PREDICATE DRIVEN MODEL OF DATA FLOW COMPUTATION

Ashoke Deb*

Oregon Graduate Center
19600 N.W. Walker Rd.
Beaverton, OR 97006

OGC TR CS/E 83/008

Abstract**

The concept of data flow computation has several attributes and provides interesting solutions to problems encountered in von Neumann style of computation, namely simple synchronization mechanism, side effect free programming, increased exposure to parallelism and asynchronism. But, the conventional tools for representing a data flow computation is either based on imperative-style program structures or a very low level graph model, usually obtained from an imperative-style program. This makes the understanding and exploitation of parallelism difficult and also the underlying architecture becomes highly complex.

We propose a model, called Predicate Driven Model (PDM), for representing data flow computations. This model has several advantages:

i) the types of constructs, namely Function-node, Input initialization and Line composition, are minimal and simple,
ii) two synchronization tools, one by 'encapsulation' and the other by 'arrival', are available and implicit in the model,
iii) transformation to a graph notation is direct, and
iv) the elimination of imperative structures, exposes 'hidden' parallelism much more effectively than the traditional approaches.

KEY WORDS: Data flow computation, von Neumann system, Imperative programming, Functional programming, Single-assignment, Parallelism, Synchronization.

---

* On leave from Memorial University; St. John's, Newfoundland.

** This work was supported in part by the Natural Sciences and Engineering Council of Canada under Grant A-4176.

I. Background

A growing body of computer architects, language designers and software developers believe that the traditional ideas built around von Neumann model of computation should be abandoned.

From the architectural viewpoint, von Neumann designs have the following characteristics. (1) It distinguishes control elements from data operational elements. (2) Space (e.g., variable locations), as opposed to values, are the primary objects. (3) As a result of (1) and (2), control initiates operation whose domain and range are spaces. An operation modifies a space. This modified space is responsible for the next control signal. Thus, control-operation-space model works entirely on side effect (3). Even our notion of categorizing systems as (instruction stream, data stream) pair is an implicit way of classifying control-operation-space model.

From the language designer's viewpoint, the influence of von Neumann characteristics are obvious. (1) The most basic element in a language is a variable, which sometimes represents a value and sometimes a location, depending upon its context. (2) Most such languages are endowed with features facilitating side effects e.g., common block, call-by-reference, call-by-name, assignment statements of the form A := A + B, to name a few. (3) Transfer of control features e.g., GOTO, procedure CALL, conditional branch, came into existence in order to optimize the use of operations and spaces in that control-operation-space model.

Over the number of years, software developers were trained to be 'clever' users of side effects and control transfers. Sometimes higher level constructions were developed to hide the low level complications, but understanding and implementation of these high level constructs depended on extensive (and often complex) use of side effects and control transfer mechanisms. For example, (1) implementation of P, V operations on semaphores or other variations of it depend exclusively on side effects produced on the semaphore variable. (2) After a decade of realizing the danger of GOTO or its equivalent, these constructs are still well and alive. Mathematicians try to prove the correctness of a program with great difficulty and in a roundabout way. For example, it is difficult to mathematically justify A = A + B. Denotational semantics of GOTO is not easily given. Also, programs written using conventional constructs such as IF-THEN-ELSE, FOR-DO, Assignment etc. do not expose parallelly computable parts and unravelling of such structures is hard due to intricate 'data dependencies'.

Hence realized was the need of (1) computer architectures [2,4,6,8,10,14] that can free itself from the von Neumann design bottlenecks. (2) A language [1,2,3,9,13] that can be used to express parallelism within a computation, such that it is mathematically sound, free of side effects, free of complex control mechanics and also which is readable and descriptive of a computation.

At present, data flow computation [1,2,4-8,11,14] seems to provide a viable alternative to von Neumann computation. In data flow, 'values' as opposed to variable locations are considered as the primary objects. Operations are performed on operand values to produce result values. The usual interpretation of a data flow computation is that the computation is described as a network such that values flow (operand values arrive at the operation node and resultant values flow out of the operation node) through the network, finally producing the result.

To illustrate the difference between von Neumann computation and data flow computation, we show the following example.

Example:

To evaluate the following function

$$f(n) = n * (n - 1) * (n - 2)$$

von Neumann style of computation will be as follows:

```
        LOAD    N
        SUB     ONE
        STORE   TEMP
        SUB     ONE
        MULT    TEMP
        MULT    N

N       /* The value of n */
ONE     /* The constant 1 */
TEMP    /* A temporary variable location */
AC      /* An implicit variable location */
           used by an operatoin as accumulator */
```

On the other hand, data flow style of computation can be expressed as follows:

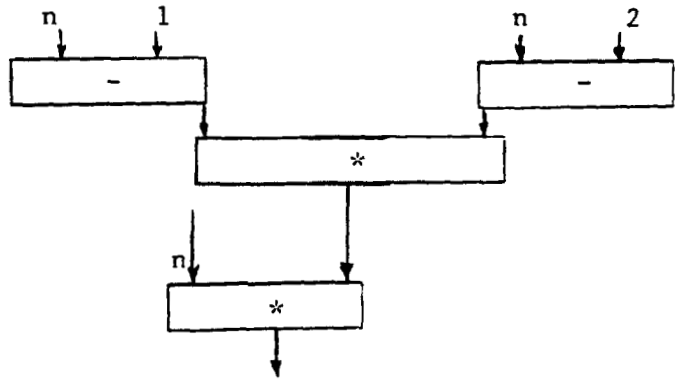

Figure 1.

Points of difference between these two styles of computation are that in the first method (1) variable locations are key objects to be manipulated, (2) side effects (e.g., AC = AC * TEMP and AC = AC * N etc.) are used to minimize the code length (hence execution time) and the storage space, (3) as a result of that, parallel computations are obscured, whereas in the second method (1') values are the key objects and functions are applied to produce result values, (2') an operation is activated by the arrival of valid input values, (3') there are no side effects and (4') parallelism of computations can be directly expressed and computations can proceed asynchronously.

End of Example.

Recently, functional (applicative) languages [1,2,7,9,13] have received a good deal of attention because (1) they do not use any side effects, (2) these were designed (in purest form) to manipulate values rather than variables and (3) being functional, mathematical theorems can be used more easily for reasoning.

To illustrate the difference between von Neumann (imperative) style and functional (applicative) style of programming, we show the following example.

