PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

A DATA-DRIVEN MODEL FOR PARALLEL INTERPRETATION OF LOGIC PROGRAMS[1]

Lubomir Bic

Department of Information and Computer Science
University of California
Irvine, California 92717

ABSTRACT

A model for parallel interpretation of logic programs is presented. It is based on the idea of representing logic programs as graphs and graph templates in which resolution may be viewed as a process of graph matching. This process is carried out by tokens propagating asynchronously through the underlying graphs. The main problem is to decide how a given template should be distributed over as many independent tokens as possible and how to guide these through the underlying graphs. The scheme presented in this paper is based on transforming a given template into a spanning tree which permits both OR and AND-parallelism to be exploited during processing.

1. INTRODUCTION

New advanced technologies that have emerged in recent years have made it possible to manufacture large numbers of inexpensive components such as processing elements and memory units. When attempting to utilize this potential in the construction of highly parallel machines the inadequacies of von Neumann architectures have been recognized /Arvind and Iannucci 1983/ and resulted in the development of new models for parallel computations. One such model, referred to as **dataflow** /COMPUTER 1982, Treleaven et al. 1982/, is based on the principles of data-driven, asynchronous computation. A dataflow program is a graph consisting of nodes interconnected via directed arcs. Each node is an operator supported by a logically independent processing element; it is capable of accepting, processing, and emitting value tokens traveling asynchronously along the graph arcs. Hence it is the availability of data, rather than control signals, which trigger the execution of operations.

In /Deliyanni and Kowalski 1979/, the authors have discussed the idea of representing logic programs as collections of graphs and graph templates, called extended semantic networks, in which resolution may be viewed as a special form of pattern matching among graph structures. There are many possible schemes one could envision to implement the necessary graph matching procedures; the model presented in this paper is based on the principles of dataflow, where finding a given pattern is accomplished by an asynchronous propagation of tokens through the underlying graph. Since many processing elements may be engaged in the replication and forwarding of individual tokens, a high degree of parallelism can be achieved.

One of the main issues in this approach is to decide how a given graph template should be distributed over a number of independent tokens and how to guide these through the underlying graph. In /Bic 1984/ it was suggested to transform the pattern into a linear sequence in order to keep the guiding procedures as simple as possible. The linearization procedure was based on finding Euler paths through the given pattern. In this paper we will consider a more complex scheme based on constructing a spanning tree, which increases the amount of potential AND-parallelism during processing.

It should be mentioned at the outset that in this paper we are concerned with only 'pure' logic programming; it would be premature to include constructs that have been added to create an actual programming environment such as PROLOG. Furthermore, we will restrict the current model to only binary predicates as advocated in /Deliyanni and Kowalski 1979/. Finally, it should be mentioned at this point that the area of applications envisioned for the model is within the realm of database or knowledge base systems /Dahl 1982, Gallaire and Minker 1978, Minker 1978, Warren 1981/, as opposed to mathematically oriented computations.

2. REPRESENTING LOGIC PROGRAMS AS NETWORKS

A logic program is a set of **clauses** of the form

$$p_0 \leftarrow p_1, ..., p_n \ .$$

Each p_i is called a **literal** and has the form $p(t_1, ..., t_m)$, where p is a **predicate** symbol and $t_1, ..., t_m$ are **terms**. Terms may be constants, variables, or functors.

Figure 1 shows a sample program which records the relationships 'mother' and 'father' among individuals as assertions (lines 1-5) and the relationships

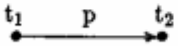
[1] *This work was supported by the NSF Grant MCS-8117516.*

'parent' and 'grandparent' as clauses with free variables (lines 6-8). Line 9 is the goal to be solved.[3]

(1) father(bill,john) ←
(2) mother(bill,jane) ←
(3) father(john,hans) ←
(4) father(jane,fred) ←
(5) mother(john,ann) ←
(6) parent(X,Y) ← mother(X,Y)
(7) parent(X,Y) ← father(X,Y)
(8) grandparent(X,Y) ← parent(X,Z), parent(Z,Y)
(9) ← grandparent(bill,Y)

Figure 1

We can transform any logic program (restricted to binary predicates) into a collection of graphs by representing each literal $p(t_1, t_2)$ as a directed arc of the form



The arrow head is used to record the order in which the terms of the literal were given. This information must be preserved when the literal represents an asymmetric relation.[4]

Literals sharing the same term result in arcs connected to one another via the corresponding node. Since many terms may be shared among different literals, graphs of arbitrary complexity may result.

