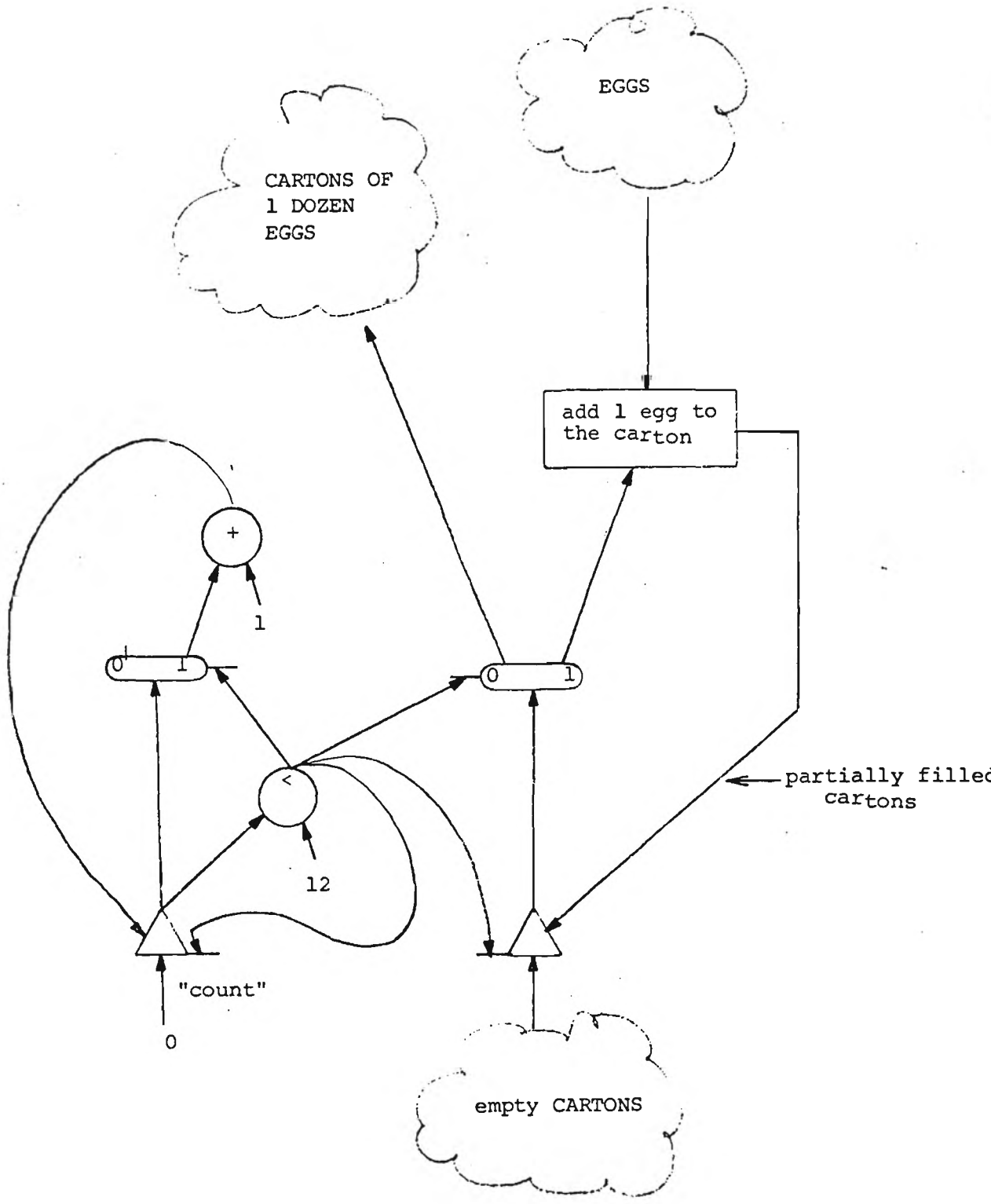


Figure 14: Egg Boxing Factory DDN

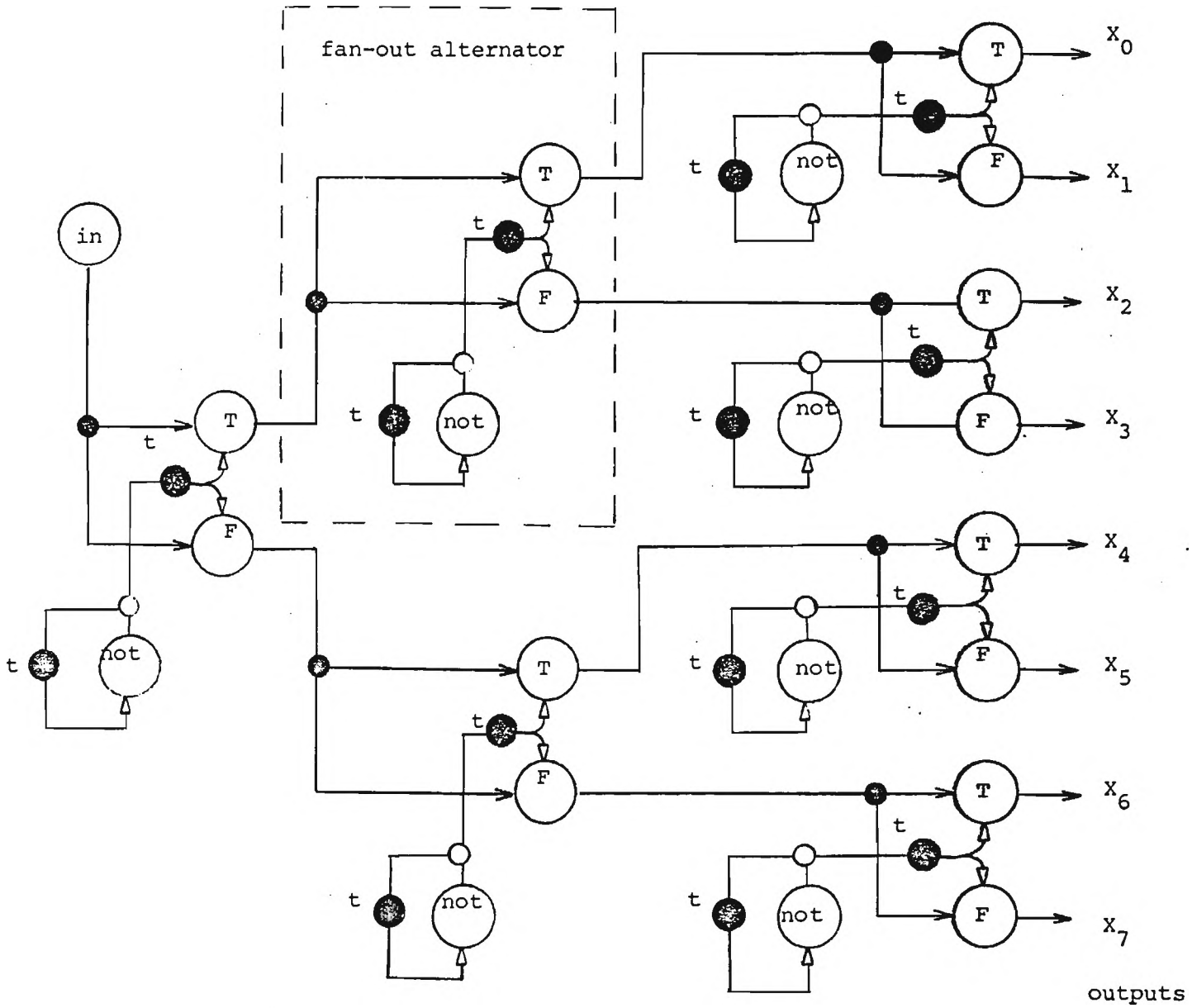


Figure 15: Dennis tree of fan-out alternators

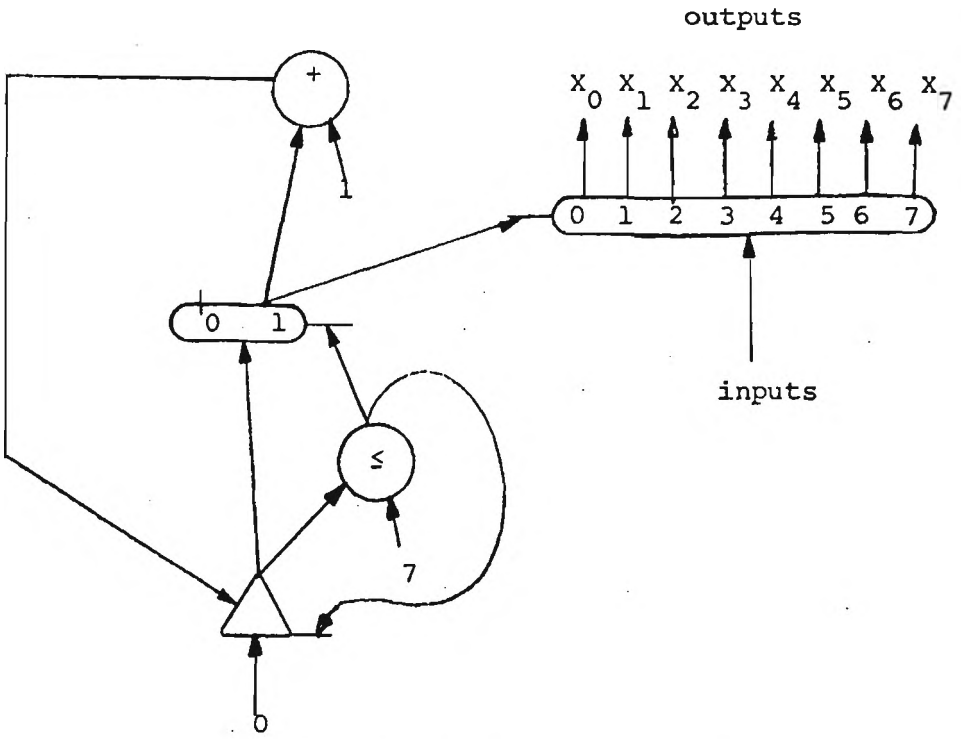


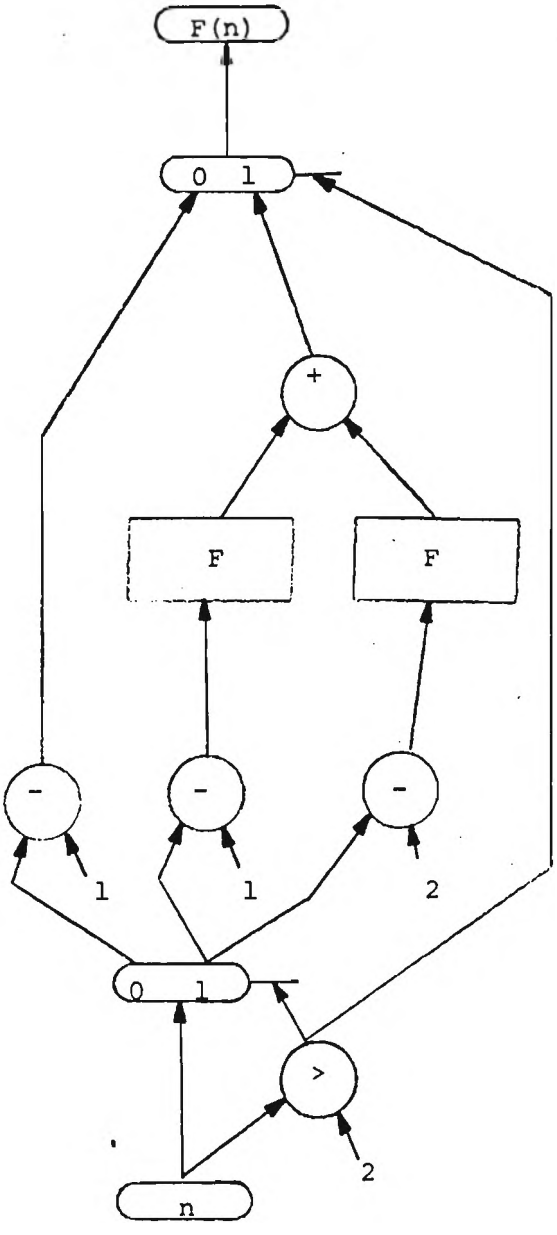
Figure 16: Sample Distribution DDN

operation, and requires no initial marking to be generated as in the net of Figure 15.

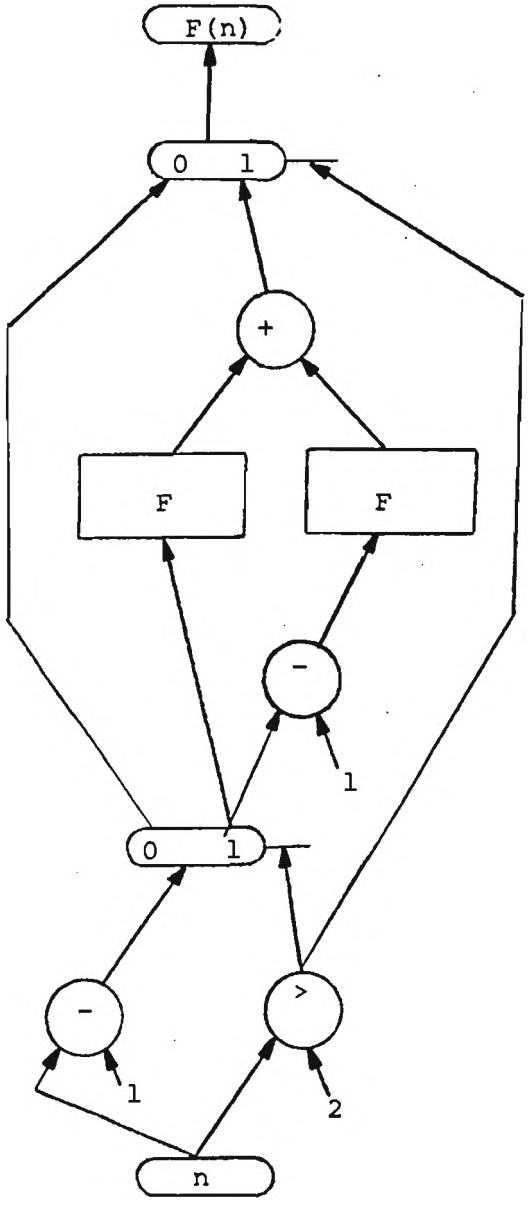
Figure 17 shows two nets, each containing two parallel recursive calls to calculate the nth Fibonacci number, for positive integers n. 17a shows the obvious net, while 17b shows a net which will execute as fast with two processors as 17a does with three processors (assuming that > and - operations require equal time to compute and that execution speed is mainly dependent on critical path length). Figure 17b takes advantage of the fact that the incoming n gets decremented regardless of its value.

Most of the data-flow net programs shown in the literature are small cook-book tutorial problems which do not really give a feel for the complexity of data-driven programs in general. The following ALGOL program generates recursively all solutions to the eight queens problem. The algorithm is essentially the same as in Wirth[11]. A data-flow solution to the eight queens problem has also been given by Dennis[4], which serves as a further comparison of the two schema.