Example:

To find the sum of the squares of the numbers from 1 to n.

In an imperative style this might be written as:

```
sum := 0;
FOR i := 1 to n DO
  sum := sum + square (i);
```

Above code does not need any explanation to most 'experienced' programmers, because they 'know' the meanings of 'sum := sum + square (i)' and 'FOR i := 1 to n DO' etc.

In applicative style one might write:

```
Listadd o Elementwise-square o Allnumbers [1,n]
```

The applicative program is shorter, free of intermediate variables and more meaningful to one who is not corrupted by imperative style of thinking. (Read the above program as "Add up elementwise square of all numbers from 1 to n".) In this, 'Listadd', "Elementwise-square', 'Allnumbers' are functions. 'o' is the function composition operator. A function application is right associative.

End of Example.

Currently, one of the major issues in the design of functional languages is to provide a set of functions and functional operators which can be used to construct other functions.

Aside from the mathematical elegance of functional languages, the current approaches do not provide enough 'granularity' to allow the programmer to dictate the flow of data. For example, in the last functional style program, if the programmer wants to modify the program so that computation terminates as soon as square (i) > BIGNUMBER, for some i, $1 \leq i \leq n$, and a constant BIGNUMBER, modification becomes implementation dependent and not so immediate. Also, once a function is defined, the domain and the range associated with the function is fixed. A simple way of imposing property on the domain (or the range) and associating this directly with a function definition is desirable, which seems to be a common usage in mathematics.

For example, given a definition of * (multiply), define F_2 such that F_2 (x,y) = r = x * y such that r ≠ 16, i.e., F_2 'produces' results of product of two numbers such that the result is not sixteen. In other words, the range of F_2 is a restriction of the range of *.

In the context of data flow computation, the direct association of properties (i.e., restrictions) on both the domain (i.e., input values) and the range (i.e., output values) of a function has important implications which will be discussed later in the paper.

Traditionally, a data flow computation is represented as a program by using a 'data flow language', such as ID or VAL [1,2], or by a data flow program graph, sometimes referred to as the base language of the data flow machine [5,6, 12].

In the following, we will discuss why the above tools are not adequate for representing data flow computations.

Data flow languages (e.g., VAL) allow structures such as IF-THEN-ELSE, WHILE-DO, Assignment, etc. for writing programs imperative style.

We will show an elementary data flow program written in VAL and its data flow graph.

Example:

(Assume that x and w are some given values.)

```
For  y := x; t := 0;
do   if t ≠ w then
       if y > 1 then iter y := y/2 enditer
                else iter y := y * 3 enditer
       endif;
       iter t := t + 1 enditer
       endif
endfor
```

End of Example.

Another commonly used tool for representing data flow computations is data flow graph whose constructs are based on 'links' and 'actors' and their 'firing rules'. In one such model [6], the links, actors and firing rules are as follows [Fig. 2-5]. Figure 6 shows the data flow graph of the example program written in VAL.

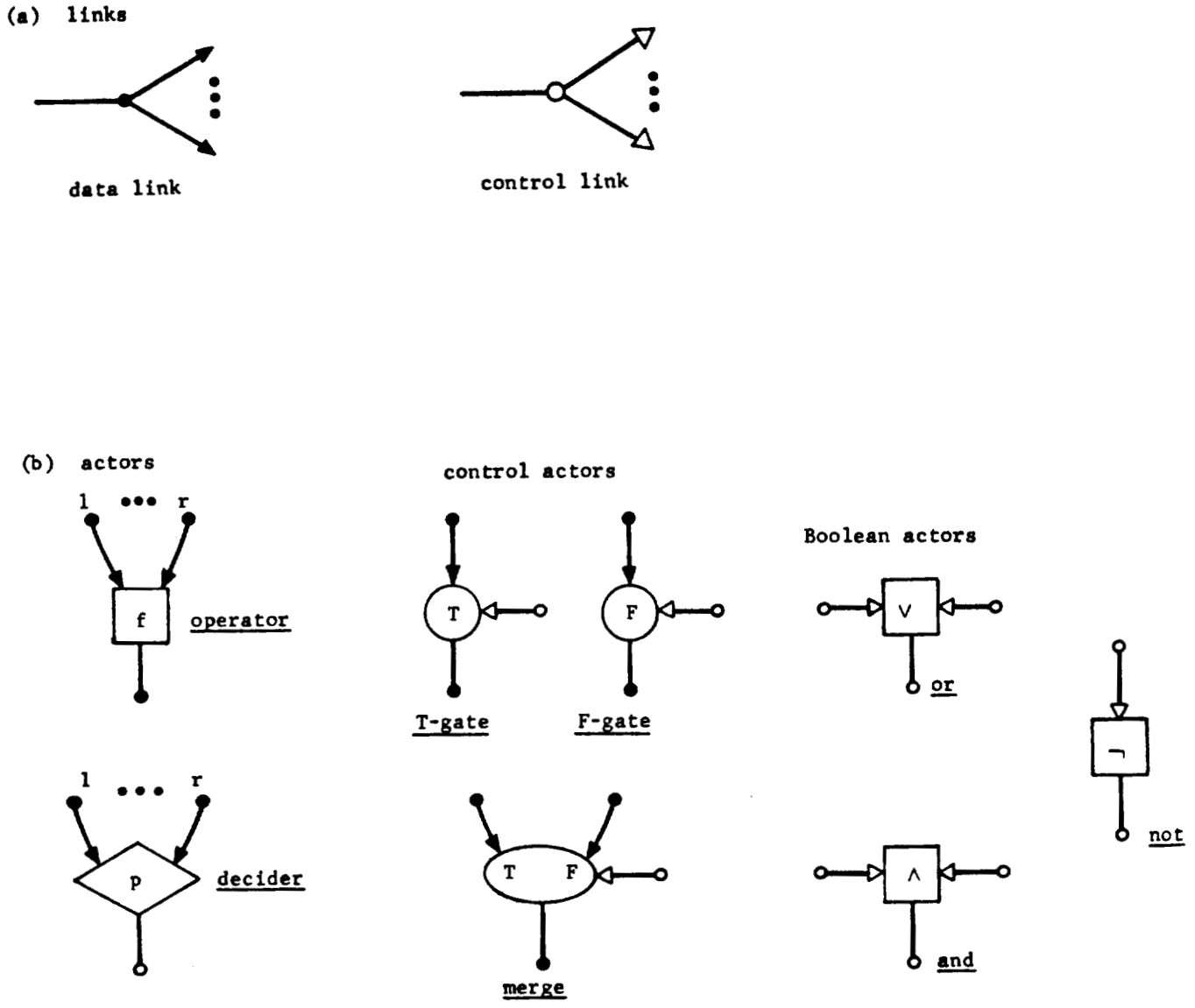

Figure 2: Node types for data flow program.

