A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture

by

Arvind
Kim P. Gostelow

Technical Report #72

October 1975

This work was partially supported by the Distributed Computing System project, under NSF grant GJ1045.

Revisions

Revised February 1976 (minor editorial changes).

1. page 17:line 18 "etc." to "etc. This is shown in Figure 5d."

2. page 20:line 6 "...(Fig. 5g)..." to "...(Figure 5g)..."

also:line 8 "fig." to "Figure"

also:line 11

also:line 16

"...to the end first ..." to "...to the end..."

page 4:line 8

"...come out..." to "...been produced..."

page 22:line 7

"P. Two procedures p_1 and P_2 (Figure 7a) are interconnected* (Figure 7b) by"

Abstract

The execution of a program may be viewed as the processing of a statement in a programming language by an underlying interpreter. This report discusses briefly the advantages of a data flow language over conventional programming methods, and then presents a new interpreter for a data flow language. Using as a base the data flow language of Dennis ["First Version of a Data Flow Procedure Language" Computation Structures Group Memo 93, Project MAC, MIT, Nov. 1973], the new interpreter magnifies the apparent asynchrony and speed of data flow, and it does so by (quite literally) exchanging blocks of processors for slices of time. The report gives details of the operation of this new interpreter, and identifies the consequences of the new interpreter on machine architecture and design.

ACKNOWLEDGEMENTS

We wish to thank Messrs Wil Plouffe, Larry Rowe, Paul Mockapetris, Allen Foodym, and Bob Thomas for their interest and comments on this report, and Dave Farber and the Distributed Computing Systems project for supporting the research described here. Also thanks to Peg Gray for typing, and special thanks to Shirley Rasmussen, without whose PDP-10 expertise this report would never have been produced on time.

A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture

1. INTRODUCTION

Our interest in data flow schemas stems from a desire to develop improved architectures for computing systems. It is our feeling that necessary improvements in computer systems behavior can come about only by some rather new and radical approaches to the basic design of machines [1]. The use of a data flow language as the computer's machine language is one such approach. Some advantages of data flow over conventional approaches are:

(1) more functional behavior, and hence more modularity in programming. This is achieved through the elimination of the concept of a variable.

(2) more asynchronous data-driven control of programs; as opposed to conventional machines, statements are ordered only by the data constraints of the algorithm.

(3) more operations on structured data are provided at the elementary level of the language.

Other advantages are discussed in [9].

This report discusses a new interpreter for data flow schemas. For those familiar with data flow languages such as Dennis' [8,12], the following is a brief description of the functioning of the new interpreter. (For those not familiar with data flow languages, we have included an introduction in Section

2.) The interpreter of Dennis' data flow language specifies an output queue length of one. That is, a maximum of one token may be present on any arc at any one time. This implies the need for a feedback connection from receiver to sender so that the receiver of the token can inform the sender when the token has been removed from the arc, thus allowing the sender to produce further output. As is pointed out later, the maximum queue length of one may limit asynchrony. The new interpreter, on the other hand, imposes no limit on the number of tokens on an arc. Furthermore, under the new interpreter it is possible for (most) statements to be executed out of order. For example, if all input queues to a function statement s have tokens present in queue position i, then s may execute with the data from position i regardless of the presence or absence of tokens in queue positions i-1, i-2, ..., or i+1, i+2, ... We show in Section 4 that the results produced are the same as if all statements were executed in sequence. Thus the new interpreter increases the degree of asynchrony in the system over other data flow systems and removes the need for a feedback communication system.

The new interpreter has significant architectural implications. It requires run-time binding of processors to each instance of each statement's execution. Also required is a flexible communication system which can isolate procedure invocations from one another. Further remarks on architectural requirements appear in Section 5.

This report is organized in the following fashion: Section 2 reviews schemas in general, and Section 3 describes the new data flow interpreter within the context of the particular data flow language of Dennis. Section 4 proves that the new interpreter produces the same computation as the standard interpreter, and Section 5 discusses some architectural implications of this new interpreter.

2. COMPARATIVE SCHEMATOLOGY

2.1 Background

Many schemas have been devised to represent asynchronous computation. Roughly speaking, the models fall into two general classes: those with a random access memory and data variables [2-4], and those without data variables [5-10]. Figure 1 gives an indication of how programs in the two classes might compute the same function. The rules of operation are similar for both: Figure 1a (with variables) shows computation units, initial data conditions, and the flow of control along arcs. The tokens shown on the arcs represent initial control conditions. A box may compute whenever all input arcs have a token present, and no output arc contains a token. Values are read from and written into the external memory during computation. Figure 1b (a schema without variables, or data flow) shows the computation units connected by arcs. The arcs hold tokens, and the tokens contain the data values (thus the initial input value for f is 1, and for g it is 2). A box in the data flow schema may compute whenever

all input arcs contain a token (data is present), and no output arc contains a token (old output data has been absorbed).

Note that synchronization of accesses to data is not necessary (i.e., races for data values are impossible) in data flow since there are no shared variables. All "control flow" is incorporated in the flow of the data itself, and the behavior is intrinsically functional.

Further arguments in favor of data flow are mentioned in [9].

2.2 Dennis' Data Flow Language

Of the several data flow languages, that of Dennis [12] has been selected as the basis for further work. There are two primary reasons for this selection. First the language has well defined semantics for data handling operations. (These are not discussed here, but may be found in [12] with a complete description of the language.) These same semantics also contribute to a significantly reduced data communications load, via the use of pointers to data structures. Second, even though this language may not yet be a mature programming language, it is more developed than any of the other data flow languages. It has the expressive power of programs written using assignment statements, while...do... constructions and if...then...else... constructions. Some important properties of this language have already been proven [8]. We expect many changes in this language. For example, such changes may involve the

incorporation of an arbiter [9] and a non-deterministic [14] operator. We believe that a machine design based on Dennis' data flow language will not be unnecessarily restrictive and will have many of the important features that data flow languages of the future will require.

For those unfamiliar with data flow languages, the remainder of this section gives a sample program and explains its operation. An example of a program in Dennis' Data Flow (DDF) language is shown in Figure 2: the calculation of a root of the function f by Newton-Raphson approximation (i.e., the successive calculation of $x_{i+1} = x_i - f(x_i)/f'(x_i)$.

Two initializing value tokens are required to begin the computation. A token with value zero must be placed on the arc inbranching to the False side of statement S1, and an initial root approximation value token must be similarly placed on statement S8. In addition, two initial control tokens (true and false values only) are shown as inputs to the control side of S1 and S8. Given this configuration, computation may begin. Either S1 or S8 or both may execute. Each is a merge statement, and by definition of a merge, a false control token selects for its input a value token from the False side. The merge then simply copies the input value onto a new value token and the token is output from the merge. The input value token and the control token are destroyed. In the case of S1, only one output token is produced, but S8 produces four output tokens. Each of these four output tokens carries the same value (copied from the input value

token), but each goes to a different destination. One token goes to S9 to compute the function f, one token goes to S10 to compute the derivative of f, etc. Both S9 and S10 may execute with the arrival of the output value tokens from S8. Statement S12 must wait for more input before execution can occur.

The basic rule of operation is

> a Data Flow statement may execute anytime after the arrival of the input tokens required for its execution, provided no tokens are present on its output lines; those tokens required on input are destroyed, the calculation is made, and output tokens (if any) are produced in finite but non-zero time.

Thus all precedence among statements is established by explicit flow of data in the form of values written on tokens. Asynchronous execution is specified by an absence of explicit precedence, and is therefore the default behavior.

After the function and its derivative have been calculated, the division is made; the result is the total increment in the value of x. Statements S14 and S15 test the increment against ϵ - the allowed error. If the incremental change in value is less than ϵ, then the predicate box S15 outputs a <u>false</u> control token to indicate that sufficient accuracy has been attained. Statement S5 will then output a <u>false</u> token to S16 (regardless of

the truth value of the S3 result). Statement S16 is a gate-if-False statement which produces an output value token whenever an input false control token and an input value token arrive; the output value is the same as the input value. It is essentially an output gate. If, however, a true control token arrives at a gate-if-False statement, no output token is produced at all, but the inputs are still destroyed. Thus if the error is not acceptable to S15, a true token arrives at S14 and no answer is produced.

At this point we come to the statements which set up the activity for a loop. We also encounter the addition we have made to the DDF language -- the D-box ("D" for delay). A D-box is an identity function and simply passes the input value to the output. We require a D-box be present at those points, and only those points, where there is an initial input token in DDF. The initial token is then placed on the output arc of the D-box before execution. (In DDF the D-box essentially performs no operation but is necessary when discussing the new interpreter.) So, assume S3 and S15 have both produced a true token (S3 will be discussed momentarily). Then S5 will produce five true control tokens as output, two of which pass through D-boxes to S1 and to S8. S8 will merge the new approximation, gated from S13, back into the computation loop. This looping will continue until either the error is acceptable (S15), or until we have looped more than n times (statements S1-S6). Statements S1-S6 are counting, in parallel with the main computation, the number of times a new root has been tried. If no acceptable solution is

found by at least the n^{th} iteration, then statements S1-S6 force the approximation so far computed to be output. This will also halt the main loop S7-S15.

3. TWO INTERPRETERS FOR DATA FLOW SCHEMAS

In this section we will describe two hypothetical computers called FI(feedback interpreter)and NFI(non-feedback interpreter) on which a given data flow program could be executed. We assume that the programs specify an initial distribution of control tokens, and that the programs are well-behaved [8,12]. Well-behaved programs are defined to be the programs that

(1) start execution by accepting one input token per input arc,

(2) terminate execution and output one and only one token per output arc after a finite but non-zero time, and

(3) restore the initial distribution and value of tokens and leave no other tokens behind in the program.

The concept of a well-behaved DDF program is the same as that of properly terminating nets [4, 11].

3.1 Feedback Interpreter (FI)

The machine FI consists of an ensemble of a large number of Processing Elements (PEs), where each PE is capable of executing any DDF statement. In order to execute a program on FI, we first assign and fix one PE to each statement of the program and then interconnect these PEs (either logically or physically), according to the program specifications. Thus, suppose statement s of procedure P belongs to some loop within P and that s is

executing on a PE for the i^{th} time in the current invocation of P. Then we may speak of the activity "P.s" occurring for the i^{th} time in the current invocation of P.

As previously stated in the definition of DDF (similar definitions are used by Kosinski and some investigators of Petri nets), a PE can fire or begin execution whenever it has received all its operands, except that no token may be output onto an arc which already holds a token. Therefore in FI, to ensure that no more than one token is output onto a data path at any one time, we need explicit acknowledgement of the last token received on that particular data path. That is, all the communication between PEs must be of send/acknowledge (feedback) type as, for example, in [15]. Hence the name feedback interpreter.