Find the nth Fibonacci number:
where F(1) = 0
F(2) = 1
F(n>2) = F(n-1)+F(n-2)



a) obvious net



b) same speed but requires only 2 Processors for maximum speed

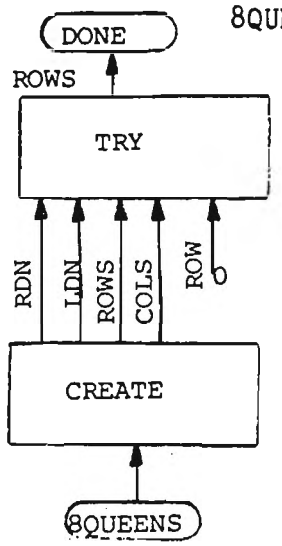
Figure 17: Fibonacci DDP's

```
BEGIN     %PROGRAM TO FIND ALL SOLUTIONS TO THE 8 QUEENS PROBLEM
     INTEGER ARRAY ROWS[0:7];
     BOOLEAN ARRAY COLS[0:7], RDNDIAG[-7:7], LDNDIAG[0:15];
     INTEGER I;
     FILE TTY(KIND=REMOTE, MYUSE=IO);
     %MAIN RECURSIVE PROCEDURE IS TRY
     BOOLEAN PROCEDURE TRY(ROW);
          VALUE ROW;
          INTEGER ROW;
          BEGIN
               INTEGER COL;
               FOR COL:=0 STEP 1 UNTIL 7
                    DO
                         IF COLS[COL] AND LDNDIAG[ROW+COL]
                                      AND RDNDIAG[ROW-COL]
                              THEN
                                   BEGIN
                                   ROWS[ROW]:=COL;
                                   COLS[COL]:=FALSE;
                                   LDNDIAG[ROW+COL]:=FALSE;
                                   RDNDIAG[ROW-COL]:=FALSE;
                                   IF ROW < 7
                                        THEN
                                             TRY(ROW+1)
                                        ELSE
                                             WRITE (TTY,<8I2>,
                                             FOR I:=0 STEP 1
                                                   UNTIL 7
                                             DO ROWS[I]);
                                   COLS[COL]:=TRUE;
                                   LDNDIAG[ROW+COL]:=TRUE;
                                   RDNDIAG[ROW-COL]:=TRUE;
                                   END
          END TRY;
     %INITIALIZE THE ARRAYS AND CALL TRY
```

```
    FOR I:=0 STEP 1 UNTIL 7 DO COLS[I]:=TRUE;
    FOR I:=0 STEP 1 UNTIL 15 DO LDNDIAG[I]:=TRUE;
    FOR I:=-7 STEP 1 UNTIL 7 DO RDNDIAG[I]:=TRUE;
    TRY(0)
END.
```

Much of the complexity of the equivalent DDP solution is due to the low-level nature of the DDN representation. The DDN schema is best viewed as a machine language. In this light the resulting complexity is not so bad. The equivalent machine language program is 258 instructions on the B6700, a machine which is well-suited for executing recursive ALGOL programs such as this one.

The equivalent DDP solution is shown in Figure 18.



8QUEENS: A recursive DDP for generating all solutions to the 8 queens problem.

Figure 18a: 8QUEENS NET

CREATE - creates the vectors LDN, RDN,, ROWS, and COLS and initializes them.