Similar to the model presented in /Bic 1984/, we will distinguish two types of graphs: An **assertion graph** is constructed from the sequence of all assertions containing only ground terms (i.e. terms without free variables), and thus represents the collection of explicit facts. Figure 2 shows the assertion graph corresponding to the program of Figure 1. Note that multiple occurrences of any ground term are mapped onto the same node of the assertion graph.

The assertion graph is assumed to be a **dataflow graph** which implies that each node is an active element capable of receiving, processing, and emitting value tokens traveling asynchronously along the graph arcs.

All other clauses are interconnected via pointers into a directed structure as follows: a literal in the body of a clause points to all clauses whose heads are unifiable with that literal. This (possibly cyclic) collection of graphs will be referred to as the **goal structure** and may be interpreted as follows: a literal L with pointers to other clauses may be solved either by unifying L itself with one of the assertions in the assertion graph, or by solving one of the clauses pointed to by L. Figure 3 shows the goal structure constructed from the program in Figure 1.

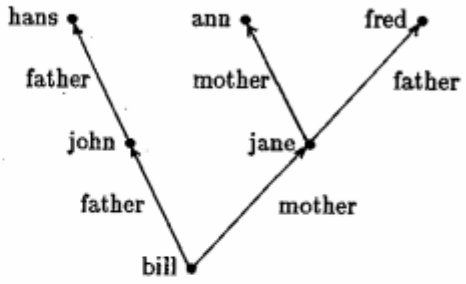


Figure 2

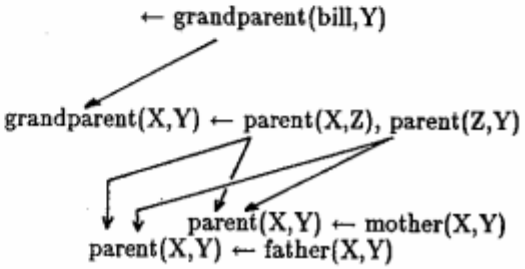


Figure 3

The body of each clause in the goal structure may itself be viewed as a graph, similar to the assertion graph, if terms are interpreted as nodes interconnected by arcs. Since each such clause usually contains free variables it will be referred to as a **graph template**. For example, Figure 4a shows the graph template corresponding to the initial goal (line 1) of Figure 3. Similarly, Figure 4b shows the graph template corresponding to the body of the clause on line 2 of Figure 3. (The variable X has already been bound to the constant bill.)

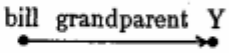


Figure 4a

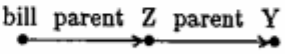


Figure 4b

Notation: To avoid drawing an excessive number of figures, we will use the following notation to denote an arc labeled p between two nodes T_1 and T_2: $T_1 \xrightarrow{p} T_2$. Arcs sharing a common node are joined to form connected sequences. For example, the graph template of Figure 4b, would be transcribed as bill $\xrightarrow{parent}$ Z $\xrightarrow{parent}$ Y.)

[3] *Throughout this paper, lower case letters are used to denote constants while capitals are used to denote free variables.*

[4] *Terminology: Since an arc is just another way of representing the same information contained in a literal, we will use the expressions 'literal' and 'arc' as synonyms. Similarly, the expressions 'term' and 'node' will refer to the same concept.*

3. STRATEGIES FOR GRAPH MATCHING

Assume an arbitrary sequence of literals p_1, ..., p_n is to be solved. In order to exploit parallelism within such a clause (referred to as AND-parallelism), it is desirable to process as many literals p_i concurrently as possible. This, however, is limited by free variables shared among different literals since each such variable must be bound to the same term during execution. As a first step, we can divide the original sequence of literals into groups such that any two literals belong to the same group if and only if they share at least one free variable. Each such group will be referred to as a **cluster**. From its definition it follows that any cluster may be fitted into the assertion graph independently since no free variables are shared among clusters. Hence we can concentrate on finding procedures for fitting individual clusters rather than arbitrary templates.

The fitting of a cluster should be carried out by tokens propagating through the underlying dataflow graph. In the most general case, the template formed by a given cluster is an arbitrarily interconnected graph. Since tokens in dataflow systems propagate asynchronously and there is no centralized control, the detection of cycles is difficult. To alleviate this problem, we can attempt to transform the given template into some other form, for which simpler token-guiding procedures can be devised.