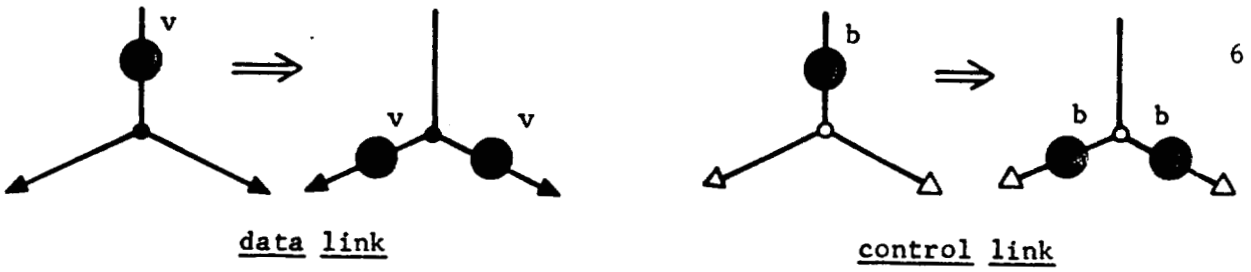

Figure 3. Firing rules for link nodes.

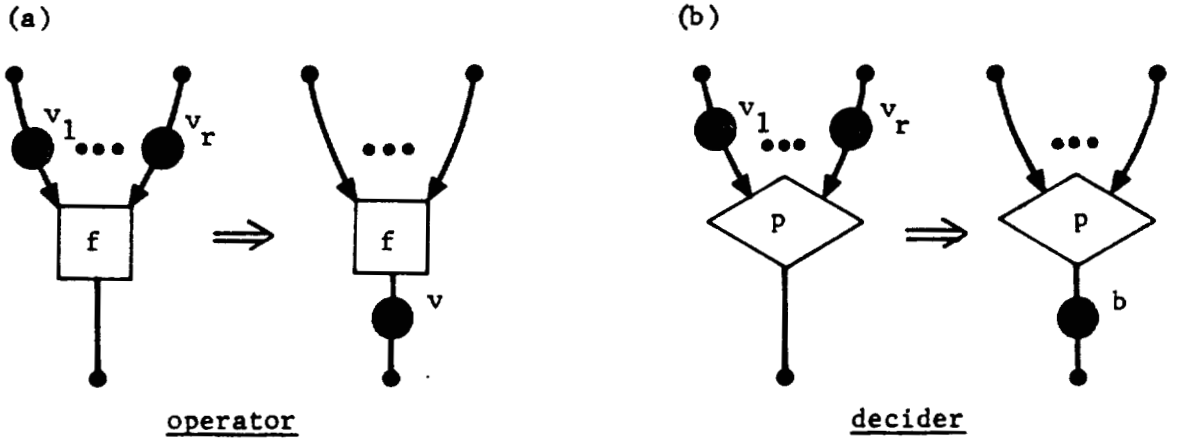

Figure 4. Firing rules for operators and deciders.

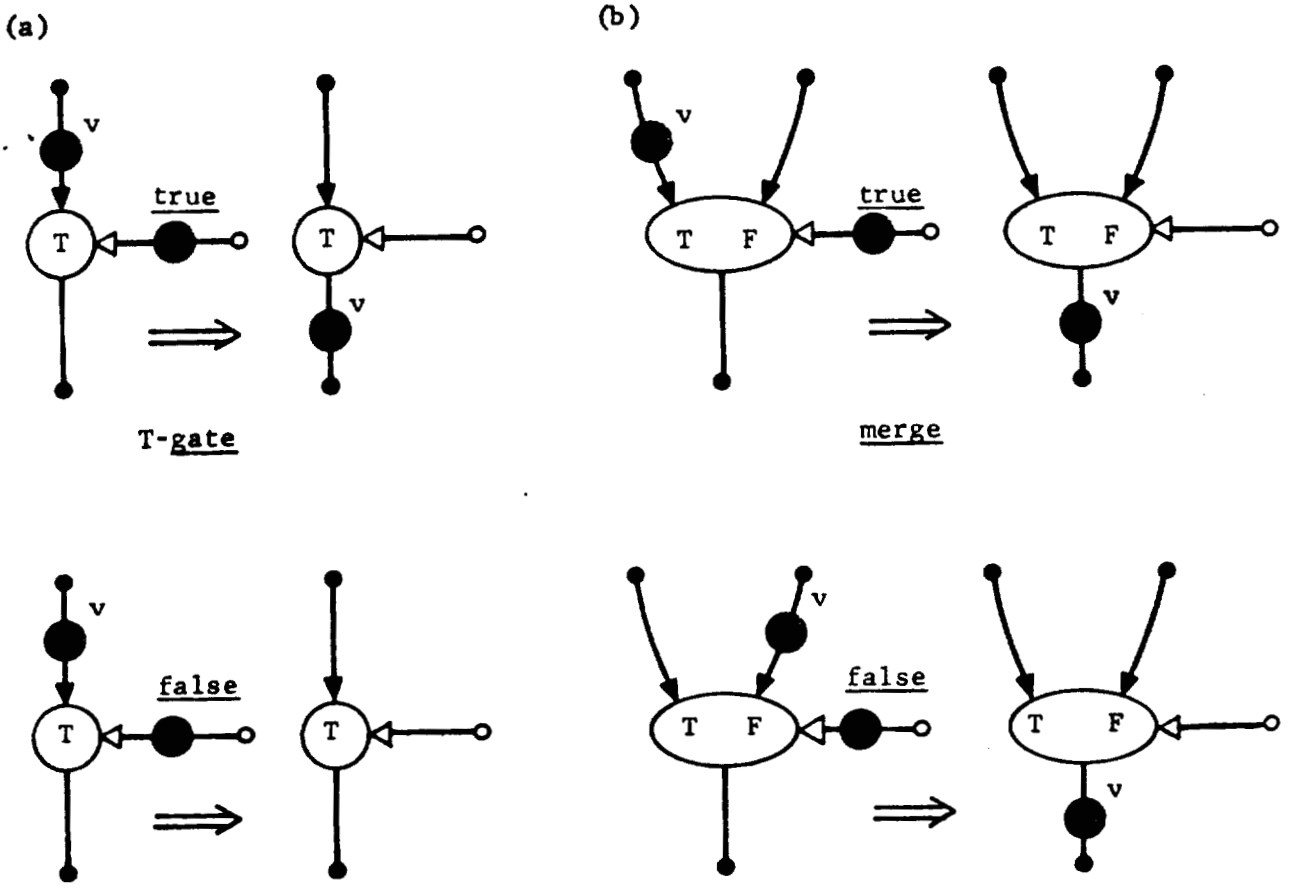

Figure 5. Firing rules for control actors.