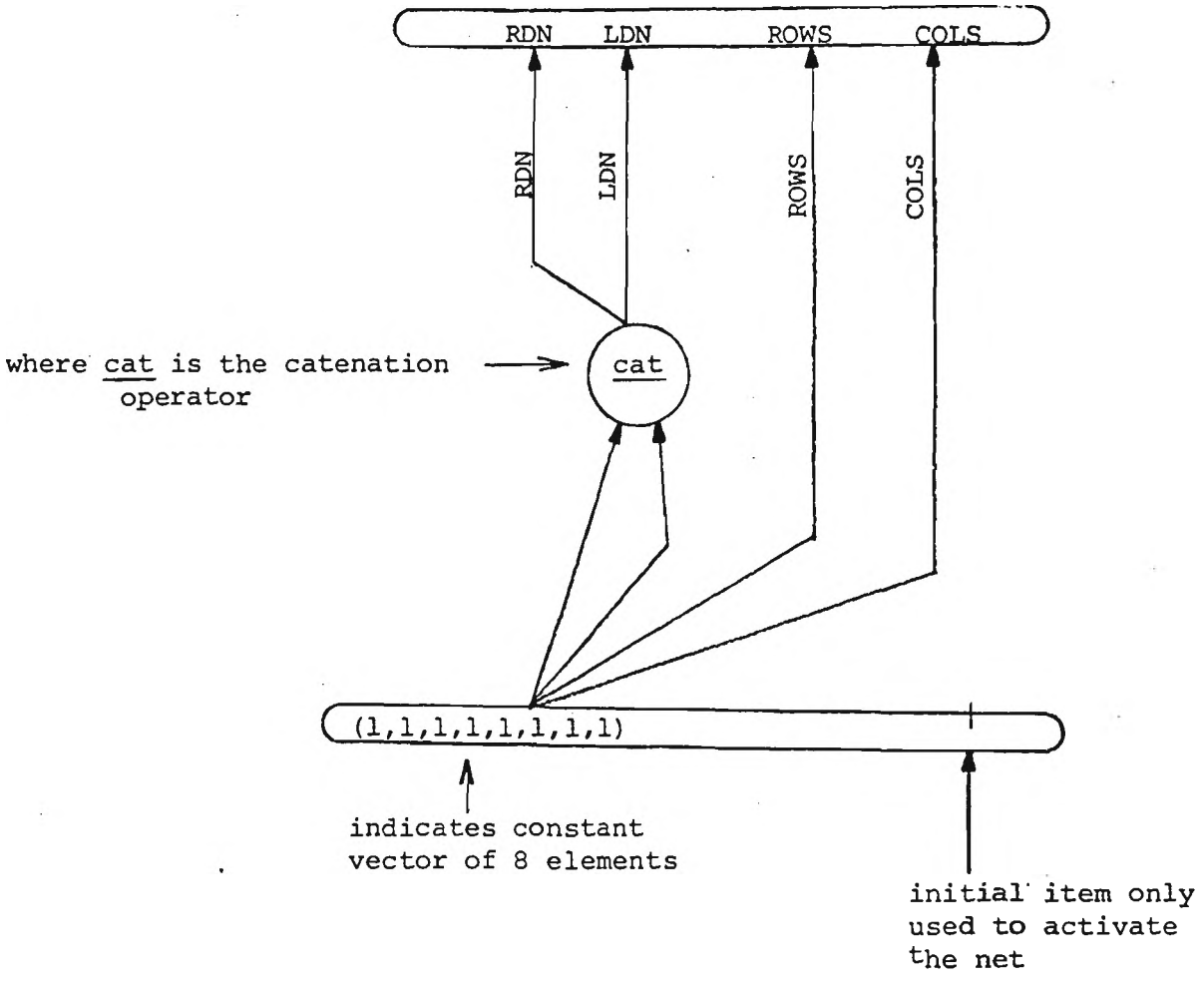


Figure 18b: 8QUEENS CREATE DDP

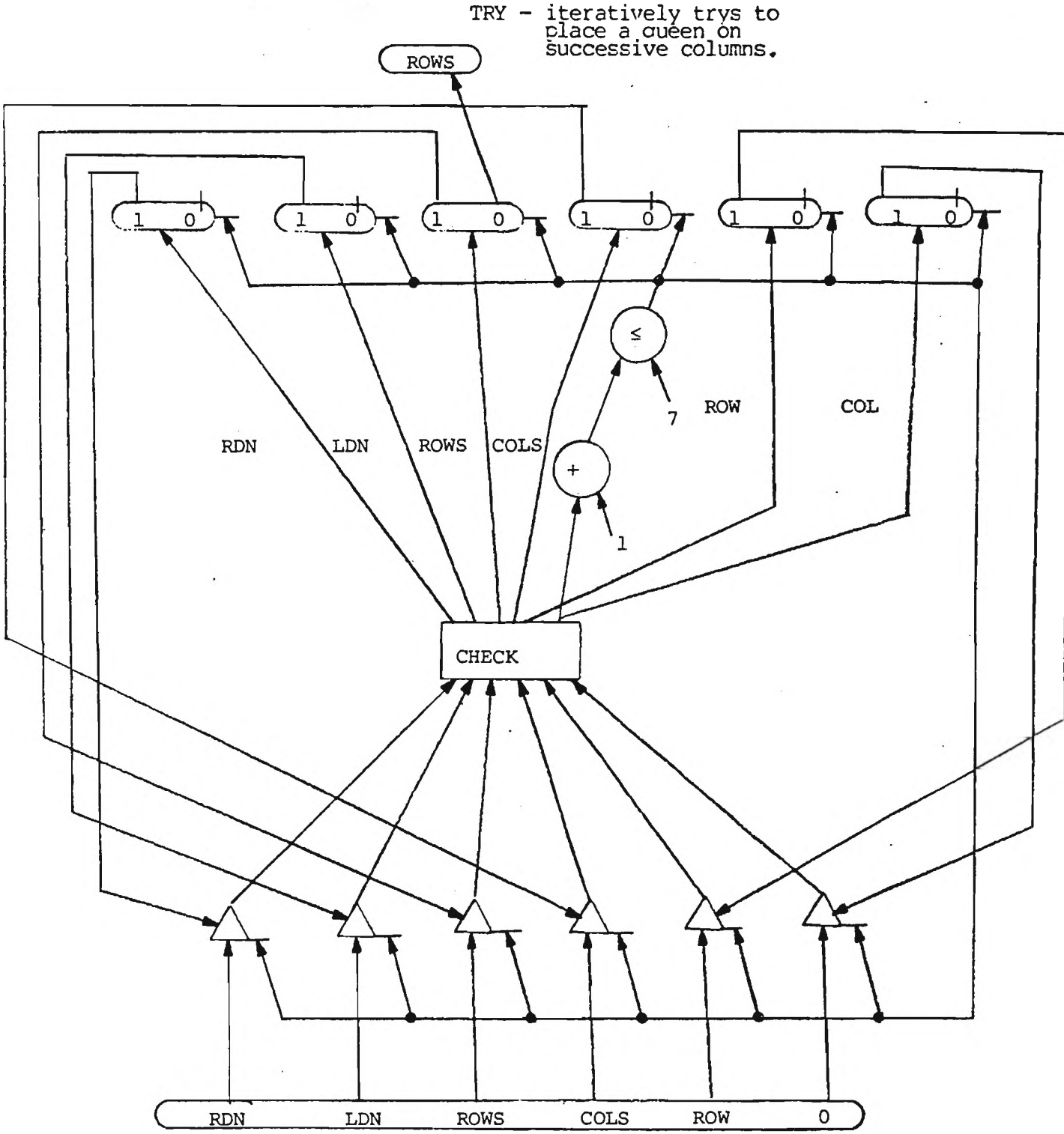


Figure 18c: 8QUEENS TRY DDP

CHECK - checks the current queen placement and allows recursion to advance if that position is safe.

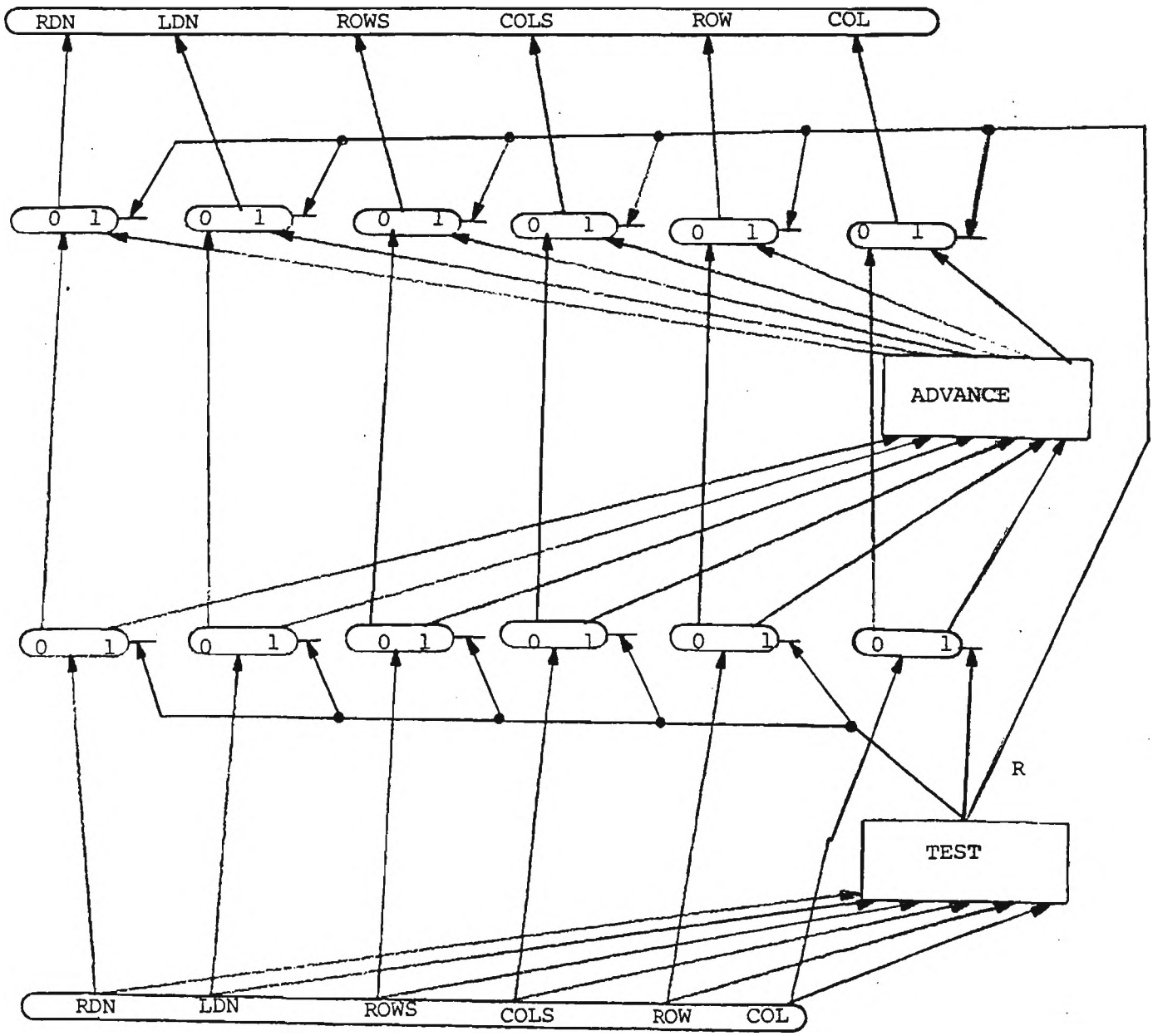


Figure 18d: 8QUEENS CHECK DDP

TEST - actually subscripts the vectors and does the compares to see if current position is safe.

Note: [is the subscript operator, the convention here is that the subscript expression arrives on the right data path and the structure arrives on the left data path.