3.2 Non-Feedback Interpreter (NFI)

It is possible, however, to relax the condition that an operator can fire only when the output arc is empty. This can be done without destroying the determinacy properties of the system. According to Patil [13], if we can ensure that no token on any arc is lost and that a strict first-in first-out condition is maintained on each arc, then the system will still be determinate. Unfortunately, no finite amount of memory alone on the data paths can guarantee these conditions.

3.2.1 General Operation of NFI

The second machine that we define, called NFI, avoids the problem of send/acknowledge communication. NFI also allows us to speed-up computation by increasing the apparent asynchrony and allowing more activities to be created and assigned to PEs than in FI. The PEs in NFI are the same PEs as in FI, but in NFI the PEs must communicate with each other by dynamically created names. Thus in NFI, each PE in use will be assigned an activity name. Since activity names are (can only be) created at execution time, it is impossible to assign PEs in NFI in a static manner. Thus all tokens must carry the activity name of the destination PE as well as the datum.

Before we describe the procedure for creating activity names in NFI, note that a search for the PE with the proper name must be carried out for each token. If no PE is found, then a new PE must be assigned to carry out the activity designated by the activity name on the token. Thus each PE of NFI repeatedly goes through basically the following cycle: (deadlock is not discussed here, but is easily avoided).

(1) If a PE is free, then it is assigned an activity name A by the control of NFI. (This will happen when at least one token with activity name A has been produced by some other PE as its output.)

(2) The PE which has been assigned activity name A, waits for its operands to become available.

(3) When all the operands have been received, the PE starts execution of the desired activity and after a finite but non-zero amount of time it terminates execution. Result tokens are then produced.

(4) The PE becomes free.

The scheme described below for creating activity names removes the need for send/acknowledge, or feedback, communication; hence the name non-feedback interpreter. It is imperative to understand the activity name generation procedure in order to understand NFI.

3.2.2 Generating Activity Names for NFI

Suppose statement s of procedure P belongs to some loop within P, and that s is executing for the i^{th} time in the current invocation of P. Further assume that the context from which procedure P has been called is represented by symbol u. Then the activity name of this execution of statement s is given by u.P.s.i. For example, let u represent the context from which the procedure sum (Figure 3) has been called. Assume input data has been presented to sum which will cause it to loop three times. Figure 4 shows all activities which will be created during the course of this execution. The i^{th} position in the vertical dimension for a statement in Figure 4 represents the i^{th} initiation of that statement as an activity. All statements at the i^{th} level will have "u.sum.__.i" in common in their activity names. The name generation process must determine the appropriate iteration (or initiation) count i for each token produced. This will be further discussed later.

Consider now the three activities that perform the "apply P" operation in Figure 4. Although not detailed in Figure 4, each of these activities invokes procedure P, and thus gives rise to an entire 3-dimensional execution structure of P in the

"neighborhood" of each apply P box. Since several invocations of P might be active concurrently (this is not possible in FI), the name-context from which P is called must be passed on to each invocation unambiguously. Therefore, if statement t of procedure P is initiated for the j^{th} time during the i^{th} invocation of statement s in procedure sum, then the activity name of t must be u'.P.t.j where u' = u.sum.s.i (s refers to the statement "apply P").

We now give precise rules for the creation of activity names in NFI for each type of statement.

3.2.2.1 Functions, predicates: These m-input n-output statements have the property that their input and output tokens all have identical initiation counts for any given execution. Such function statements are represented by a box as in Figure 5a, while predicates are represented by a diamond as in Figure 5b.

3.2.2.2 Gates: A gate has the property that it only occasionally sends output. Suppose a gate statement G is to be initiated for the i^{th} time, and so far the value token was allowed to pass through the gate only j times. Let the output of the gate G be connected to statement a. Now if only j tokens have passed through the gate, then statement a must have been initiated exactly j times. Therefore when the gate subsequently sends its output token to a, the activity number must be u.P.a.j+1. The value of j cannot be derived from u.P.G.i alone. Therefore, for proper name generation, a gate must keep track of the number of times it produces output. How this is done is described in the

following paragraph.

Every gate waits for a dummy token as well as the usual value and control tokens as shown in Figure 5c. The dummy token carries the value j which is the number of tokens so far passed through G on previous executions. When a gate executes, the gate sends a dummy token to u.P.G.i+1 with value j+1 or j depending upon whether the value token was allowed to pass through the gate statement (j+1) or not pass (j). Figure 5c shows the rule for a gate-if-True statement; a corresponding rule exists for a gate-if-False statement.

3.2.2.3 D-box: A D-box has the property that if the input token is the i^{th} token received, then the output token (whose value is a copy of the input token value) will be the $i+1^{th}$ token sent to the destination of the D-box. This is because a D-box is present if and only if an initial token was specified, and the D-box may be said to have produced its first output token for no tokens input. The first token input is thus the second token produced, etc. This is shown in Figure 5d.

3.2.2.4 Merge: A merge (essentially the inverse of a gate) produces an output token on each initiation, but only requires tokens from two of the three inputs at each initiation. For example, a true control token and a value token on the True input side is sufficient to initiate and terminate one merge operation. Suppose a merge M has been initiated i times; then i tokens must have been received at its control input. Furthermore, suppose j tokens on the True input and k tokens on the False input have

been absorbed in the past i initiations. Then i=j+k, and subsequently arriving tokens on the control, True, and False inputs would bear the activity names u.P.M.i+1, u.P.M.j+1, and u.P.M.k+1, respectively. In general, the activity names of tokens input to the same execution of a merge operation will not be the same, and thus we have a problem of coordinating the inputs. Tokens with different activity names cannot go to the same PE.

Our solution to this problem is to break each merge statement M into three parts called M_C, M_T, and M_F, and let each part be executed by a separate PE; thus each input is received by each part separately. Since the M_C, M_T, and M_F parts of a merge may be in different planes, knowledge of the current initiation number of each part is needed for proper communication. The complete operating description of each part is given in Figure 5e.

As can be seen from Figure 5e, M_C waits for a control token and for a dummy token with value [j,k] at the i=j+k-1 initiation. Depending upon the value of the control token, it either sends a dummy token with value i to u.P.M_T.j or to u.P.M_F.k. It also sends a dummy token with value [j+1,k] or [j,k+1] to the control box M_C of the next initiation of the merge. The activity name of the next initiation of M_C is u.P.M_C.i+1. Both M_F and M_T wait for the arrival of a value token and a dummy token. One of M_T and M_F receives both input tokens and fires. Then it sends a token to the next statement with activity name u.P.a.k, where k was

communicated through the value portion of the dummy token sent from M_C to M_T or M_F.

3.2.2.5 Apply: Apply statement A needs two types of arguments to execute: a procedure Q and a list of arguments for the procedure. Since Q can be a procedure already active (e.g., consider the case when the procedure containing the apply statement A is procedure Q - a recursive call) statement A must cause the activity names used in this invocation of Q to be different from the activity names currently in use. If we assume the activity name associated with the PE that executes statement A is u.P.A.i, then the unique activity names for statements in the called procedure Q can be created by assigning names of the type u'.Q._._ where u'=u.P.A.i. Since u' is unique, u'.Q._._ is guaranteed to be unique.

A reverse process must take place when procedure Q activated by A is terminated. The last statement to execute in Q must send the result tokens to u.P.A.i. However, there is one problem. According to the rules for PE allocation, once a PE starts execution it must terminate after a finite but non-zero time. Hence, (as in [12]) we break each apply statement A into two parts called A_A (for activate) and A_T (for terminate) as shown in Figure 5f. Let us assume that the first and last statements of each procedure are begin and end, respectively. These statements have some special properties which are described in the following paragraph.

Complete specification of a program must include the information regarding the distribution and value of initial tokens in the procedure. The begin statement contains precisely this information. The activate portion of the apply statement A sends all input tokens required by the procedure Q to the begin statement of Q (Figure 5g). Statement begin in turn sends all initial, input, and dummy tokens to the various statements within the procedure Q (Figure 5h). If procedure Q terminates (note that Q is well behaved) one and only one result token is produced on each output arc, and these in turn are sent to the end statement of Q (Figure 5i). The special statement end first determines the context from which Q was called (context u′) and then passes all result tokens to the terminate portion of the apply (which exists in environment u) with activity name $u.P.A_T.i$ (Figure 5j). A_T simply copies the u.P.-.i part of activity names and outputs these tokens (Figure 5k).

It is obvious that all activity names with prefix u.P (or prefix u′.Q) are related to each other in some manner. We will call the set of PEs with common activity name prefix u.P the domain of procedure P when called from context u. When a procedure terminates it may leave behind some initial and dummy tokens in its domain. Therefore one task of A_T is to free all the PEs in the domain of a procedure after the result tokens have been produced. The signal to free these PEs is produced by A_T as shown in Figure 5k. We have also shown throughout Figure 5 a token being sent from A_A to A_T. This is done in anticipation of a later broadening of the scope of the apply statement. If we

wish to incorporate the notion of nondeterministic machines, then several procedures may be activated simultaneously and as soon as the first acceptable result is produced, all domains created by the apply will be destroyed. Domains are discussed further in Section 5.

4. THE FUNCTIONAL EQUIVALENCE OF DDF PROGRAMS UNDER THE FEEDBACK AND NON-FEEDBACK INTERPRETERS

In this section we give an informal but rigorous proof that the feedback interpreter (FI) and the non-feedback interpreter (NFI) both produce the same results for a DDF procedure, given the same arguments. It should be noted that all DDF statements are functional, i.e., they have no writeable local memory.

First, we give some definitions. An arc a connects one output of a statement s to one input of a statement t. Arc a is then an output arc of s and an input arc of t. In the following, let a be an output arc of statement s in procedure P. If P is called in context u, then vector A associated with arc a (Figure 6a) is defined as follows:

Feedback interpreter (FI): the list of all tokens appearing in sequence on arc a such that A(i) is the i^{th} token. Note that the i^{th} token on an arc under FI can appear only after the $i-1^{th}$ token has appeared.

Non-feedback interpreter (NFI): the list of all tokens such that the token with activity number u.P.s.i is element A(i). Figure 4 shows that activity u.P.s.i can occur before

activity u.P.s.i-1 occurs.

The input (output) matrix (Figure 6b) of a statement s in P is an ordered set of all vectors A associated with arcs input to (output from) statement s. Similarly, the input (output) matrix of a procedure P is an ordered set of all vectors A associated with arcs input to (output from) procedure P. Two procedures P_1 and P_2 (Figure 7a) are interconnected* (Figure 7b) by connecting j output arcs of P_1 to j input arcs of P_2, and connecting k output arcs of P_2 to k input arcs of P_1. Lastly, let P_{EQ} be the class of all procedures in DDF such that if P is a member of P_{EQ}, then for any input matrix, P under FI produces the same output matrix as P under NFI.