The simplest case would be a transformation into a connected linear chain of literals. This can be accomplished by first transforming the template into a graph with a circular Euler path and then, by retracing that path, deriving the desired linear chain of literals. Unfortunately, this transformation requires certain literals to be replicated in order to obtain an Euler path, and thus causes unnecessary repeated unifications to be carried out. This wasteful behavior can be eliminated by forming several edge-disjoint paths through the template and retracing these in sequence; this is the approach discussed in /Bic 1984/.

Transforming the template into a linear chain prevents any AND-parallelism to be exploited within a cluster. This situation can be improved if the template is transformed into a tree structure instead. One possible approach is to find a depth first spanning tree, which partitions the template into two sets of edges – *tree* edges and *back* edges. (A procedure for constructing such spanning trees may be found, for example, in /Aho et al. 1976/.) Figure 5b shows the spanning tree constructed for the template of Figure 5a; back edges are shown as dotted lines. By replicating the nodes at which back edges terminate, we obtain a tree structure as shown in Figure 5c.

At each node of the template, the corresponding subtrees represent independent AND-goals; these can potentially be processed in parallel since no free variables are shared among subtrees. For example, when binding the root T_1 of the tree in Figure 5c the two newly created leaves (both T_1) are bound as well and hence the two subtrees form independent clusters.

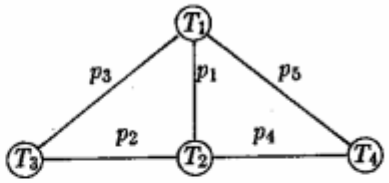


Figure 5a

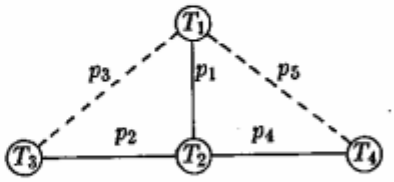


Figure 5b

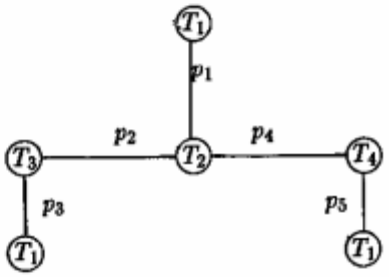


Figure 5c

To simplify the subsequent presentation, we assume that subtrees will be processed one at a time. This can be accomplished by transforming the given tree into a linear sequence of nodes and arcs using, for example, a preorder traversal algorithm. The following section develops the necessary procedures for fitting such (linear) sequences into the underlying dataflow graph.

4. SOLVING GOALS USING TOKEN PROPAGATION

4.1. Subgoals without Pointers

The sequence of literals constituting the body of a clause is usually referred to as a **goal** while each of the individual literals is called a **subgoal**. We first consider subgoals without pointers to other clauses. In the graph representation, solving such a subgoal $p(T_1, T_2)$ corresponds to the following *graph fitting* problem: determine possible bindings for the terms T_1 and T_2 such that the graph template $T_1 \xrightarrow{p} T_2$

matches some arc of the assertion graph. Operationally, this is accomplished by placing the graph template on a token and injecting it into specific nodes of the assertion graph. From each of these nodes the token is replicated along existing arcs in an attempt to find a match. We can distinguish the following four cases:

(a) Both nodes T_1 and T_2 are bound to ground terms t_1 and t_2, respectively. Since there can be only one occurrence of each of the nodes t_1 and t_2 in the assertion graph, the token is injected into one of these, say t_1. This node then replicates the token along all arcs labeled p that emanate from t_1. If one of the nodes receiving the replicated token matches the second term t_2, the subgoal is solved successfully; otherwise there is no direct match for this template.
The same result is obtained when the token is initially injected into t_2 from which it replicates in a search for t_1. This will be denoted by reversing the direction of the arc: $T_2 \xleftarrow{p} T_1$.

(b) The node T_1 is bound to a ground term t_1 while the node T_2 is a free variable. As in the first case, the token is injected into the node t_1 from which it is replicated along all arcs labeled p. This time, however, any node t_2 receiving the replicated token may be bound to the variable T_2 and hence presents a solution to the given subgoal $p(T_1, T_2)$.

(c) The node T_2 is bound to a ground term t_2 while the node T_1 is free. In this case, reversing the arc to $T_2 \xleftarrow{p} T_1$ yields a situation analogous to (b), where the first term is bound while the second is free. Hence the same approach can be taken.