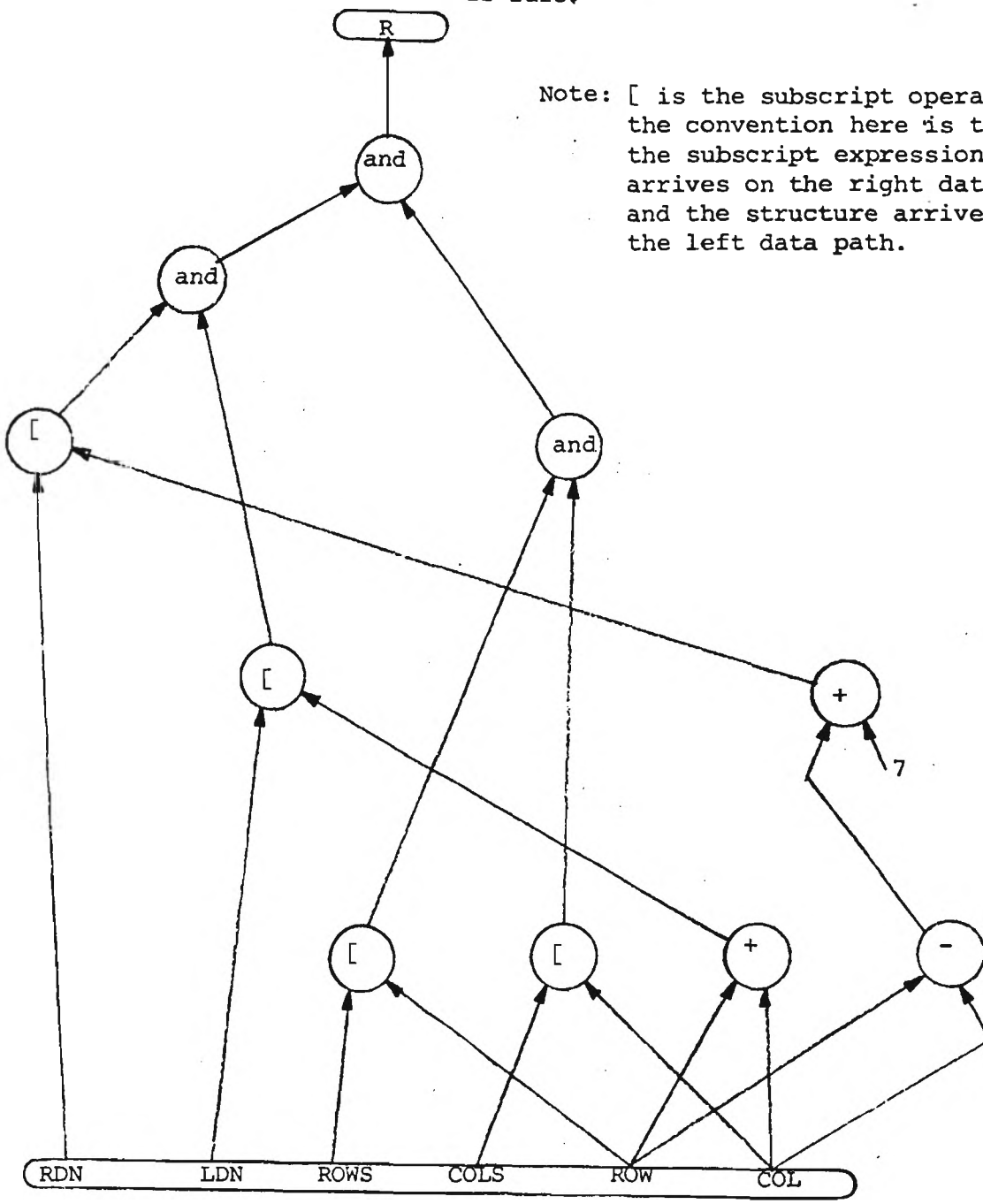
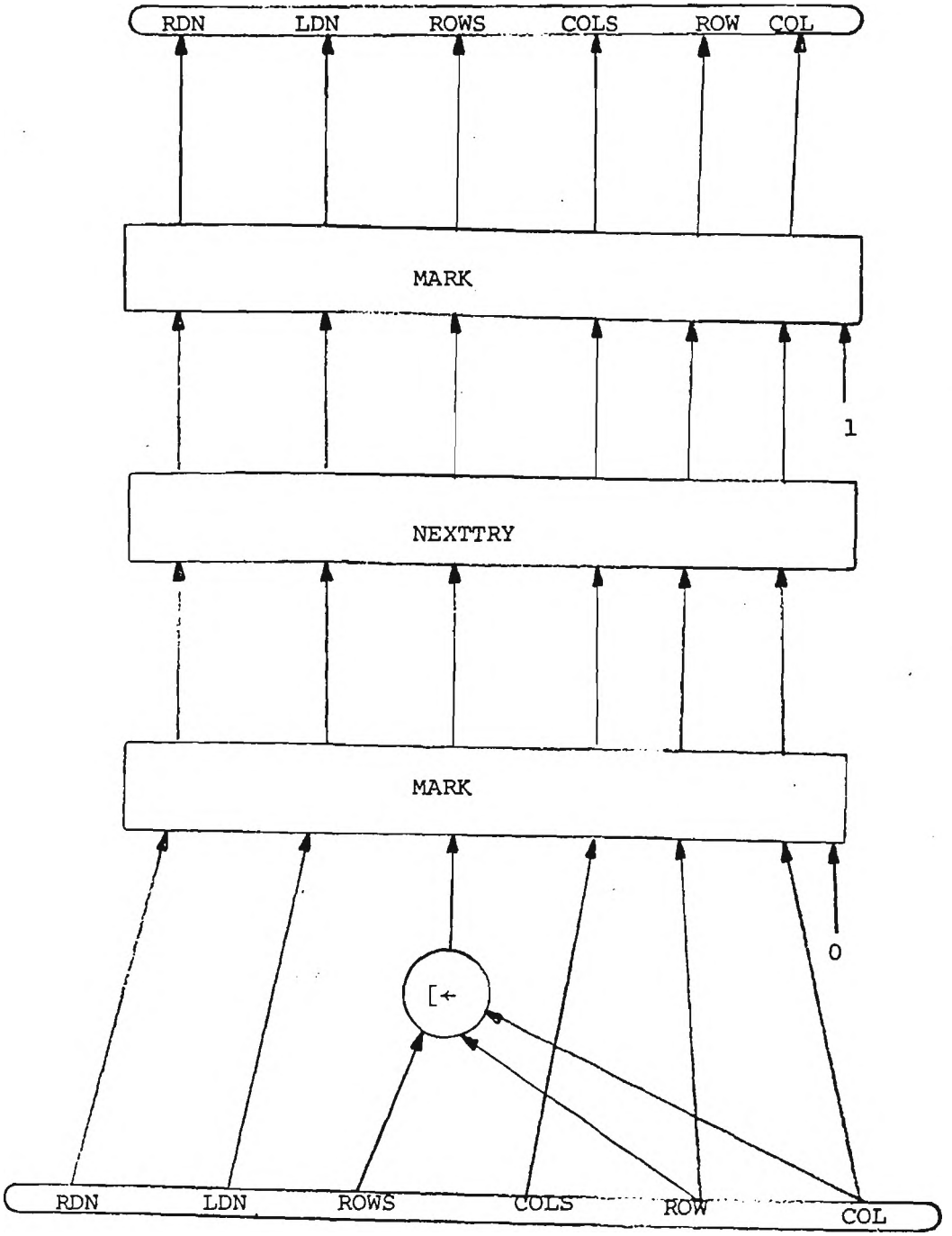


Figure 18e: 8QUEENS TEST DDP

ADVANCE - advances the recursion



Note: [← is the subscripted write, the convention here is that the structure is on the left, the subscript expression is in the middle, and the write value is on the right.