The following lemma shows that all individual statements in DDF produce the same results under FI or under NFI, thus proving that P_{EQ} is nonempty.

Lemma 1: Given identical input matrices, statements in DDF under FI and under NFI produce identical output matrices.

Proof -

I. To prove this lemma, we first observe the behavior of each statement type under FI and define the relationships which exist between input matrix positions and output matrix positions.

---

*This general method of interconnecting two asynchronous systems is from Patil [13].

To begin, let $P_A(i)$ be a partial function on the input vector A associated with input arc a of statement t. This function $P_A(i)$ gives the position of the output token in the output matrix of statement t corresponding to the input token in position i of input vector A. For each statement type, the definition of $P_A(i)$ is given below.

Type 1 - function, predicate, apply statements

In particular, for a function $(y_i,z_i)=f(v_i,w_i,x_i)$ as shown in Figure 6b. Thus $P_I(i)=i$ for all i, and for all vectors I input to the function.

That is, position i of vector I input to a function statement determines the output vector only in position i. This holds for any input to any function statement. Similarly, the relationship $P_I(i)=i$ holds for any input to any statement of types predicate and apply as well.

Type 2 - D-box statement

$P_I(i)=i+1$

By definition, a D-box is present if and only if there is an initial token, i.e., the first token is produced by a D-box for zero tokens input. The first token input will be the second token output, etc. The D-box is present only for the purpose of incrementing initiation numbers of tokens.

For Type 3 (merge) and Type 4 (gate) statements, we need to define a function which counts the number of true and false control tokens arriving on a control input. Let Ftrue(i) be a function giving the position in the control input vector corresponding to the i^{th} true control token appearing in sequence on the control input line, starting at position 1. Similarly, let Ffalse(i) be the input position corresponding to the i^{th} false control token. Both Ftrue(i) and Ffalse(i) may be seen as counters looking for the i^{th} true or false tokens, respectively, in the control input vector. Given a control input vector, functions Ftrue(i) and Ffalse(i) are uniquely defined.

Type 3 - merge statement

$P_C(i)=i$

$P_T(i)=$Ftrue(i)

$P_F(i)=$Ffalse(i)

Type 4 - gate statement

(a) gate-if-True: There are two input vectors C and V (V for value). Not every input produces an output, so the input-output relationships are

$$P_C(i)=P_V(i)=\begin{cases} j \text{ if } i = \text{Ftrue}(j) \\ \text{undefined otherwise} \end{cases}$$

(b) gate-if False:

$$P_C(i)=P_V(i)=\begin{cases} j \text{ if } i = \text{Ffalse}(j) \\ \text{undefined otherwise} \end{cases}$$

These equations give the relationships among the input and output matrix positions under FI for all statements in DDF. It remains to show that these, and only these relationships are preserved under NFI.

Case 1 - *function, predicate, apply* statements

By observation of Figures 5a, 5b, and the first and last snapshots of *apply* in Figures 5e and 5j, the required relationship $P_I(i)=i$ is preserved.

Case 2 - *D-box*

By observation of Figure 5c, $P_I(i)=i+1$ holds under NFI.

Case 3 - *Merge*

There is only one output token produced at each termination of a *merge* statement, and its position in the output vector is determined by the position of the control token in the control input vector. To show that the proper relationship between the input matrix and output vector of a merge under NFI is preserved, we make the following assertions by examining the rules given in Figure 5k.

(a) The control part of the merge (M_C) is initiated the k^{th} time only after the $k-1^{st}$ initiation of M_C has terminated.

(b) The value of the dummy token received by M_C at the k^{th} initiation is $\langle i,j \rangle$ (where $k=i+j-1$) if $i-1$ true control tokens and $j-1$ false control tokens have so far been received.

(c) If the k^{th} control token is true then a dummy token with value k is sent to i^{th} initiation of M_T. This is precisely Ftrue(i) which is $P_T(i)$. Similarly in the case of a false token, a dummy token with value k is sent to the j^{th} initiation of M_F, which is precisely Ffalse(j)=$P_F(j)$.

(d) Either M_T or M_F will output a token which will go to output vector position k, that is, $P_C(k)=k$.

Case 4 - Gate-if-True (Similarly for gate-if-False)

Since a gate does not output a token on every termination, the input-output relation for a gate is partial. We make the following assertions about the execution of a gate under NFI (please see Figure 5d).

(a) The k^{th} initiation of a gate occurs only after the $k-1^{st}$ initiation has terminated.

(b) The value of the dummy token received by the gate at the k^{th} initiation is i if $i-1$ true control tokens have previously been received.

(c) If and only if the k^{th} control token received is true (i.e., Ftrue (i)=k) then an output token will be produced and it will go into the i^{th} position of the output vector. This is precisely the definition of $P_C(k)$ and $P_V(k)$.

II. Thus, the proper relationship between input and output

positions for any statement in DDF holds. Since under both FI and NFI the input matrices are the same, and the statements calculate the same function, then the output matrices must be the same.

QED.

Corollary 1: All single statement procedures are in P_{EQ}.

Proof - Immediate by Lemma 1.

Theorem 1: Let P be formed by interconnecting procedures P_1 and P_2. If P_1 and P_2 are members of P_{EQ} then P is also in P_{EQ}.

Proof - The proof to follow uses two program pieces P_1 and P_2, as shown in Figure 7a. These two pieces are connected together in the fashion shown in Figure 7b to form a new system. Those inputs to P_1 which are connected to outputs of P_2 are called V; those inputs to P_2 which are connected to outputs of P_1 are called U. The other inputs and outputs of P_1 and P_2 remain unconnected. The inputs to the new system are X, and the outputs are Y. The proof shows that the operation of program composition is closed over P_{EQ}.

I. (=>) If X is an input matrix to P producing output matrix Y under FI, then P will also produce Y under NFI.

(a) Let P produce matrices U, V, Y under FI. Assume P under NFI produces output matrix Y', where Y' is different from Y. Then there is an error in Y_1, or Y_2, or both.

(b) There can be an error in Y_1 if and only if there was an error

in the input matrix of P_1. Similarly, an error in Y_2 can be due only to an error in the input matrix of P_2. Since X_1 cannot be in error, there must exist an error in matrix V produced under NFI.

(c) Let NFI make the first error in time in U, or V, or both, and let this error be in position k_U, or k_V, or both, respectively. (All matrix position denotations are pairs; that is, k_U and k_V are each a pair specifying a single entry in matrix U and V, respectively.)

(d) 1. Assume entry $U(k_U)$ is wrong. Entry $U(k_U)$ depends upon some entries in the input matrix to P_1. Since P_1 is in P_{EQ}, $U(k_U)$ can be wrong if and only if some corresponding entry in the input matrix was wrong. X cannot be wrong, so the error must be in some entry V(i). $U(k_U)$ could not be produced under NFI until V(i) has been produced. This is because every statement requires non-zero time to produce output tokens after all the required input tokens have been received. But V(i) cannot be wrong because $U(k_U)$ was hypothesized to be the first such error. Thus $U(k_U)$ cannot be incorrect. Contradiction.

2. Assume $V(k_V)$ is wrong. By the same argument as above, $V(k_V)$ cannot be wrong. Contradiction

3. If both $U(k_U)$ and $V(k_V)$ are wrong, then again by the same arguments as above there must be entries U(i) and V(j) which are wrong and which have occurred before $V(k_V)$ and $U(k_V)$, respectively. Again, a contradiction.

(e) Thus, there can be no Y′ different from Y.

II. (<=) If X is an input matrix to P producing output matrix Y under NFI, then P will also produce Y under FI. The proof is the same as above, and can be stated simply by interchanging NFI and FI in Part I.

QED.

5. IMPLICATIONS FOR MACHINE ARCHITECTURE

A data flow program is a collection of statements or activities which initiate when inputs have arrived, then execute, and produce outputs upon termination. We envision a machine composed of a large number of small physical processing elements of the type described in NFI. These PEs are interconnected by a communication system which transports tokens output by one activity to the input of another activity. We note in passing that such PEs may be built with current LSI technology.

With this view in mind, we note the following points:

(1) There is no method to determine a priori the total number of activities which may be generated by any given invocation of a procedure. Since activities are created dynamically, automatic allocation and deallocation of PEs to each activity is required.

(2) Due to dynamic activity creation, there can be no fixed connections among particular PEs; all token traffic is by logical (activity name) destination addressing.

(3) A machine architecture that caters to the dynamic allocation/deallocation of PEs will have inherent modularity, i.e., it should be possible to easily increase or decrease the total number of physical PEs at any time.

(4) Activity name management is very important. Because of the large number of names which may be generated and the "length" of these names (due to procedure calls), it is necessary to partition the system by constructing firewalls which surround procedures in execution. These firewalls serve to hold the domain of a procedure's invocation. Since the domain has name u.P, only the s.i part of an activity name is carried by tokens within the firewalls. Also, when tokens with activity name A are output from a PE, the communication system must search to find the PE with activity name A. These same firewalls also provide bounds on the activity name search space.

(5) Automatic PE deadlock avoidance is necessary, and is accomplished by the implementation given in Figure 5: once a PE initiates execution, it is always able to complete execution and to be deallocated.

(6) No global control memory is necessary in order for the data flow interpreter to operate. All information required for the creation of activity names is carried by the tokens themselves. This allows a completely distributed control.

6. REFERENCES

[1] Glushkov, V. M., and M. B. Ignatyev, V. A. Myasnikov, V. A. Torgashev, "Recursive Machines and Computing Technology", IFIP Preprints, Stockholm, pp. 65-70 (1974).

[2] Karp, R. M. and R. E. Miller, "Parallel Program Schemata", J. of Computer and System Sciences, vol. 3, pp. 147-195 (1969).

[3] Slutz, D. R. The Flow-Graph Schemata Model of Parallel Computation, Ph. D. Dissertation, MAC-TR-53 (thesis), MIT, Sept. 1968.

[4] Gostelow, K. P. and V. G. Cerf, G. Estrin, S. Volansky, "Proper Termination of Flow-of-Control in Programs Involving Concurrent Processes", Proceedings of ACM National Conference, Boston, vol. 2, pp. 742-754 (Aug. 1972).

[5] Holt, A. W. and F. Commoner, "Events and Conditions", Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, Woods Hole, Mass., pp. 3-52 (June 1970).

[6] Keller, R. M., Vector Replacement Systems: A Formalism for Modeling Asynchronous Systems, TR 117, CS Laboratory, Department of Electrical Engineering, Princeton University (Dec. 1972, revised Jan. 1974).