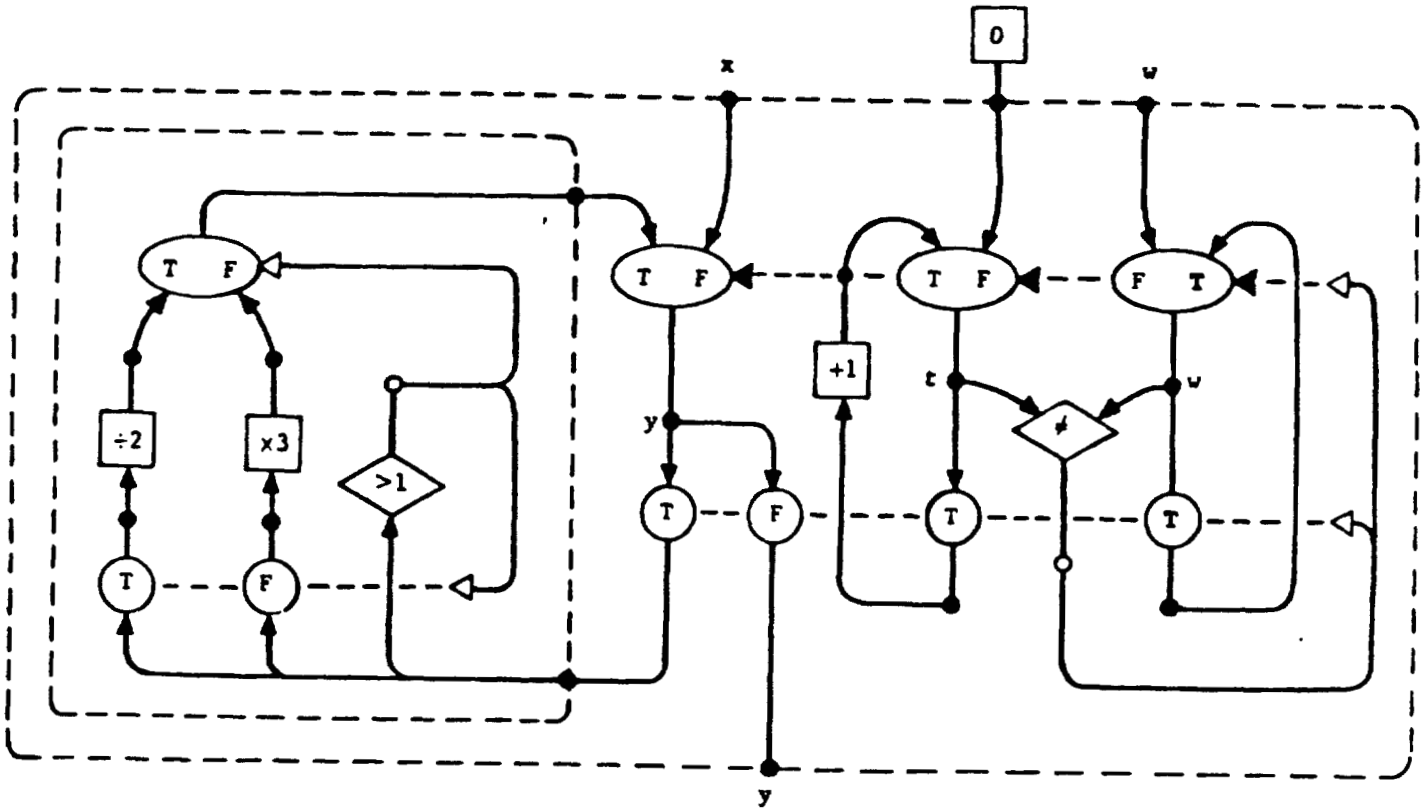

Figure 6: An elementary data flow program.

It should be noted that in VAL, expression of the form 'iter y := y/2' is different from the imperative form 'y := y/2' (as a matter of fact, VAL does not allow such assignments). 'iter y := y/2' like expressions can be used only inside a loop and essentially means that the 'value of y in the next iteration' is equal to the 'value of y in the current iteration' divided by 2.

Still a closer look at languages of this type and their semantic models reveal several drawbacks.

(1) Use of traditional constructs (or their modifications) such as IF-THEN-ELSE, WHILE-DO, etc. which do not lend themselves to clear specification of parallelly computable parts. In the last example, the fact that 'iter y := y/2', 'iter y := y/3' and 'iter t := t + 1' are parallel computations is not revealed. This is also the case in the graphical view of the computation.

(2) Although expressions of the form y := y/2 is not allowed and the 'principle of single assignment' is used, the fundamental idea, that the value of y from one cycle of computation is fedback (in true sense), is kept implicit. Notice that in von Neumann style of computation where controlling the instruction sequencing was an important issue, programmers were given facilities to accomplish 'flow of control'. On a similar basis, in data flow style of computation, programmers should be given facilities to directly specify the 'flow of data values'.

(3) Finally, we need a higher level semantic model, which will express functionality, parallelism and synchronization in a structured form.

In the next section, we present a model, called Predicate Driven Model (PDM), of data flow computation. Some of the important characteristics of the PDM are that (1) there are three main constructs which are used to represent a data flow computation, and (2) the semantics of these constructs rely on two levels of synchronization--by containment and by arrival, on queuing of the input values and on 'local' data validation at the input and output lines of a node. There are two other important features a special value λ and 'memory less' property of the output lines, which will be discussed later.

In the last section, we will discuss our future goals and areas of research.