Figure 18f: 8QUEENS ADVANCE DDP

MARK - marks the queen postion

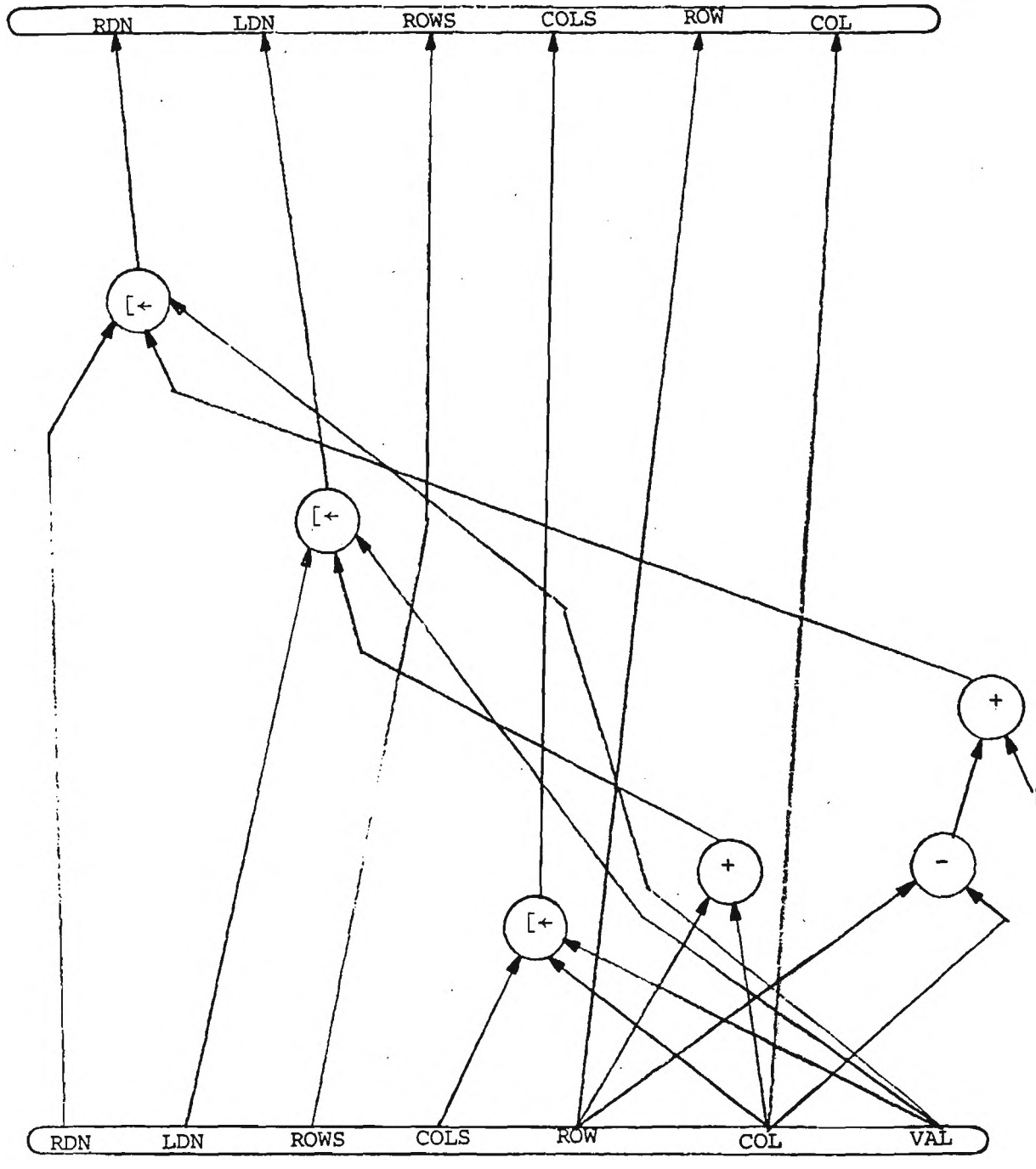


Figure 18g: 8QUEENS MARK DDP

NEXTTRY - bumps the row count
and recurses

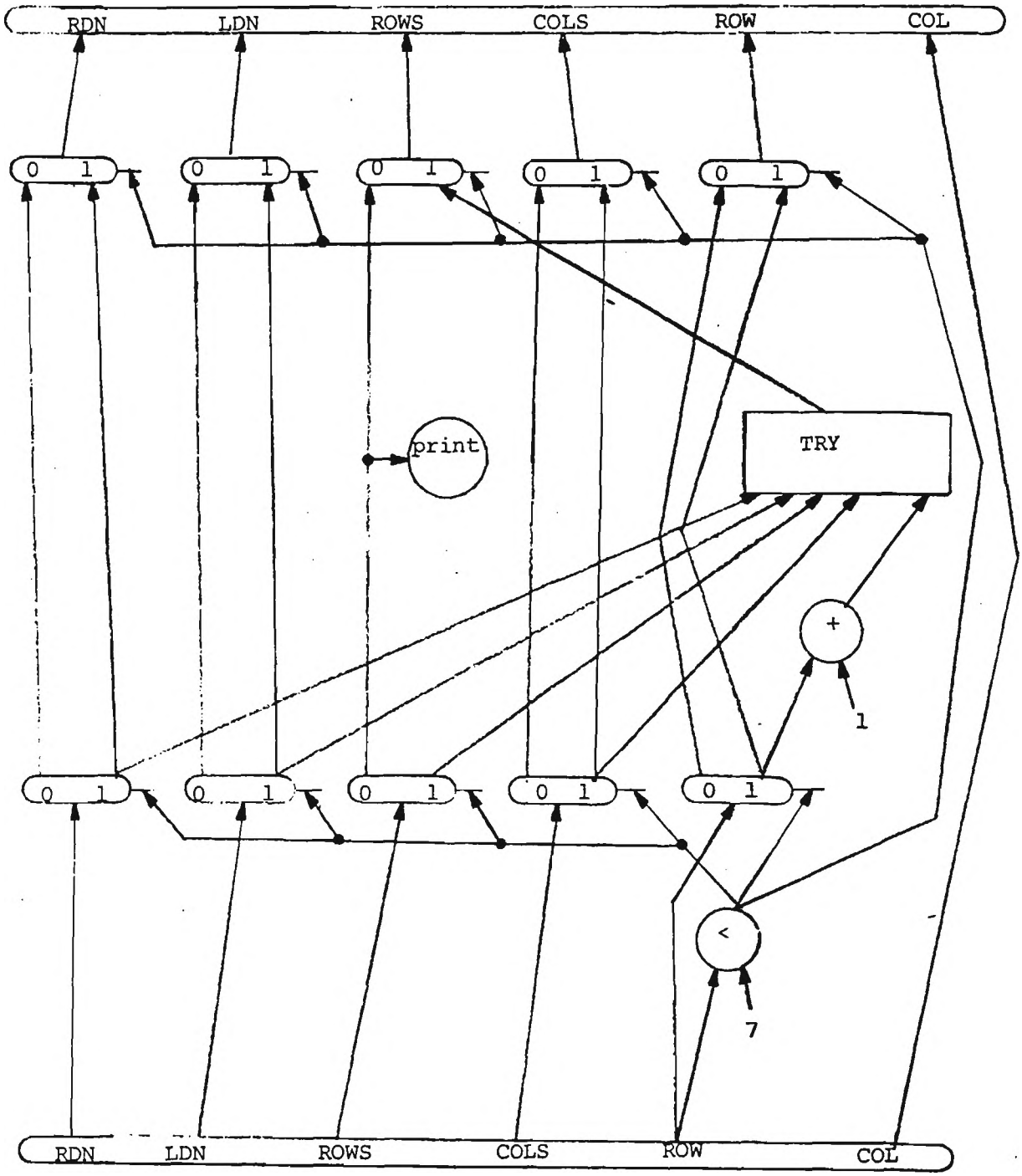


Figure 18h: 8QUEENS NEXTTRY DDP

A careful examination of the 8QUEENS DDP will reveal the usage of all the DDN cell types except the arbiter. This net also points out some of the pathological problems encountered in DDN processes. The major problem is that all non-atomic data structures must be destroyed every time a single element of such structures is accessed. Such copies of large data structures present a serious problem in terms of both time and space.

A detailed discussion of data structure handling is beyond the scope of the issues discussed here. It is appropriate however to mention a few considerations relating to a better method. One can consider DDN's to consist of two files: 1) a static file (so far- the net description), and 2) a dynamic file (until now - the data items). A more general way is to allow the data item file to be either the static or the dynamic file (and similarly for the net description). The basic nature of data-driven computation indicates that the dynamic file elements will be destroyed upon cell firing, and therefore some copying will be necessary. The proper choice for the dynamic file would be the file (data item or net), which would minimize the copying requirements. In instances where large data structures are used, the static file would be the data structures and the net description would be the dynamic file. In this instance the data structure would be treated as a static resource which could then be shared by a number of concurrent processes. To avoid the possibility of access conflicts to the structure, an ARBITER cell can be used to guarantee first come first served (but sequential) access to the structure. In case of a tie, any pending request is chosen randomly.

Figure 19 shows a net for controlling the shared reads of a vector. The inputs P1, P2, and P3 are the indices from concurrent processes 1, 2, and 3 respectively. The vector input is the vector to be loaded into place. It is assumed that the load input arrives before any of the Pn inputs.