[7] Rodriguez, J. E. A Graph Model for Parallel Computation, Ph.D. thesis, MIT, Department of Electrical Engineering, MAC-TR-64, Cambridge, Mass. (Sept. 1967).

[8] Dennis, J. B. and J. B. Fosseen, J. P. Linderman, "Data Flow Schemas", Symposium on Theoretical Programming, Novosibirsk, USSR, pp. 187-216 (Aug. 1972).

[9] Kosinski, P. R., "A Data Flow Language for Operating Systems Programming", Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices vol. 8, No. 9, pp. 89-94 (Sept. 1973).

[10] Bahrs, A., "Operation Patterns (An Extensible Model of an Extensible Language)", Symposium Theoretical Programming, Novosibirsk, USSR pp. 217-246, (Aug. 1972).

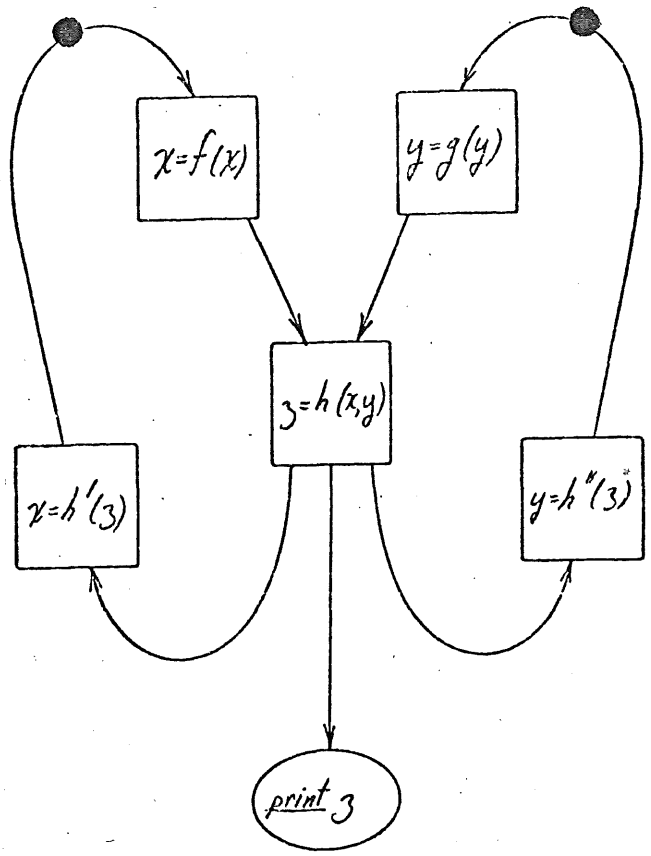
[11] Gostelow, K. P., "Computation Modules and Petri Nets", Proceedings of Third ACM-IEEE Milwaukee Symposium on Automatic Computation and Control, pp. 345-354 (April 1975).

[12] Dennis, J. B., "First Version of a Data Flow Procedure Language", Computation Structures Group Memo 93, Project MAC, MIT (Nov. 1973, revised Aug. 1974, May 1975).

[13] Patil, S. S., "Closure Properties of Interconnections of Determinate Systems", Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, Woods Hole, Mass., pp. 107-116 (June 1970).

[14] Chandra, A. K., "The Power of Parallelism and Nondeterminism in Programming", IFIP Preprints, Stockholm, pp. 461-465 (1974).

[15] Misunas, D. P., "Deadlock Avoidance in a Data-Flow Architecture", Proceedings of Third ACM-IEEE Milwaukee Symposium on Automatic Computation and Control, pp. 337-343 (April 1975).