II. Predicate Driven Model of Data Flow Computation

In PDM, a computation is a composition of functions, where:

i) Function-body is a definition of input to output mapping.
ii) Domain is a definition of all valid input values of a function-body.
iii) Range is a definition of all valid output values of a function-body.
iv) Activation of a function body takes place only if

a) previous activation of the function-body has completed
b) a set of all valid input values in the domain of the function has arrived.

Note that the input (output) values arriving at (leaving from) the function-body may not all be valid. Activation of a function-body does not necessarily require that output value(s) from its previous activation be consumed. Also, it is possible to have more than one function with identical function-body but different domains and/or ranges. Implications of these will shortly be explained.

Structure of a computation may thus be looked upon as a synthesis (or refinements) of several structures such that the range (or subrange) of one is a subset of the domain of another [Figure 7].

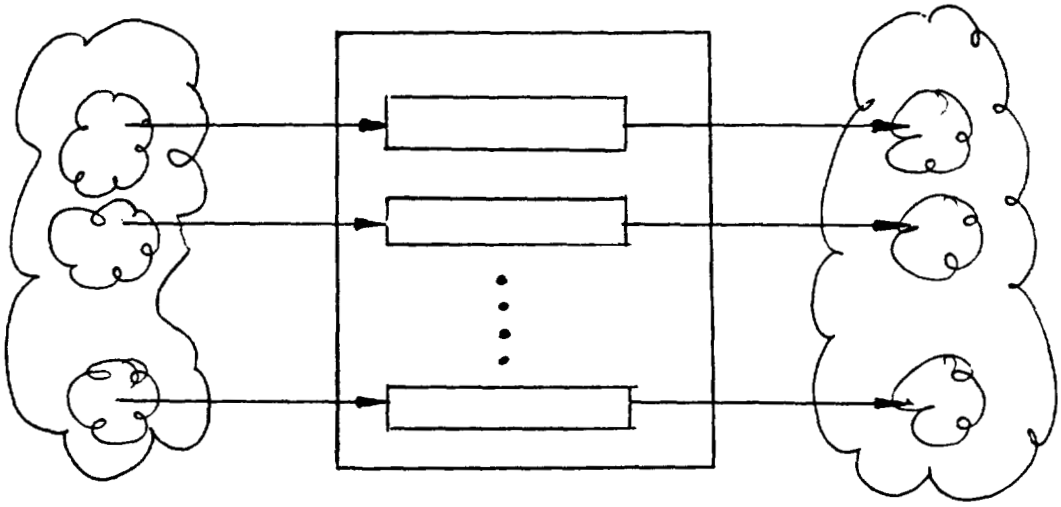

Figure 7: Structured representation of a computation, $D\{F\}R$, where F is the function-body, D is the domain, R is the range, which is a collection of lower level of computations $D_i \{F_i\} R_i$. $1 \leq i \leq n$ such that $R_j \subseteq D_k$ for some j, k's.

For the sake of subsequent discussions, we will make the following assumptions:

i) V is a set of values, $V = \{v_1, v_2, \ldots, v_N\}$, called the Universe of Discourse, which includes a special value λ. The highest level domain and range are subsets of V, e.g., V = {integers, λ}. The meaning and use of λ will be explained later.

ii) OP is a set of primitive operators, $OP = \{O_1, O_2, \ldots, O_n\}$, such that

$$O_i : v^{n_i} \rightarrow v^{m_i} ,$$

for some integers n_i and m_i and $v \in V$, e.g., OP = {+, -, /, *, Div, Mod, <, =, >} .

The PDM uses three basic constructs, namely Function Node, Line Composition and Input Initialization, to represent a computation.

Function Node:

Syntax of a function node f, without input initialization, is as follows:

f: D{F}R

where $D = [C_1]\ I_1\ [C_2]\ I_2 \ldots [C_n]\ I_n$,

$R = [C_1']\ O_1\ [C_2']\ O_2 \ldots [C_m']\ O_m$,

$F = f : (I_1, I_2, \ldots, I_n) \rightarrow (O_1, O_2, \ldots, O_m)$.

In the above,

f is the function name, which is optional,

$[C_1[\ I_1\ [C_2]\ I_2 \ldots [C_n]\ I_n$ is the list of predicated input lines,

$I_1, I_2, \ldots, I_n$ are symbolic names, called input lines,

$C_1, C_2, \ldots, C_n$ are predicates, called input predicates such that C_i is formed by using logical and relational operators from OP, constants from V and $I_1, I_2, \ldots, I_n$ such that any occurrence of I_j is free.

Similarly,

$[C_1']\ O_1\ [C_2']\ O_2 \ldots [C_m']\ O_m$ is the list of predicated output lines,

$O_1, O_2, \ldots, O_m$ are symbolic names, called output lines,

$C_1', C_2', \ldots, C_m'$ are predicates, called output predicates, such that each C_i' is formed by using logical and relational operators from OP, constants from V and $I_1, I_2, \ldots, I_n, O_1, O_2, \ldots, O_m$ such that any occurrence of I_k or O_j is free.

Lastly,

$\{f: (I_1, I_2, \ldots, I_n) \rightarrow (O_1, O_2, \ldots, O_m)\}$ is called the function body which consists of zero or many mappings of the form $E_j \rightarrow O_i$. E_j is an expression formed by using operators from OP, constants and V and $I_1, I_2, \ldots, I_n$. Also there does not exist two mappings in a function body such that $E_j \rightarrow O_i$ and $E_k \rightarrow O_i$ for $j \neq k$.

The semantics of a function node f, without input initialization, is as follows:

Let v be the value present in input line I_i (output line O_j) at an instant. v is said to be valid if the corresponding input predicate C_i (output predicate C_j') is true. If an input or output predicate is the constant TRUE, then it is usually omitted.

A function node is said to be firable if (i) the activity due to the last firing is complete and (ii) a set of valid input values are available in all input lines.

If a value v is present at the input line I such that v is not valid, then v is called invalid input and its effect is equivalent to the removal of the value v from line I.

Note: Since each input line validates its own value, it is possible to allow two input lines, I_1 and I_2, with predicates C_1 and NOT (C_1) respectively and the input value v to both I_1, I_2. This may provide for two distinct actions for two classes of input values to be taken. An example of this is where 'incorrect' values are used to produce error messages.
End of Note.

If a value v is obtained for output line O such that v is not valid, then the special value λ is produced at line O.

An input line I is allowed to maintain a sequence of values.

An output line O is not allowed to retain its value.