(d) Both variables T_1 and T_2 are free. This case differs from the previous three in that there is no unique injection point for the token. Rather any node of the assertion graph is a potential binding for either variable and hence the token must be injected into *all* nodes of the assertion graph.[5] Each of these nodes binds the first variable T_1 to its own content and replicates the token along all arcs labeled p in the same way as described under (b). In other words, the search is started in all nodes simultaneously.

4.2. Sequences of Subgoals without Pointers

In this section we extend the scheme for solving individual subgoals presented above to cope with sequences of subgoals of the form

$$p_1(T_{1.1}, T_{1.2}),\ p_2(T_{2.1}, T_{2.2}),\ \ldots,\ p_n(T_{n.1}, T_{n.2}),$$

where each $T_{i.1}$ must satisfy one of the following restrictions:

(1) it must match the term $T_{(i-1).2}$, or

(2) it must be bound to a constant.

In the first case the arcs corresponding to the two adjacent literals p_i and p_{i-1} are connected, resulting in a graph template of the following form:

$$\ldots \xrightarrow{p_{i-1}} T_{i.1} \xrightarrow{p_i} \ldots .$$

In the second case, where $T_{i.1}$ is bound to a constant, we introduce a dummy arc, denoted as $\Longrightarrow$, to create an artificial connection between the two adjacent literals p_i and p_{i-1}, i.e.,

$$\ldots \xrightarrow{p_{i-1}} T_{(i-1).2} \Longrightarrow T_{i.1} \xrightarrow{p_i} \ldots .$$

A sequence of literals where each $T_{i.1}$ satisfies one of the two above restrictions has the following important property: all literals in that sequence, except possibly the first, will have at least one term bound when the sequence is processed from left to right. We will refer to such a sequence as a **linear form**. Note that any tree may be transformed into this form by applying a simple preorder traversal.

Assuming that none of the subgoals p_i has a pointer to other clauses, the processing of a linear sequence then corresponds to the following *graph fitting* problem: determine possible bindings for all terms $T_{i.j}$ such that the graph template matches some path in the assertion graph. Operationally, this is accomplished as follows: A token, carrying the entire graph template, is injected into nodes of the assertion graph that may be bound to the first term $T_{1.1}$. (As was the case with individual subgoals, only one such node t_1 will exist if $T_{1.1}$ is bound to a ground term; otherwise the token must be injected into all nodes of the assertion graph.) Each node t_1 receiving the injected token will replicate it along all arcs that match the template arc p_1. Each of the nodes t_2, receiving the replicated token, will attempt to bind itself to $T_{1.2}$. At this point, the next arc of the template is either a regular arc p_2 or it is a dummy arc $\Longrightarrow$. In the first case, t_2 would continue the propagation of the token along all arcs matching the name p_2. In the second case, the token is simply sent to the node that matches the corresponding term $T_{2.1}$. There is at most one such node since, by definition, $T_{2.1}$ must be bound.

The same steps are performed by all nodes t_i receiving the token, which results in a stepwise expansion of the graph template into all possible directions of the assertion graph. Each branch continues to grow until one of the following conditions occur:

(a) A node t_i is unable to bind itself to the corresponding node $T_{i.2}$ (i.e., $T_{i.2}$ is already bound to a term different from t_i), or, no arc emanating from t_i matches the corresponding template arc p_i. In this case a special token, (called end-of-stream as will be discussed in Section 5.1), which indicates that no solution can be found along this

[5] *In terms of a conventional implementation, the ability to inject a token into a node corresponds to indexing on arguments rather than on predicate names. Currently we are investigating a scheme which would correspond to indexing on predicate names. In this case the token would not have to be replicated to all nodes of the assertion graph but only to those connected to an arc labeled p. This could be viewed as injecting the token into specific arcs instead of nodes and would considerably reduce the number of injected tokens.*

path, is returned by the node t_i to the sender of the received token.

(b) The last node $T_{n.2}$ of the template has been reached, implying the detection of a match for the given graph template. At this point, a **reply token**, carrying all the bindings made during the forward propagation, is created and returned along the same path to the original injection point. It represents one complete solution to the original goal (the linear sequence).

4.3. Goals with Pointers to Other Clauses

The scheme described so far only finds solutions that result from processing the given goal against the collection of all assertions; no clause substitutions were considered. We now extend the scheme to utilize all clauses that may contribute to solving the given goal.

Consider the general situation depicted in Figure 6, where p is the goal to be solved.