cell	value
x	1
y	2
3	0
⋮	⋮

Figure 1a

Asynchronous computation with variables

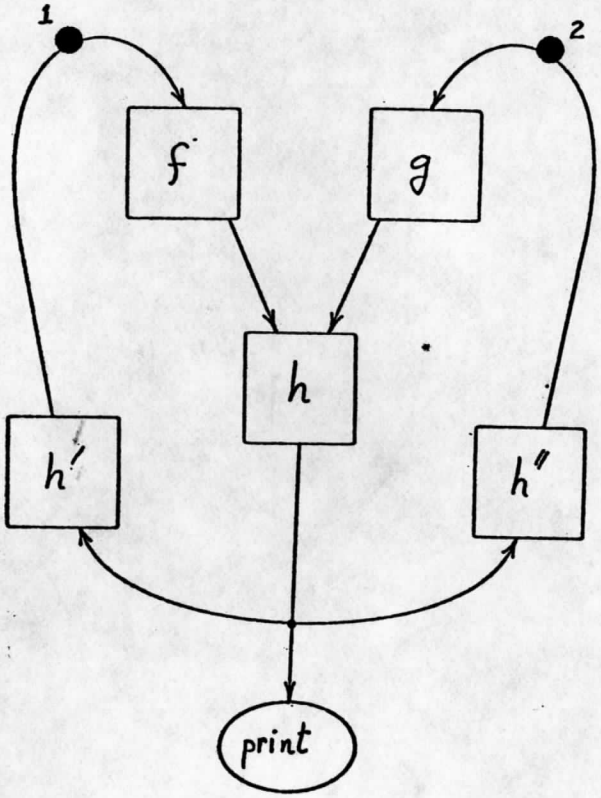


Figure 1b

Asynchronous computation without variables - data flow

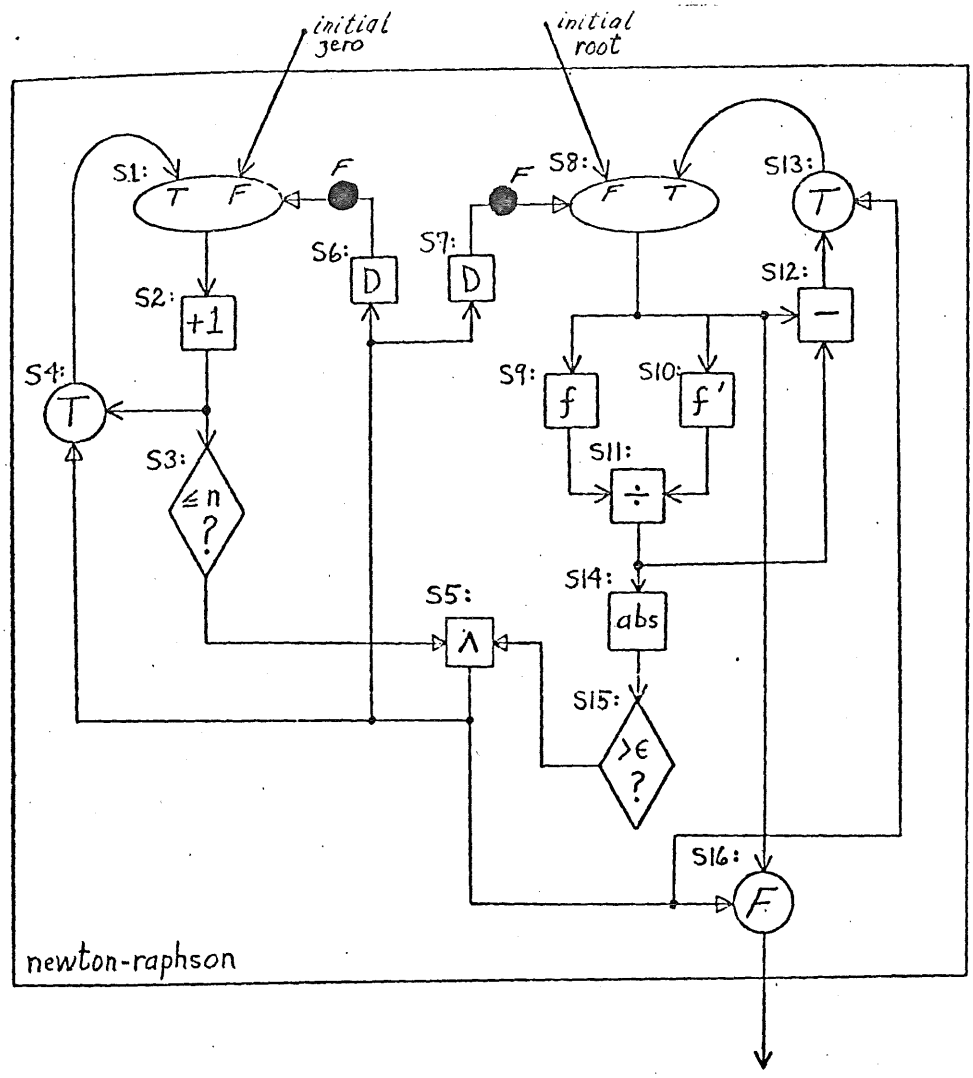


Figure 2

A program in Dennis' Data Flow language

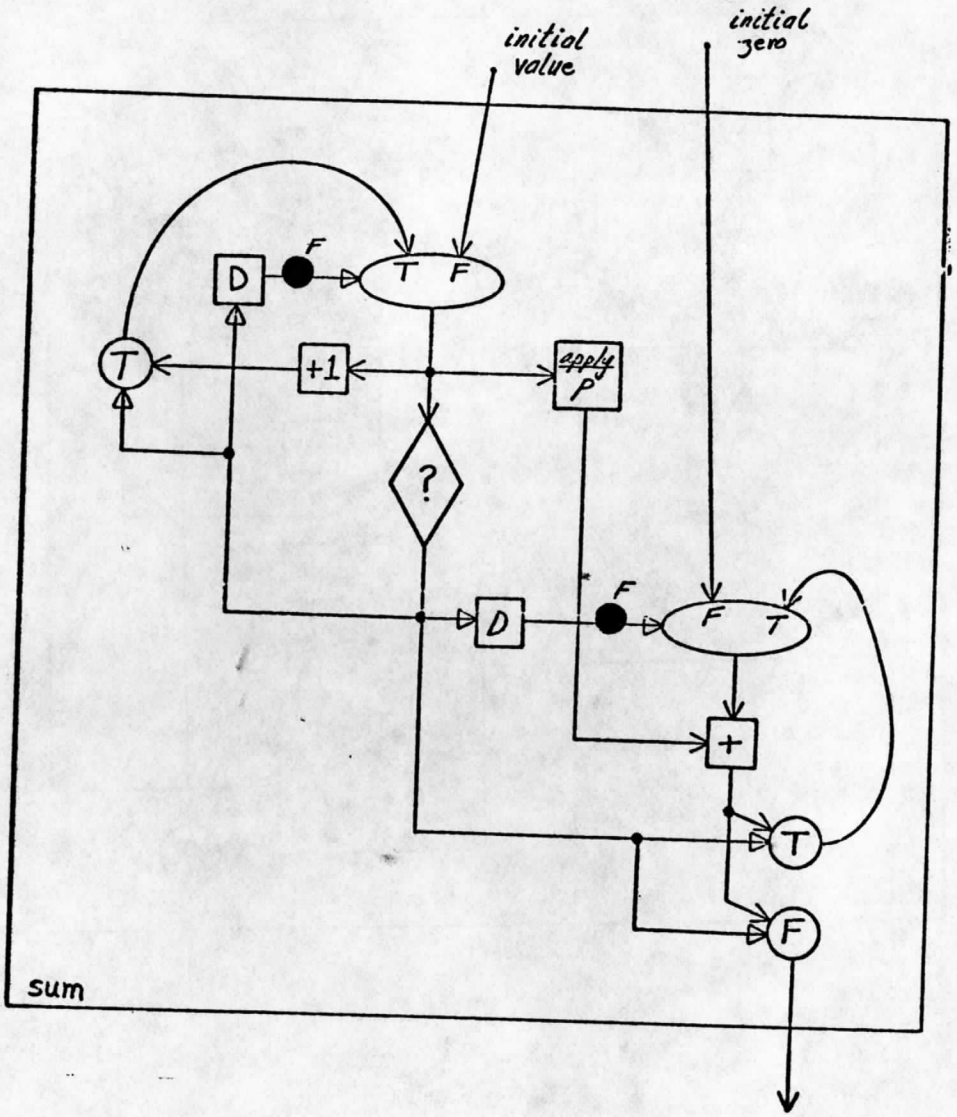


Figure 3

The data flow procedure sum

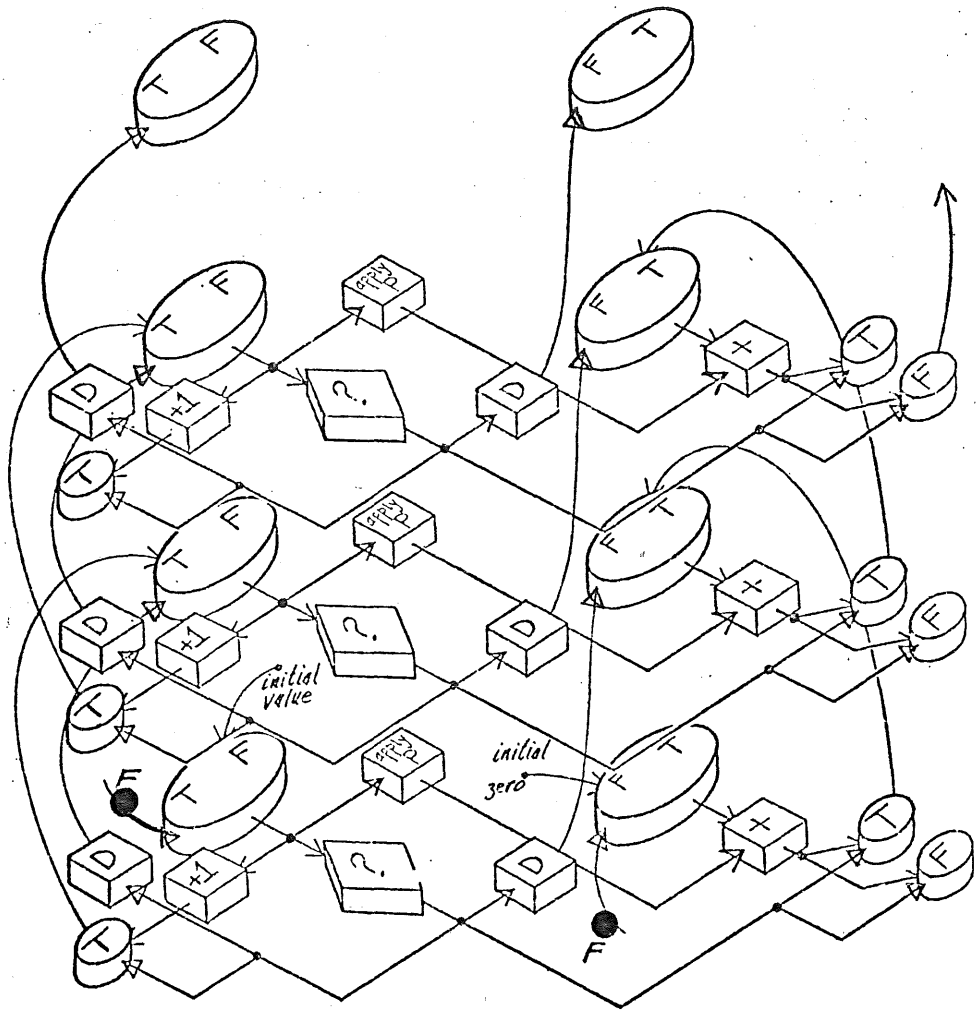


Figure 4