Firing of a function node is said to be complete when the entire function body has been executed, values have been produced at the output lines and the present input values have been removed from the input lines.

Note: According to the above requirements of the firing of a node, two levels of synchronization are present. One is achieved by the arrival of all valid input values. The other is achieved by the encapsulation of several operations of the form $E_j \rightarrow O_i$ within the function body. All such operations within a function body may proceed in parallel.

Also, a sequence of output values may be retained by 'channelling' those values to an input line.
End of Note.

In the following, we will present and discuss a few examples.

Example: Let

$$f_1 : [15 > I_1 > 10]\ I_1[I_2 = 5 \text{ OR } I_2 = 6]\ I_2\ \{I_1 * I_2 \rightarrow P\}\ [\ ]\ P \quad .$$

This is equivalent to the imperative style statement

IF ((15 > I_1 > 10) and ((I_2 = 5) OR (I_2 = 6)))

THEN P := I_1 * I_2 .

Also, let

$$f_2 : [\ ]\ I_1\ [\ ]\ I_2\ \{I_1 * I_2 \rightarrow P\}\ [15 > I_1 > 10 \text{ AND } (I_2 = 5 \text{ OR } I_2 = 6)]\ P$$

Although f_1 and f_2 appear similar, there are a number of important differences between them. In f_1, the function body will be executed for only a subset of integer for which the predicates (15 > I_1 > 10) and (I_2 = 5 OR I_2 = 6) are true. On the other hand, in f_2 there are no input predicates associated with it; the function-body of f_2 will be executed for all values of I_1 and I_2. Secondly, f_1 will never produce λ output, whereas f_2 may produce λ output (when the output predicate is false). Thirdly, in terms of actual execution sequence, in f_1, the evaluation of the input predicates and the execution of the function body is strictly serial. Whereas in f_2, the execution of the function body and the evaluation of the output predicate may proceed in parallel.

End of Example.

Example: Let

$$f : [I_1 > 5]\ I_1 \left\{ \begin{array}{l} \text{square } (I_1) \rightarrow P; \\ \\ \text{maxint} \rightarrow Q \end{array} \right\} \quad [P > \text{maxint}]\ P\ [P \geq \text{maxint}]Q$$

where 'square' is an element of OP and 'maxint' is an element of V.

A logically equivalent imperative style statement would be:

```
IF (I1 > 5) THEN
        BEGIN
            P := square (I1);
            Q := maxint;
            IF (P > Q) THEN Q := λ   ELSE P := λ
        END.
```

End of Example.

Another type of construct used in PDM, called Line Composition, is defined as follows.

Line Composition

The syntax of a line composition is as follows:

$$I \leftarrow O$$

where O and I are output and input lines, respectively.

A set of line compositions of the form $I_1 \leftarrow O$, $I_2 \leftarrow O$, $I_3 \leftarrow O$, ..., $I_p \leftarrow O$, called Fan-out Line Composition, is equivalently expressed as

$$I_1, I_2, I_3, \ldots, I_p \leftarrow O .$$

A set of line compositions of the form $I \leftarrow O_1$, $I \leftarrow O_2$, $I \leftarrow O_3$, ..., $I \leftarrow O_k$, called Fan-in Line Composition, is equivalently expressed as

$$I \leftarrow O_1, O_2, O_3, \ldots, O_k .$$

The semantics of line composition are as follows.

The meaning of $I_i \leftarrow O_j$ is that the value generated at O_j will arrive at I_i. The meaning of a fan-out line composition $I_1, I_2, \ldots, I_p \leftarrow O$ is that the value generated at O will arrive at each of I_i, for all i, $1 \leq i \leq p$.

The meaning of a fan-in line composition $I \leftarrow O_1, O_2, \ldots, O_k$ is that the value generated at each O_i will arrive (asynchronously) at I. The values arriving at I will be serialized according to their order of arrival. In case more than one value arrives at I at the same time, then they will be serialized arbitrarily.

In case the input line I has an input initialization sequence, the serialization of additional incoming values will take place following the initial sequence of values. (Input initialization is described later.)

In the following, we will present a few examples.

PDM provides a powerful tool for composing ('connecting) function nodes. Here we will illustrate three type of compositions, called Inside fit, Outside fit, and Partial fit. [See Figure 8]

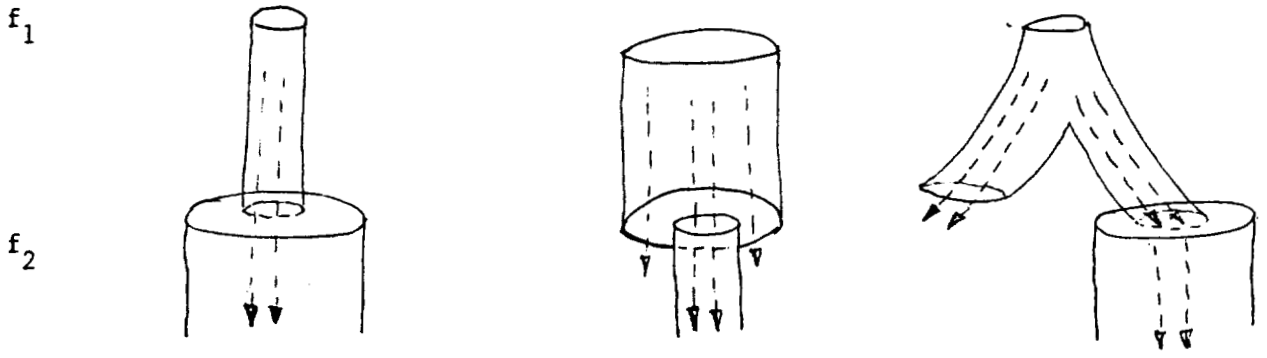

Figure 8. Inside fit Outside fit Partial fit

Let O = set of output values flowing from f_1 to f_2, and

D = domain of f_2.

We call the composition of f_1, f_2,

an inside fit, if $O \cap D = O$
an outside fit, if $O \cap D = D$ and
a partial fit, if $O \cap D \neq D$ or O.

A composition which is both an inside fit and an outside fit is called a total fit.

Example (Total Fit)

Let

$f_1 : [] I_1 \{f_1(I_1) \rightarrow O_1\} [] O_1$;

$f_2 : [] I_2 \{f_2(I_2) \rightarrow O_2\} [] O_2$;

$I_2 \leftarrow O_1$.