The SHARED RESOURCE box of this net now acts as a sequential interpreter for instructions flowing into it. The net also shows how order-preserving parallel to serial to parallel conversion takes place using the arbiter and DISTRIBUTE cells. The DDN ARBITER cell does not perform just the normal arbiter function, but also generates an index indicating which input was selected. This index preserves sufficient state information to allow the sequenced items to be

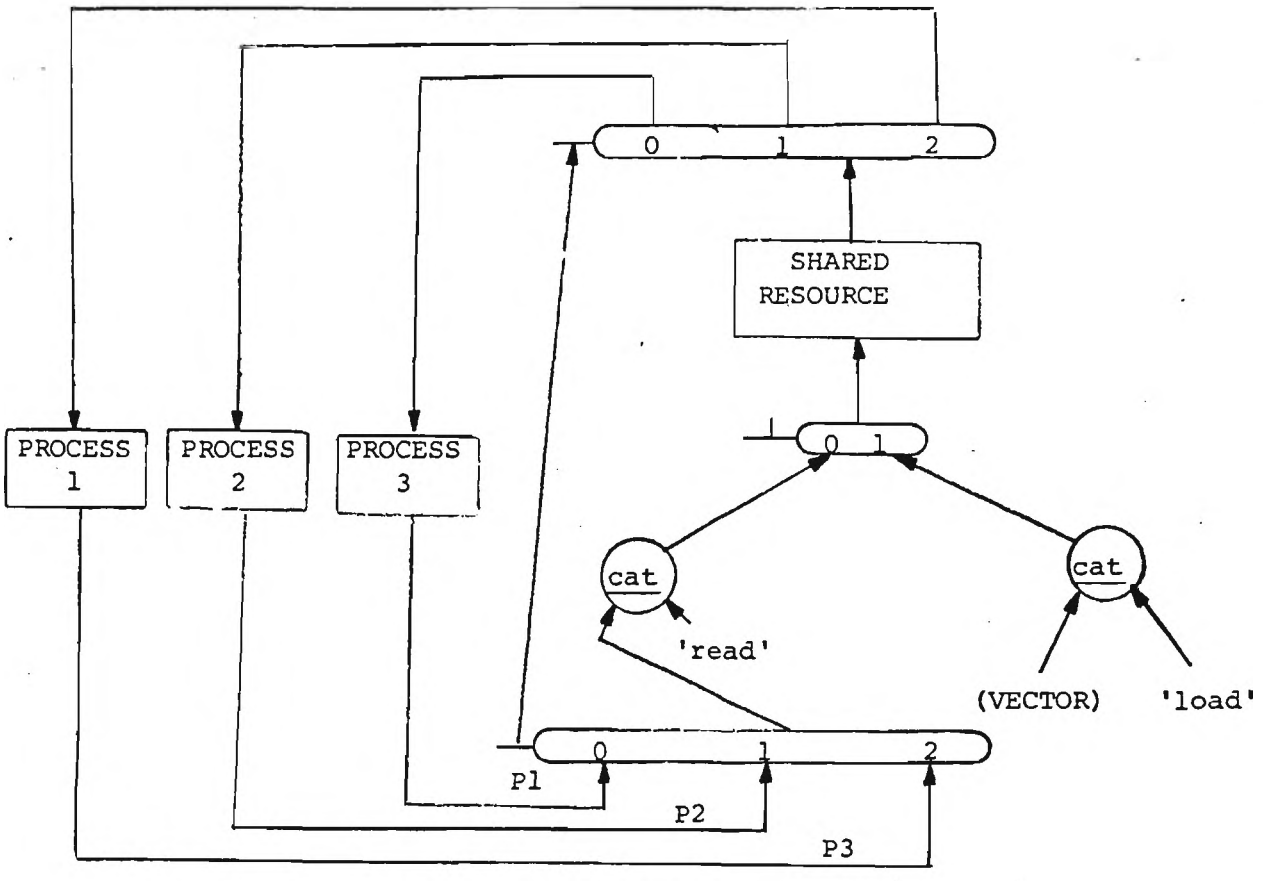


Figure 19: Shared Resource DDN

correctly "reparallelized". Any time an ARBITER cell is used in a net, it must be used in exactly the same ARBITER - DISTRIBUTE cell pair topology as shown in Figure 19. Otherwise the ARBITER cell will cause non-deterministic sequencing and the result will be a non output-functional net. Figure 20 illustrates how order-preserving serial to parallel to serial conversion is handled. Such important conversion capabilities do not exist in other data-flow representations.

VI. Errors

The basic nature of data-driven processes is that operations are "pushed" into action by the arrival of the required set of inputs. If one of these inputs is prevented from arriving at the intended destination (due to a programming problem or other type of error), then that destination cell will never fire. Consequently, all cells having firing sets containing outputs from the unfireable cell wll never fire and so on. A cell or a net which can never fire is said to hang. A cell or net which can never hang is said to be live. An example of a net which can hang due to poor programming is shown in Figure 21a.