Three-dimensional activity space of procedure sum under particular input data

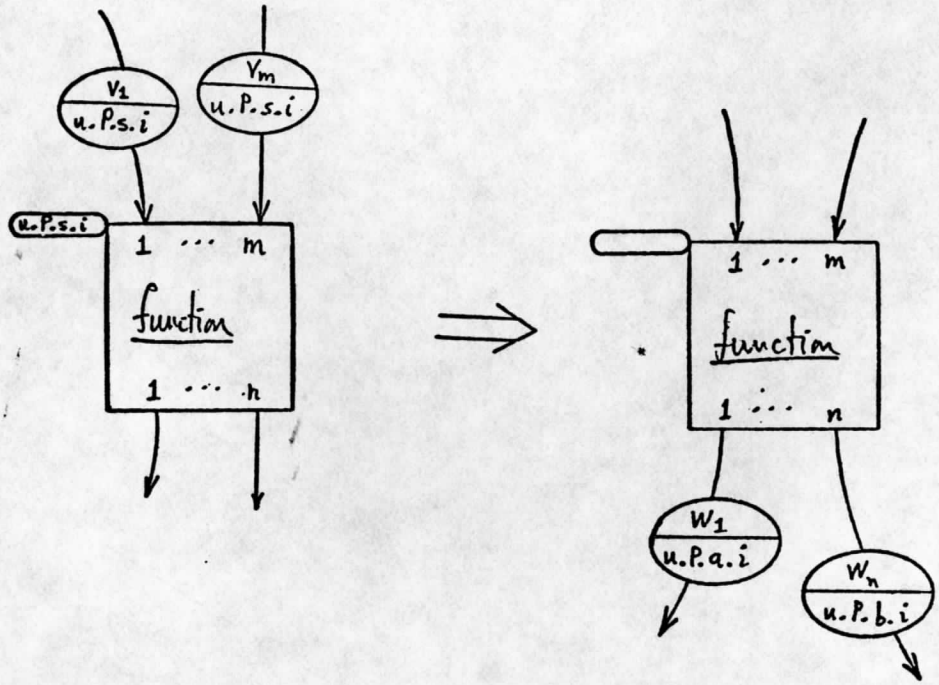


Figure 5a

Operation on activity names by function boxes

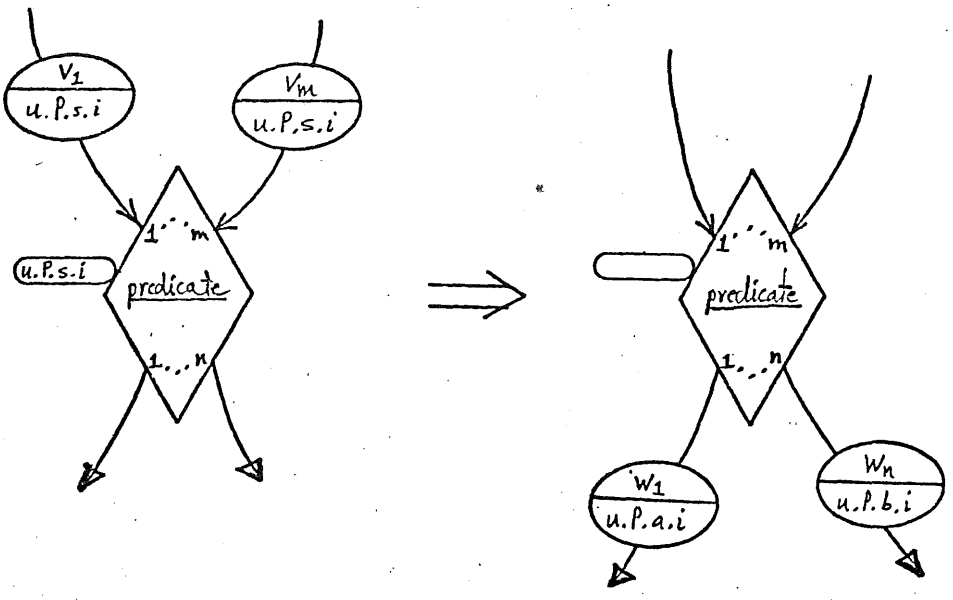


Figure 5b

Operation on activity names by predicate boxes

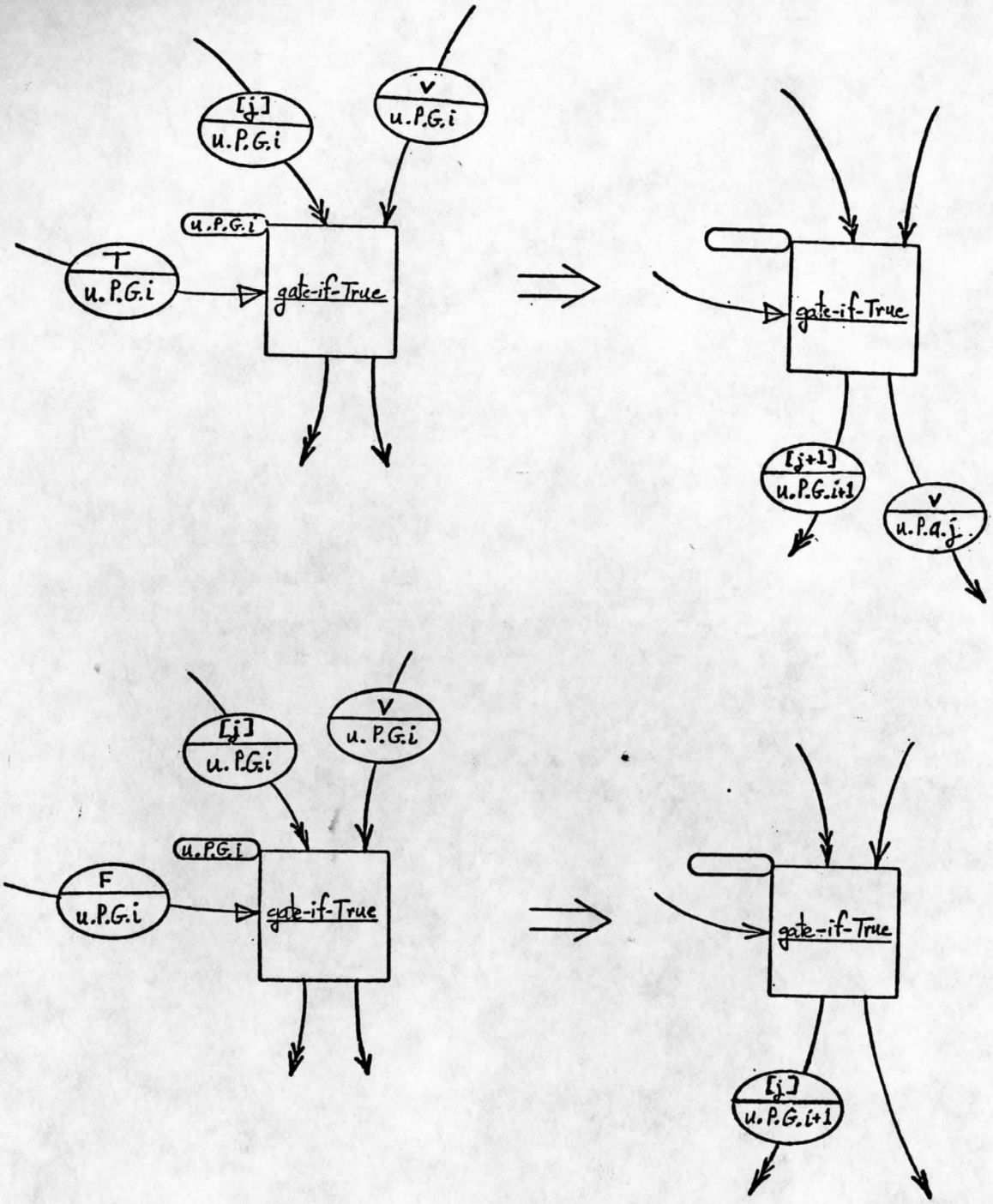


Figure 5c

Operation of a gate-if-True box on activity names
under both true and false input tokens