Here $O_1 = f_1(I_1)$ and $O_2 = f_2 \cdot f_1(I_1)$

End of Example

Example (Outside fit)

Let

f_1 : [] I_1 {$f_1(I_1) \rightarrow O_1$} [$5 \leq O_1 \leq 10$] O_1 ;

f_2 : [$I_2 = \lambda$] I_2 {$f_2(I_2) \rightarrow O_2$} [] O_2 ;

$I_2 \leftarrow O_1$.

In this example, what f_2 does is take an 'action' (denoted by the function f_2) when f_1 produces invalid outputs.

End of Example

Example (Partial fit)

Let the function $f_1(n)$ compute the nth element of the Fibonacci sequence and $f_2(x)$ compute x modulo 10.

f_1 : [n > 0] n {Fib (n) $\rightarrow O_1$} [] O_1 ;

f_2 : [even x] x {x mod 10 $\rightarrow O_2$} [] O_2 ;

$x \leftarrow O_1$.

The predicate even x is true if x is even.

End of Example

The following examples show iterative composition on a function.

Example (Indeterminate loop)

Let

f : [C] I {f(I) $\rightarrow$ O} [] O ;

I $\leftarrow$ O .

Here at termination, $O = f^n(I)$ such that $n \geq 0$, C is true for all $i < n$, when $O = f^i(I)$ and C is false when $O = f^n(I)$.

End of Example

Example (Determinate Loop)

Let

$$f : [\,]\ I\ [N \neq 0]\ N \begin{Bmatrix} f(I) \rightarrow O_1; \\ N\text{-}1 \rightarrow O_2 \end{Bmatrix} \quad [\,]\ O_1\ [\,]\ O_2\ ;$$

$$I \leftarrow O_1\ ;\ N \leftarrow O_2\ .$$

Here at termination, $O_1 = f^N(I)$.

End of Example

Example (Determinate loop with exit condition)

Let

$$f : [C_1]\ I\ [N \neq 0]\ N \begin{Bmatrix} f(I) \rightarrow O_1\ ; \\ N\text{-}1 \rightarrow O_2 \end{Bmatrix} \quad [\,]\ O_1\ [\,]O_2\ ;$$

$$I \leftarrow O_1\ ;\ N \leftarrow O_2\ .$$

Here at termination, $O_1 = f^n(I)$ such that $0 \leq n \leq N$ and for all $i \leq n$, $O_1 = f^i(I)$ and C_1 is true.

End of Example

Input initialization

Input initialization provides the facility in a function node definition where an input line may have a sequence of initial values.

The syntax of such construct is as follows.

I. init $(v_1, v_2, \ldots, v_p)$

where I is the symbolic name of an input line to a function node, init is a key word, $v_1, v_2, \ldots, v_p$ are elements of V.

The semantics of such construct is as follows.

For the first p instances, the sequence of values arriving at the input line I will be $v_1, v_2, \ldots, v_p$, in that order.

If additional values arrive at I (as a result of line composition) before its initial values $v_1, v_2, \ldots, v_p$ are consumed, the additional values will be serialized following v_p.

Example (Selective mapping)

Suppose we have a sequence of values $(v_0, v_1, \ldots, v_n)$ and a sequence of binary selection values $(b_0, b_1, \ldots, b_n)$ such that $f(v_i)$ is computed if $b_i = 1$.

f : X Y {f(I) → O} [] O where

X is [B ≠ 0] B.init $(b_0, b_1, \ldots, b_n)$ and

Y is [B ≠ 0] I.init $(v_0, v_1, \ldots, v_n)$.

End of Example

It is well known that asynchronism is used to achieve speed up. But the nondeterministic nature of it have led some to incorrectly assume that such computations are inherently difficult to express. The following example is presented to show that it is not the case. Simpler, terser and more transparent representation of a computation, which exploits asynchronism, than any imperative style of programming is demonstrated.

Example

Let $O_1, O_2, \ldots, O_n$ be the results produced by n independent computations and, hence, the arrival time of these results may be different. We want to compute the cumulative sum of these results. The following achieves this goal.

$$f : []I\ []\ S.init\ (0)\ \{I + S \rightarrow 0\}\ []0\ ;$$

$$I \leftarrow O_1, O_2, \ldots, O_n\ .$$

A comparative analysis of the above with an imperative style program is useful.

The following code

```
S := O_1 ;
FOR k := 1 TO n DO
    S := S + O_k ;
```

will add $O_1, O_2, \ldots, O_k$ exactly in that order but not as they are available.

End of Example

Before we conclude this section, we will present solutions of some common problems, as expressed in PDM, including the one illustrated in Section 1.

Example The following expresses the computation as expressed in VAL and the 'dataflow' diagram in Section 1.

$$f : [y \neq \lambda]\ y.init\ (x)\ [t \neq w]\ t.init\ (0) \left\{ \begin{array}{l} y/2 \rightarrow O_1\ , \\ y*3 \rightarrow O_2\ , \\ t+1 \rightarrow O_3 \end{array} \right\} [y > 1]\ O_1\ [y \geq 1]\ O_2\ []O_3\ ;$$

$$t \leftarrow O_3\ ;\ \ y \leftarrow O_1, O_2\ .$$

End of Example

Example Computing Factorial (n), $n > 0$.

$$f_0 : [I > 0]\ I.init\ (n)\ \{I - 1 \rightarrow O_1\}\ []O_1\ ;$$

$$f_1 : []\ Y.init\ (1)\ [x > 0]\ X.init\ (n)\ \{X * Y \rightarrow O_2\}\ []O_2\ ;$$

$$X, I \leftarrow O_1\ ;\ \ Y \leftarrow O_2$$

End of Example

Example Recognizing a string of matched and well balanced left- and right-parentheses. Assume that the string is given by $v_0, v_1, \ldots, v_n$ where v_i = (or) $0 \leq i \leq n$ and v_n = \$, end of sequence marker.

$$f : [] \; I_1.\text{init}(v_0) \; [I_2 \neq \lambda] \; I_2.\text{init}(v_1, \ldots, v_n) \left\{ \begin{matrix} I_1 \rightarrow O_2 \\ I_2 \rightarrow O_1 \end{matrix} , \right\} \; [\text{NOT C}] \; O_1 [\text{NOT C}] O_2$$

$$I_1 \leftarrow O_1 \; ; \quad I_2 \leftarrow O_2 \quad ,$$

where the predicate C is I_1 = (AND I_2 =), and NOT is logical negation.