In Figure 21a, if N is negative then R will be undefined and never receive a

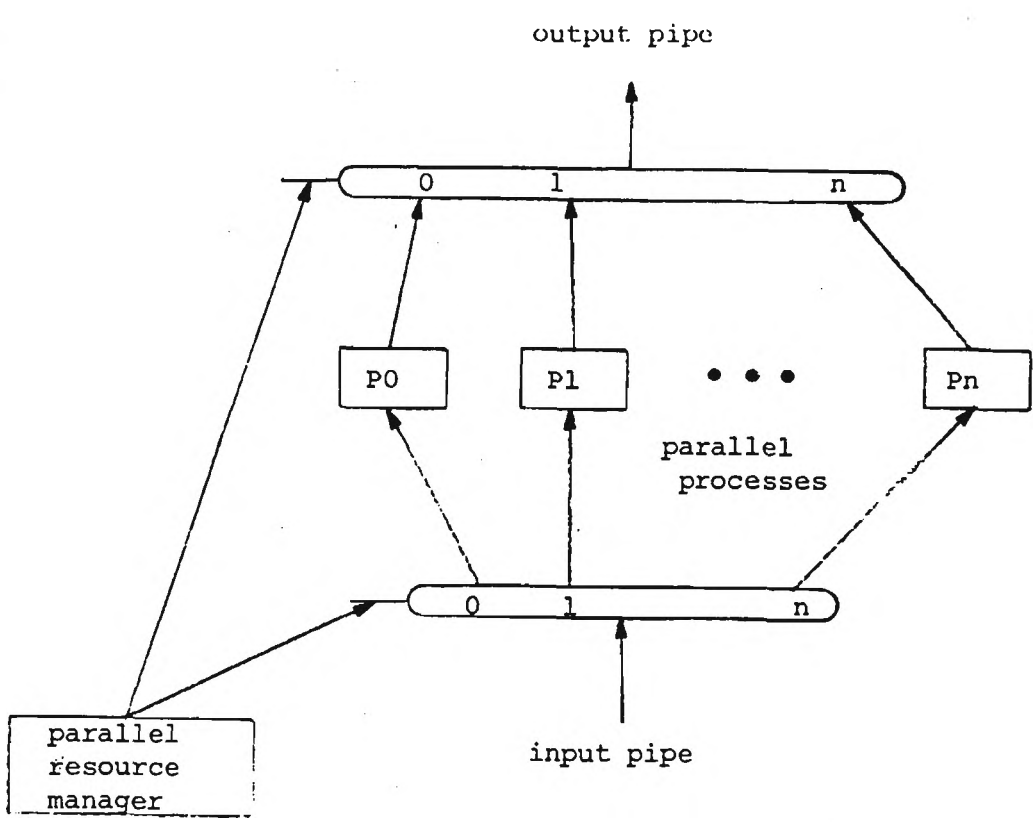


Figure 20: Serial to Parallel to Serial Conversion

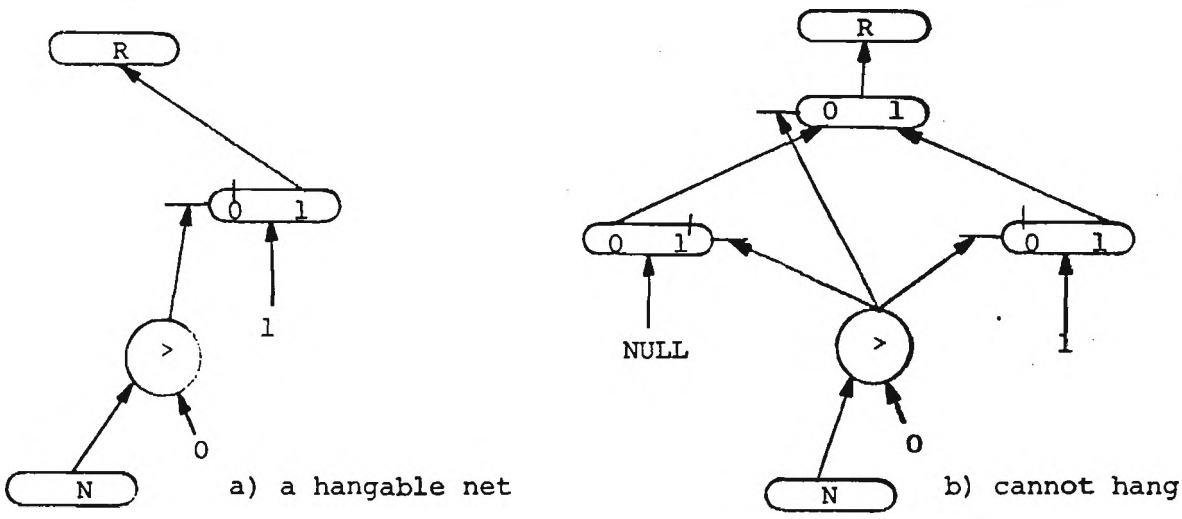


Figure 21: Correcting a hangable net

value. The output SYNCH cell cannot initiate a message in order to determine the status of the missing input, as that would be inconsistent with the data-driven firing rules. Since no cell can know whether it is waiting for an input that will never arrive (i.e. whether it is live or not), it is important to be able to guarantee liveness of DDP's by topological examination. Figure 21b indicates how a data item of special value NULL can be used to correct the problem found in

Figure 21a. While this illustrates the mechannism, the two DISTRIBUTE cells can actually be removed and the NULL and 1 inputs may be used directly as the inputs of the SELECT cell. Such a net would be functionally equivalent to the net shown, but contain fewer cells.

The output SYNCH cell of the DDP of Figure 21b will always receive an input, and is therefore live. If R=NULL as a result of executing the DDP, then the calling DDP may be programmed to invoke an error process, or whatever else is desired. The important thing is that the DDP produced some output.

The data-driven cells behave as follows with NULL valued inputs:

A1) The SYNCH, ARBITER, and CALL cells act in their normal manner.

A2) The GATE, SELECT, and DISTRIBUTE cells produce NULL tokens on all outputs, without destroying any inputs (except the conditional input NULL) when a NULL or out of range item arrives on the index or condition input.

A3) The GATE, SELECT, and DISTRIBUTE cells behave normally for NULL items on other inputs.

A4) If any input item is NULL then all output items are NULL for the OPERATOR cells except that the test (NULL = NULL) is TRUE.

NULL valued items are generated in two other ways:

B1) As a result of an illegal operation, such as divide by zero.

B2) Explicitly as in Figure 21b.

A consequence of rule A2 is that under pipelined operation the injection of NULL items may cause the various data streams to not match up in the desired fashion. The use of a SELECT cell and DISTRIBUTE cell pair to effectively "bracket" the condition, as shown in Figure 17, overcomes this problem.

It is possible for a compiler-like program to insert the NULL handling structure shown in Figure 21, or the programmer may describe such nets explicitly. The question of good style in data-driven programs is an importnt area but not one to be discussed here. The use of NULL valued tokens in DDN's allows the hung net disaster to be avoided, and other forms of error situations to be dealt with.

VII. Live, Safe, and Clean DDP's

When a conditional expression is described as a DDN, only one path of the condition will fire for a given set of inputs. For this reason, the notion of

whether a particular cell is live or not is not of much practical value, and in fact, it is impossible to topologically determine. Similarly for general DDN's the notion of liveness is somewhat nebulous, but for a DDP, liveness is an important and topologically verifiable property.

Two other important characteristics of DDP's is whether they are safe or clean. A DDP is said to be clean if when it terminates, there are no non-constant data items existing in the DDP. DDP's are clean when they are defined. If they were not, then the output values would be history dependent upon the values of the existing non-constant data items. A live DDP which when it terminates without error and is always clean is said to be safe. It can be shown that safe DDP's execute in an output functional manner under pipelining.

It is possible to determine by topological analysis of any DDN whether it is safe or not. The machine algorithm for such analysis is lengthy and will not be presented here. Such analysis would be an important part of a DDN compiler, and should be performed before execution of any DDP.

VIII. Conclusions

A low-level parallel process representation has been presented which can be used as a basis for the representation of parallel programs and adapts nicely to execution by fully distributed hardware resources. These data-driven nets have several advantages over existing data-flow representations and several properties of DDN's have been examined. The only sequencing rule of DDN programs is that of data dependency, and since no weaker sequencing relation is possible (without doing non-productive operations, Linderman [8]), DDN's naturally yield a maximally concurrent version of a given algorithm. DDN's behave on a completely local basis and operations do not share any common global environment. This and the transparancy of influence in DDP's, facilitates formal verification of process correctness, which becomes even more important in fully distributed systems, in that it is typically impossible to recreate the situation which caused the error. These attributes, when combined with the asynchronous nature of DDN's allow any DDN to be pared into an arbitrary number of subnets without altering the functional operation of the overall net. This affects freedom and ease of assigning parallel subtasks to available parallel resources for execution. Further concurrency can be obtained by pipelined operation of these

nets.

While some problems still exist with the data-driven approach, DDN's appear to be an attractive representation for distributed computing.

References

1. Adams, D.. A Computation Model with Data Flow Sequencing. Stanford University Technical Report, CS-117, (1968).

2. Arvind, and K. P. Gostelow, and W. Plouffe. Programming in a Viable Data Flow Language. University of California at Irvine, Department of Information and Computer Science, Technical Report #89 (1976).

3. Davis, A. L.. The Architecture of DDM1: A Recursively Structured Data-Driven Machine. University of Utah, Computer Science Department, UUCS - 77 - 113 (1977).

4. Dennis, J. B.. First Version of a Data Flow Procedure Language. Lecture Notes in Computer Science, 19, Springer Verlag, New York (1974).

5. Dennis, J. B., and D. Misunas, and C. Leung.. A Highly Parallel Processor Using a Data Flow Machine Laguage. MIT Laboratory for Computer Science Memo 134 (1977).

6. Hansen, P. B.. Concurrent Pascal: a programming language for operating systems design. Cal Tech Information Science Report, Report 10 (1974).

7. Kotov, V.E., and A. S. Narinyani.. Asynchronous Computational Processes on Shared Memory. Kibernetika (in Russian), No. 3, Kiev (1966).

8. Linderman, J. P.. Productivity in Parallel Computation Schemata. MIT Project MAC, TR-111 (1973).

9. Rodriguez, J. D.. A Graph Model for Parallel Computations. MIT Technical Report, MAC-TR-64, (1969).

10. Seror, Denis. DCPL - A distributed control programming language. University of Utah Technical Report, UTEC-CSc-70-108 (1970).

11. Wirth, N.. Algorithms + Data Structures = Programs. Prentice-Hall, N.J. (1976), p. 145.

12. Zervos, C. R.. Colored Petri Nets: Their properties and applications. PhD Thesis, University of Michigan, Ann Arbor (1977).