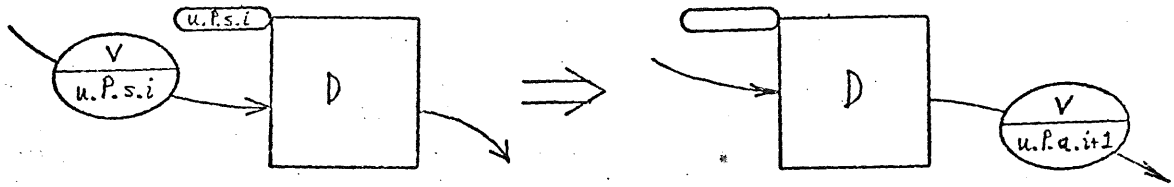


Figure 5d

Operation on activity names by a D-box

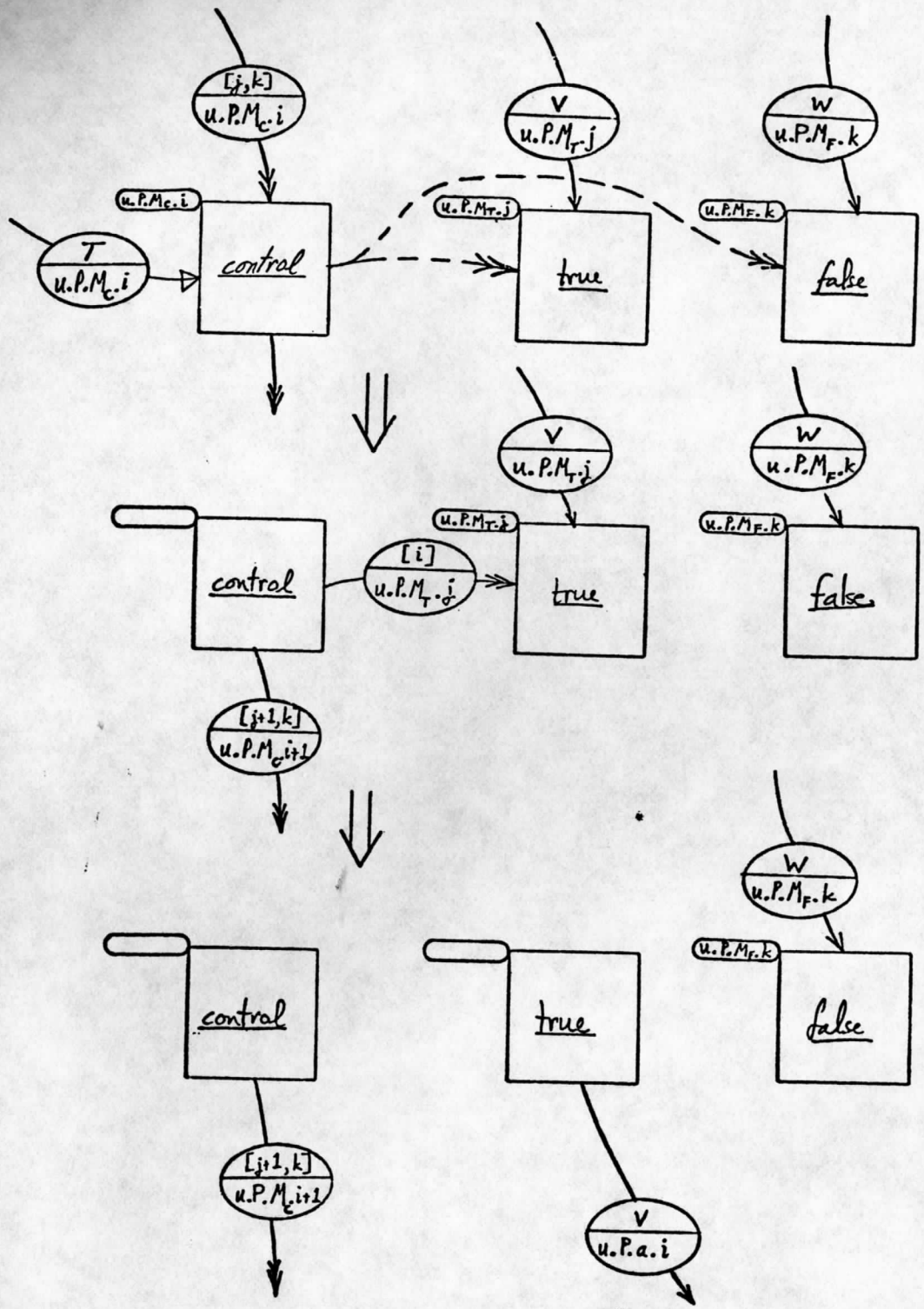


Figure 5e

Operation of a merge statement on activity names with a true input token

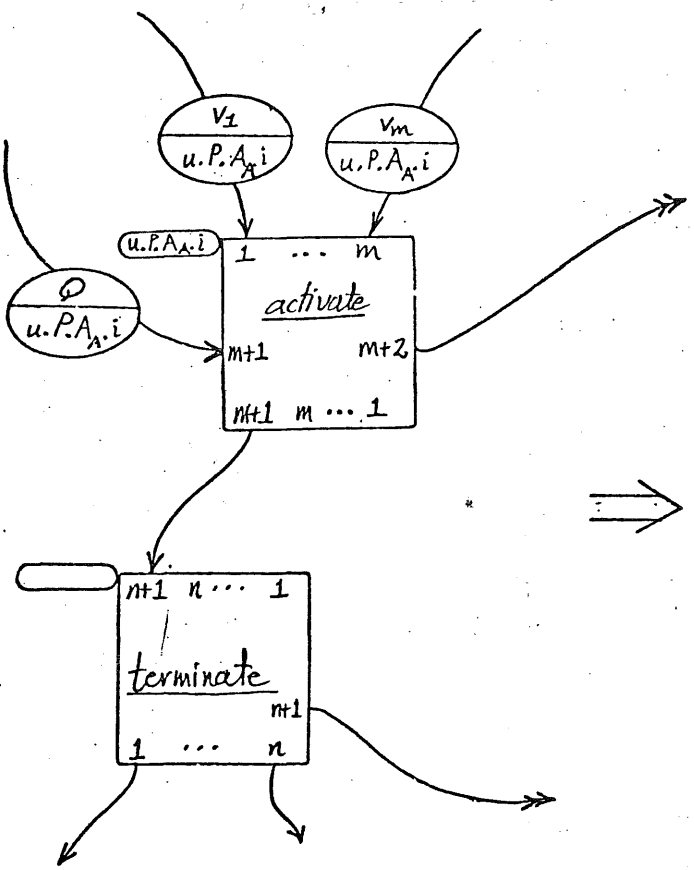


Figure 5f

The activate and terminate portions of apply

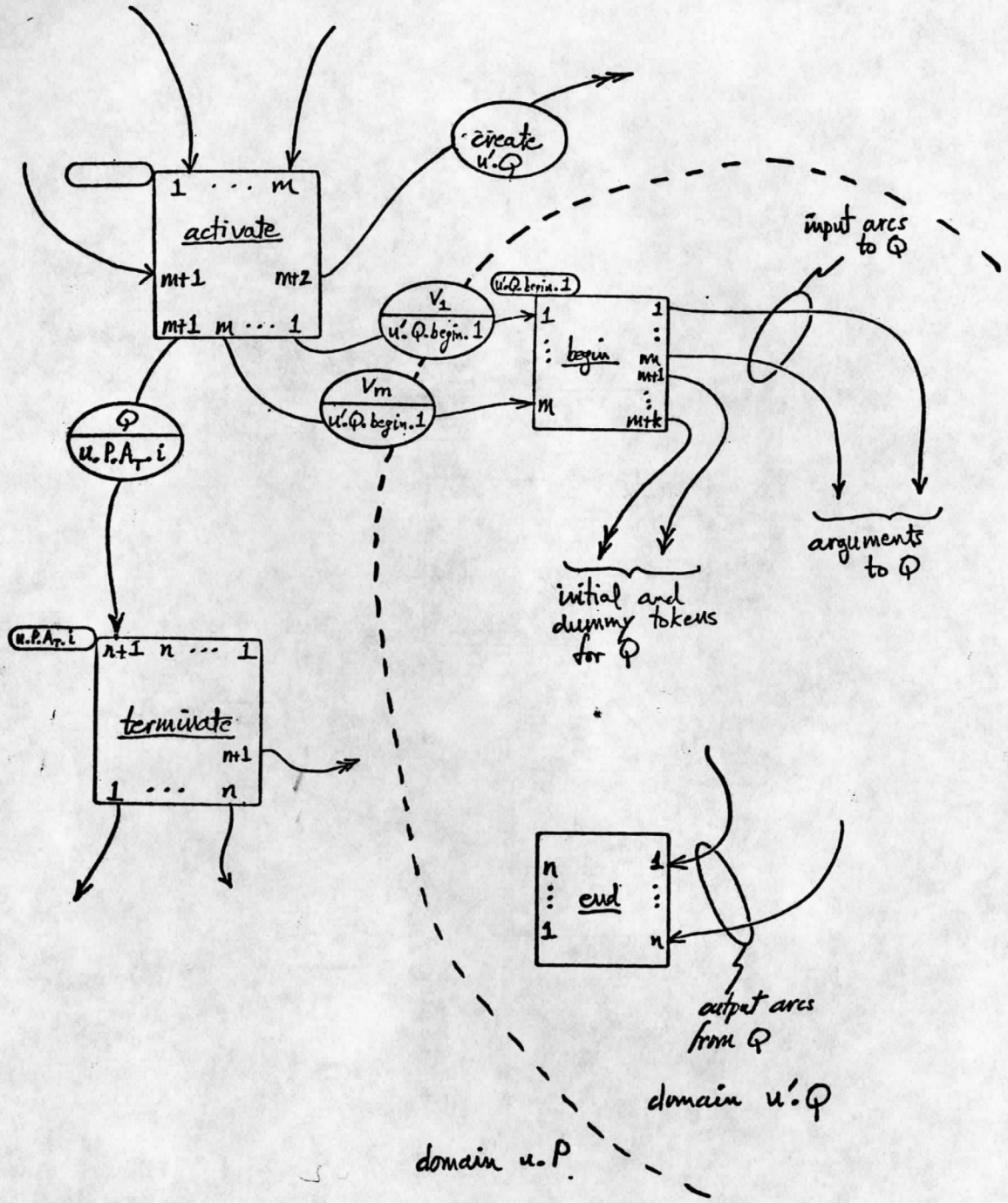


Figure 5g

Activate causes the creation of a domain for the called procedure, and begin is the first statement of that called procedure

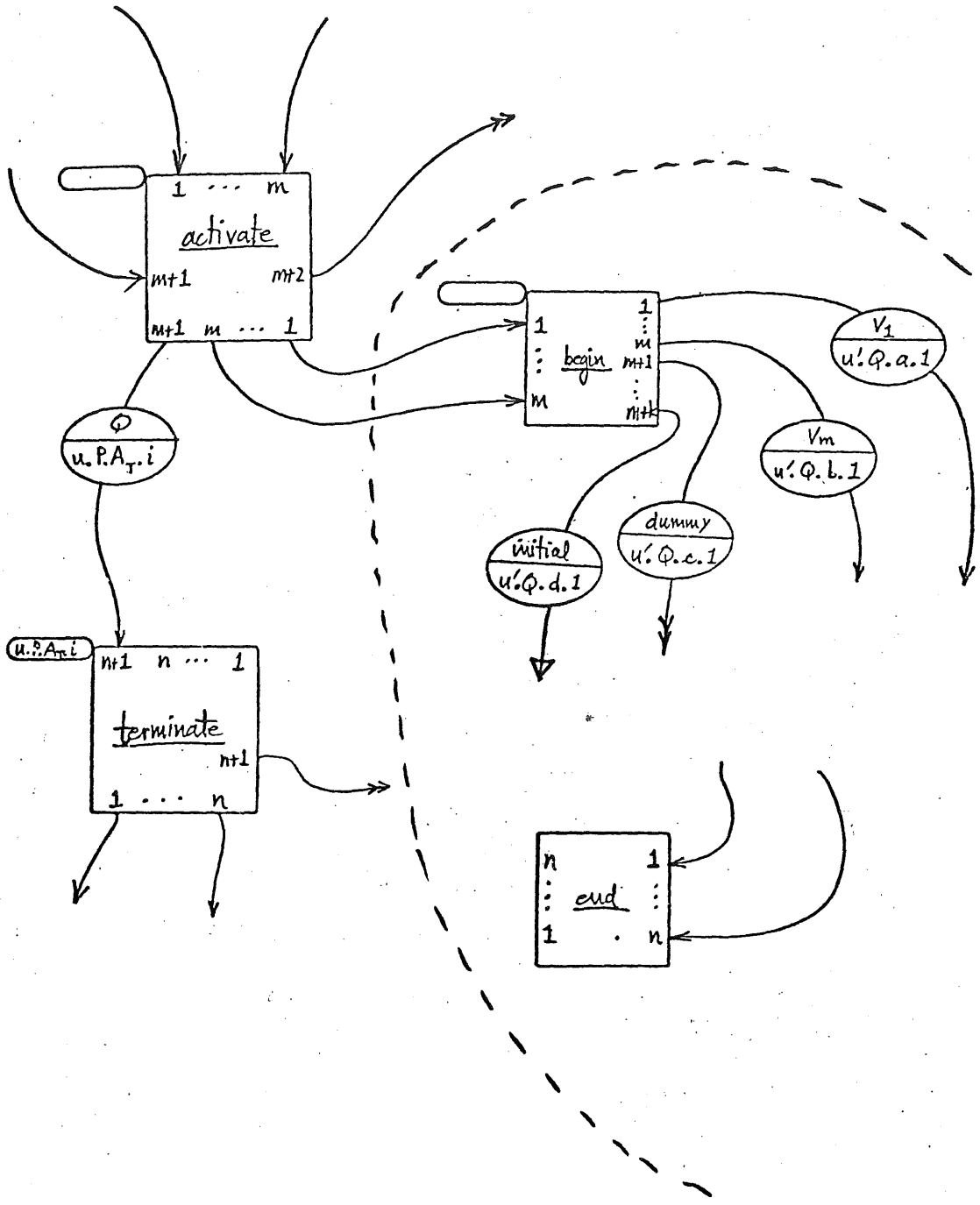


Figure 5h

begin starts the execution of the called procedure by outputting the initial, dummy, and argument tokens