End of Example

It should be pointed out that the algorithms used here are for illustration purposes only and are not meant to be "the" solution.

Example Job Dispatcher.

Previously, we studied problems where data is deterministically sent to a definite processor. In those examples, it is the arrival of data that starts a processor--i.e., a valid data arrives at a processor and waits until the processor becomes 'free'. Now suppose that there are M independent processors, all of which are capable of processing a data -- in other words, as far as a data is concerned, all processors are identical.

We also have N data items, which we would refer to as jobs.

The problem is to process N jobs on M processors as rapidly as possible.

Obviously, a deterministic passing of a job to a processor does not solve the above problem due to possible processor bottlenecks.

The solution also must incorporate the following facts that:

(1) if there is an available processor and there is an unprocessed job, the unprocessed job must be dispatched to an available processor.

(2) if there is more than one available processor, then exactly one should process the job,

(3) use minimum number of interprocessor communications and

(4) lastly and most important, the solution must not use any side effects.

In the following, we present such a solution expressed in PDM.

It should be noted that the dispatcher and the processors are assumed to be cyclic and the dispatcher at the beginning of its life will have the numbers of all the processors.

Dispatcher: $[\ ]$ AvailableProcessorNO $[\ ]$ Job.init $(Job_1, Job_2, \ldots, Job_N)$

$\{AvailableProcessNO \rightarrow O_1,\ Job \rightarrow O_1\}$

$[]O_1\ []O_2$;

$I_1, I_2, \ldots, I_M \leftarrow O_1$;

$JobR_1, JobR_2, \ldots, JobR_M \leftarrow O_2$;

Processor k: $[I_k = k]I_k\ []JobR_k$

$1 \leq k \leq M$ $\{Process\ (JobR_k) \rightarrow O_{1k}$;

$k \rightarrow O_{2k}$;

$[]O_{1k}\ []O_{2k}$;

$AvailableProcessNO \leftarrow O_{2k}$.

End of Example

III. DISCUSSIONS AND FUTURE GOALS

Earlier in the paper, we have mentioned that directly associating predicates on both the input and the output ends of a function-node have important implications in the context of data flow computations.

Some of the advantages are that:

(1) the representation of the computation becomes concise, because an additional function composition can be avoided.
E.g., $f_1 : [C_1]\ I\ \{f(I) \rightarrow O\}\ [C_2]\ O$

is equivalent to

$f_0 : [C_1]\ I_1\ \{f(I_1) \rightarrow O_1\}\ []\ O_1$;

$f_1 : [C_2]\ I_2\ \{I_2 \rightarrow O_2\}\ []\ O_2$;

$I_2 \leftarrow O_1$

(2) properties (of values) and their evaluations are kept 'local', thus avoiding 'superfluous' passing of values to another function node. E.g., suppose the predicate C in the following example, depends on inputs I_1, I_2 and the output O_1 .

$f : []\ I_1\ []\ I_2\ \{f(I_1, I_2) \rightarrow O_1\}\ [C]\ O_1$;

this would be equivalent to

$f_0 : []\ I_3\ []\ I_4\ \{f(I_3, I_4) \rightarrow O\}\ []\ O$;

$f_1 : []\ I_5\ []\ I_6\ [C]\ I_7\ \{I_7 \rightarrow O_1\}\ []\ O_1$

$I_3, I_5 \leftarrow I_1$; $I_4, I_6 \leftarrow I_2$; $I_7 \leftarrow O$.

(3) also, by placing predicates at the output end of the function node, parallelism may be exposed and exploited to a greater extend. E.g., suppose the predicate C in the following example depends on the input I .

In f: [C] I {f(I) → O} [] O , the predicate C has to be evaluated and if successful, then computation of the function-body may proceed. This may be necessary for some computation. But, there are computations where the predicate C is used only for selecting or not selecting a result of computation. In those cases, evaluation of C and the function-body could be done in parallel. That can be done in f: [] I {f(I) → O} [C] O .

At present, we have implemented an interpreter on VAX 11/780 under Unix operating system for experimental purposes.

So far, we have not addressed the issues related to dynamic data flow systems. We are currently working on several ideas which, we believe, will nicely extend the present level of PDM to dynamic systems.

REFERENCES

1. Ackermann, W. B. and J. B. Dennis [1979], "VAL - A value-oriented algorithmic language -- Preliminary Reference Manual", MIT/LCS/TR-218, June 1979, 80 pp.

2. Arvind, K. P. Gostelow and W. E. Plouffe [1978], "An asynchronous programming language and computing machine", TR 114a, Dept. of Info. and Computer Science, Univ. of California - Irvine, December 1978.

3. Backus, J. [1978], "Can programming be liberated from the von-Neumann style? A functional style and its algebra of programs", CACM, 21, 8, August 1978, pp. 613-641.

4. Comte, D., N. Hifdi and J. C. Syre [1980], "The LAU data driven multiprocessor system: results and perspectives", Proc. IFIP '80, Tokyo and Melbourne, October 1980, pp. 175-180.

5. Dennis, J. B. [1971], "On the design and specification of a common base language", Proc. of Symp. on Computers and Automata, Polytech. Press, Brooklyn, NY, 1971.

6. Dennis, J. B. [1980], "Data Flow Supercomputers", IEEE, Computer, November 1980, pp. 48-56.

7. Glauert, J. [1978], "A single assignment language for data flow computing", MSc Thesis, Dept. of Computer Science, Univ. of Manchester, January 1978.

8. Gurd, J. and J. Watson [1980], "A data driven system for high speed parallel computing", Computer Design, 19, 6-7, June/July 1980, pp. 91-100/97-106.

9. Henderson, P. [1980], Functional Programming -- Application and Implementation, Prentice Hall, 1980.

10. Magó, G. A. [1979], "A network of microprocessors to execute reduction languages", Int. JCSS, 8, 5 & 6, October-December 1979, pp. 349-385/435-471.

11. McGraw, J.R. [1980], "Data flow computing -- software development", IEEETC, C-29, 12, December 1980, pp. 1095-1103.

12. Rodriguez, J.E. [1969], "A graph model for parallel computation", MAC-TR-64, Project MAC, MIT, Cambridge, Mass., September 1969.

13. Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture, ACM #556810, October 1981.

14. Special Issue on Data Flow Computers, IEEE, Computer, February 1982.