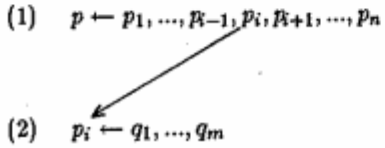


Figure 6

There are two possible sequences of literals that may yield independent solutions for p. These are

(a) The original sequence $p_1, \ldots, p_n$, and

(b) The sequence $p_1, \ldots, p_{i-1}, q_1, \ldots, q_m, p_{i+1}, \ldots, p_n$, obtained by replacing p_i in the original sequence by the sequence pointed to by p_i.

Note that both sequences have the first $i-1$ literals in common. We will use this fact to extend the previous scheme as follows:

To solve the goal p, a graph template corresponding to the sequence $p_1, \ldots, p_n$ is placed on a token and starts expanding from an injection node t_1 into all possible directions as described in Section 4.2. In addition, each time an arc p_i with pointers to other clauses is encountered a new branch of search is started by the node t_i processing the token: it fetches the clause pointed to by p_i, forms a new graph template consisting of the literals $q_1, \ldots, q_m$ and a copy of the yet unused portion of the current sequence $p_{i+1}, \ldots, p_n$, and starts replicating the new token along all appropriate arcs in the same way as the original token. It then waits for responses to both types of token, which will represent independent solutions to the templates $p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_n$ and $p_1, \ldots, p_{i-1}, q_1, \ldots, q_m, p_{i+1}, \ldots, p_n$, respectively.

The following section formulates the exact procedures to be performed by each node of the dataflow graph upon receiving a token.

5. PROCEDURES FOR TOKEN PROPAGATION

The semantics of a general dataflow system may be defined by specifying the procedures to be performed by each graph node when receiving a token. Each such procedure is invoked as soon as the necessary input tokens have arrived and it causes the generation of result tokens which are forwarded to other nodes. While the model proposed in this paper differs in many respects from a general dataflow system, it can be defined in terms of similar procedures, triggered solely by the arrival of tokens. Hence *the model is strictly data-driven – there is no need for any centralized control to synchronize concurrent operations.*

5.1. Generation of Activity Names

Before presenting the actual procedures, we need to introduce a scheme which would permit individual nodes to keep track of concurrent activities started in response to a received token, and to await the corresponding response tokens. This scheme is based on the principles employed in general dataflow systems /Arvind et al. 1978/: Each token, in addition to carrying the necessary data, contains a unique identifier called an **activity name**. This name is used by receiving nodes to disambiguate the various tokens traveling asynchronously through the graph.

The basic principles governing the generation and use of activity names is as follows. There are two types of tokens in our system: **regular tokens**, which propagate forward in an attempt to find a match for the graph templates they carry, and **reply tokens** which return along the same paths in the opposite direction and report the bindings made during the forward propagation. Whenever a regular token is propagated forward, its activity name is extended by appending to it a new component generated by the sending node. Thus activity names have the form $a_1.a_2. \ldots .a_n$, where each component a_i is an integer appended to the activity by a different node. Similarly, each time a reply token is propagated backward, the rightmost component of the activity name is detached by the sending node. Hence, within each node, activity names provide the necessary matching information. The following paragraphs discuss the exact form of activity names and their generation.

Assume that a node t_i has just received a token carrying the graph template $T_i \xrightarrow{p_i} T_{i+1} \xrightarrow{p_{i+1}} \ldots \xrightarrow{p_{n-1}} T_n$, where any of the arcs $\xrightarrow{p_j}$ could be a regular arc, representing a literal, or a dummy arc $\Longrightarrow$, as described in Section 4.2. Assume further that the activity name carried by this template is $a_1.a_2. \ldots .a_i$, which we shall abbreviate as $\bar{a}$. As described in Section 4.2, the node t_i will replicate the token along all arcs labeled p_i. These tokens will be given the activity names $\bar{a}.1, \bar{a}.2, \ldots, \bar{a}.p$ constructed by concatenating the original name, $\bar{a}$, with a new component – an integer ranging from 1 to p, where p is the number of

arcs matching p_i. All these activities are recorded by the node t_i as pending, that is, tokens with matching activity names are expected to arrive.

In addition to replicating the token along the p_i arcs, the node must start a new activity for each clause pointed to by p_i, as was described in Section 4.3. These activities will be assigned the names $\bar{a}.(p+1), \bar{a}.(p+2), ..., \bar{a}.(p+k)$, where k is the number of pointers from p_i. Each such activity is started by fetching the clause pointed to by p_i and converting it into a set of linear clusters. Hence several tokens, each carrying one cluster, are created for such an activity. These tokens will be distinguished by subactivity names of the form $\bar{a}.[(p+j).1]$, $\bar{a}.[(p+j).2]$, ..., $\bar{a}.[(p+j).l]$, where l is the number of clusters (subactivities) comprising the activity $\bar{a}.(p+j)$, for $1 \leq j \leq k$.