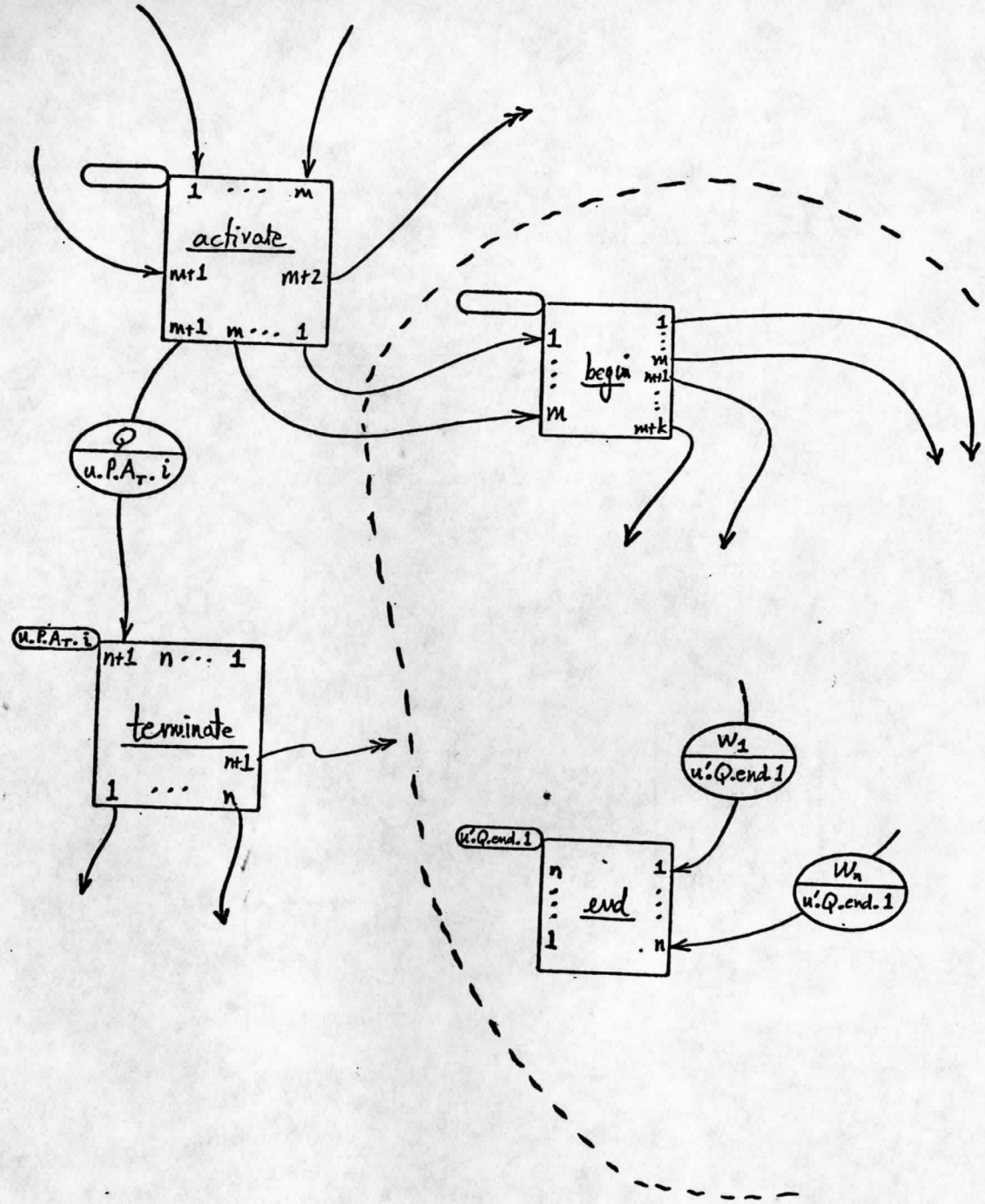


Figure 5i

Result tokens pass through the end statement at end of the called procedure

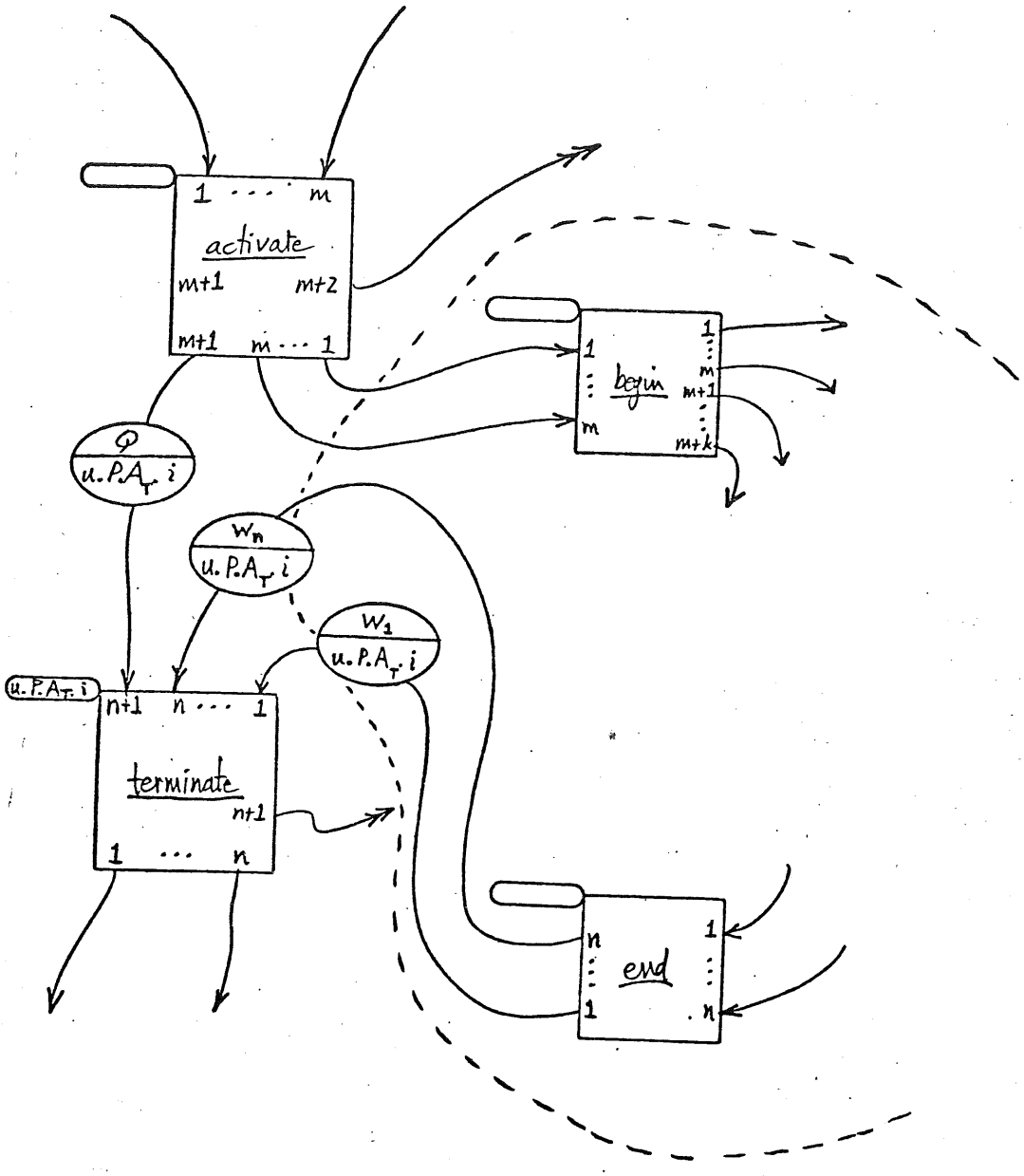


Figure 5j

The result tokens are returned to the calling procedure's domain

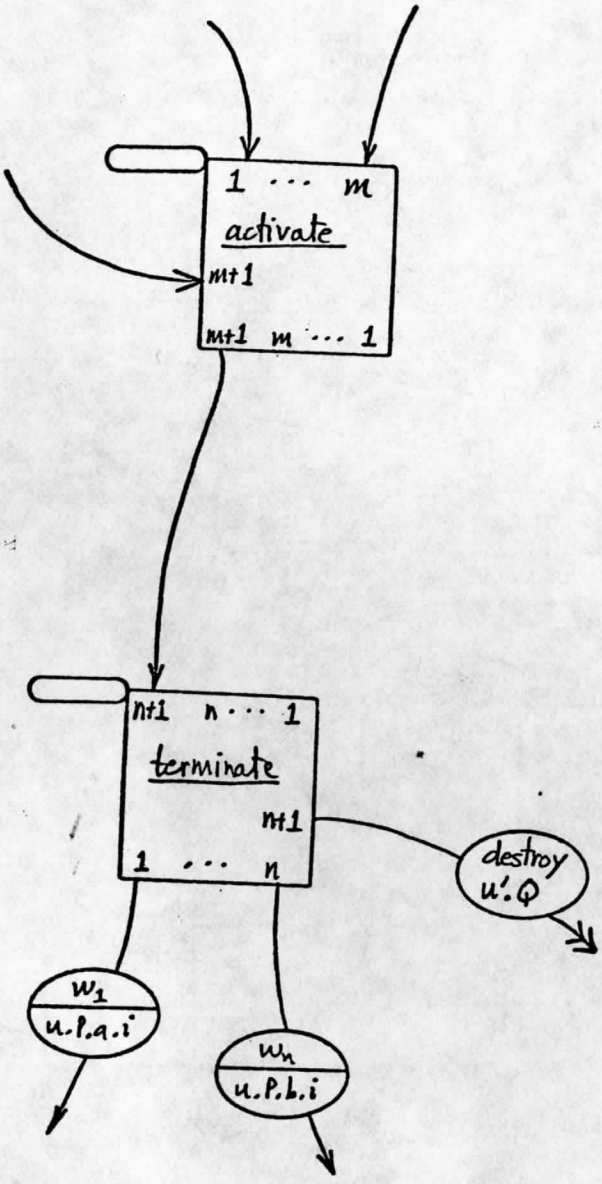


Figure 5k

The called procedure domain is destroyed and the results are distributed in the calling procedure's domain

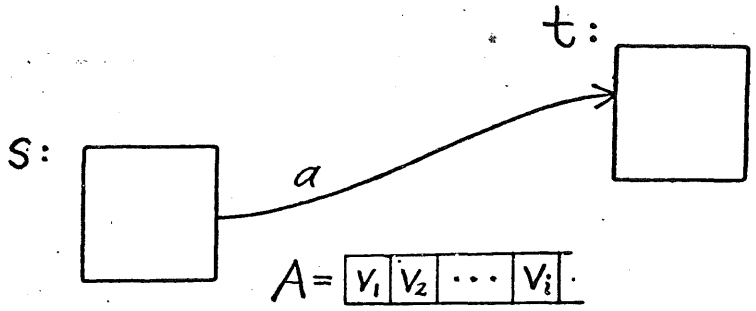


Figure 6a

Vector A associated with arc a with a sequence of values

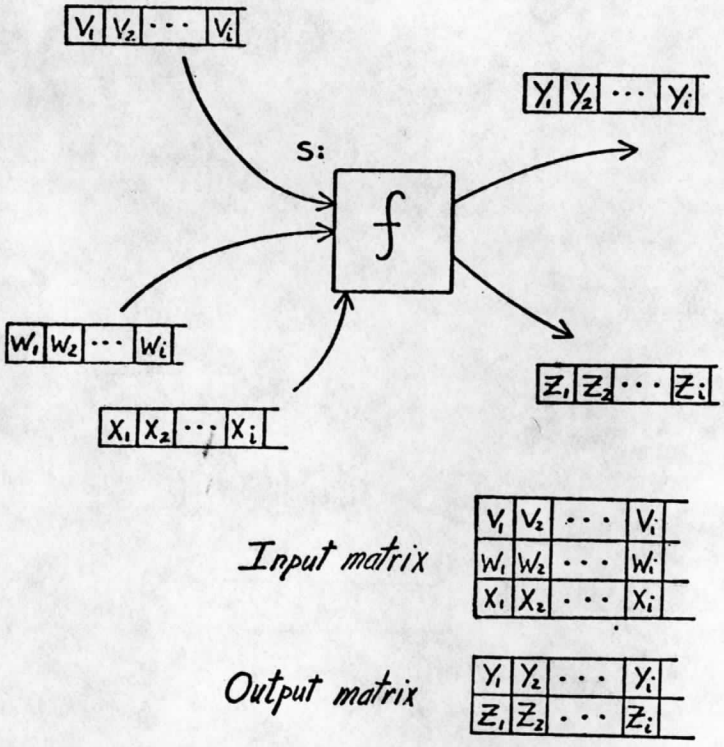


Figure 6b

Input and output matrices of a statement f

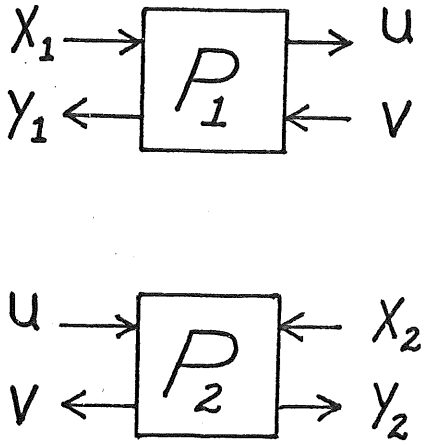


Figure 7a
Two program pieces P_1 and P_2

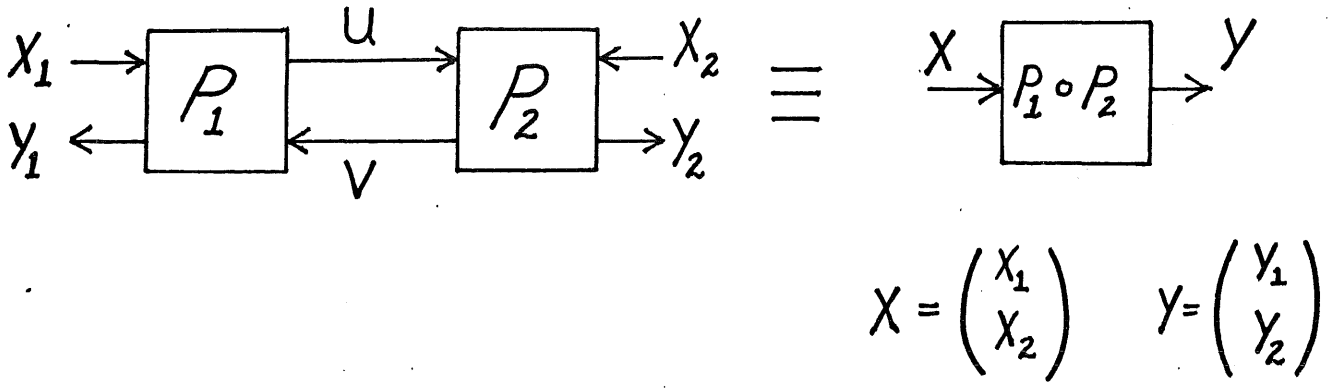


Figure 7b
The connection of P_1 and P_2 to form a new system