The following sequence summarizes the complete set of activities generated by a node when receiving a token with activity name $\bar{a}$:

$\{\bar{a}.1\}...\{\bar{a}.p\}\{\bar{a}.[(p+1).1], ..., \bar{a}.[(p+1).l_1]\}$ ...
... $\{\bar{a}.[(p+k).1], ..., \bar{a}.[(p+k).l_k]\}$

Activities enclosed in curly brackets represent OR-activities; each yields an independent solution to the received cluster. Subactivities within curly brackets represent AND-activities; all must be solved in order to obtain a solution to the corresponding OR-activity.

One more construct must be introduced before the procedures can be presented: Note that any number of reply tokens (including zero) could be received by a node for a pending activity. Due to the asynchronous nature of the model it is not possible for a node to determine when all reply tokens for a given activity have arrived. In order to solve this problem we introduce a special type of token, called **eos-token** (for end of stream), similar to that used in general dataflow systems /Arvind et al. 1978/. An eos-token, identified by an activity name, is sent by a node after all reply tokens for that activity have been emitted. It carries the number of these reply tokens which permits the receiving node to determine when all have arrived.

5.2. Procedures

This section defines the semantics of the model by specifying the procedures to be executed by a graph node upon receiving a token. The first procedure is executed when a *regular token*, carrying a graph template, is received. It causes the forward propagation of such tokens as was discussed in Section 4. The second procedure is executed when a *reply token*, carrying the bindings made during the forward propagation, is received. It causes the backward propagation of the reply tokens. Finally, the third procedure is invoked when an *eos-token* is received. These tokens, which follow sets of reply tokens, terminate the activities along the paths.

1. Procedure performed by a node t_i upon receiving a *regular token* T from a sender S; each such token carries the following information:

   activity name: $\bar{a}$

   graph template: $T_i \xrightarrow{p_i} T_{i+1} \xrightarrow{p_{i+1}} \ ... \ \xrightarrow{p_{n-1}} T_n$

   bindings made so far: This is a list L of pairs $\langle T_j, t_j \rangle$, where each T_j is one of the variables of the template and t_j is the node that bound its name to T_j when it was visited by the token.

   Procedure:
   (indentation is used to indicate the scope of *then* and *else* clauses)
   if T_i is bound to a term different from t_i
   then return eos-token with act. name $\bar{a}$ to sender S,
     discard token T
   else bind t_i to T_i (i.e., add pair $\langle T_j, t_j \rangle$ to list L)
     if T_i is the last node (T_n) of the template
     then return a reply token (carrying list L
       and activity name $\bar{a}$) to sender S,
       discard token T
     else form a new token T' with template
       $T_{i+1} \xrightarrow{p_{i+1}} \ ... \ \xrightarrow{p_{n-1}} T_n$ and the list L
       replicate T' along all arcs that match p_i;
       the activity names of these tokens will be
       $\bar{a}.1, ..., \bar{a}.p$; record these as pending.
       if p_i points to other clauses
       then for each such clause do
         fetch the clause, form l linear
         clusters as described in Section 3,
         place each on a token and send it to
         nodes that match the leftmost node of
         the cluster, record l new subactivities
         $\bar{a}.[(p+j).1], ..., \bar{a}.[(p+j).l] \quad (1 \leq j \leq k)$.

2. Procedure executed by a node t_i upon receiving a *reply token* R; each such token has the form:

   activity name: $\bar{a}.j$ (where j is the right-most component of the activity name)

   bindings: List L of pairs $\langle T_j, t_j \rangle$ as defined above.

   Procedure:
   if the activity name $\bar{a}.j$ is within $\bar{a}.1, ..., \bar{a}.p$
   then send reply token (with act. name $\bar{a}$) to sender S
   else (i.e. when activity name is greater than $\bar{a}.p$)
     record all bindings (list L) with the activity $\bar{a}.j$.

3. Procedure executed by a node t_i when receiving an *eos-token*; each such token has the form:

   activity name: $\bar{a}.j$ (where j is the right-most component of the activity name)

   Procedure:
   if the activity name $\bar{a}.j$ is within $\bar{a}.1, ..., \bar{a}.p$
   then terminate the activity $\bar{a}.j$
   else mark the corresponding subact. as completed;
     if all subactivities within the act. $\bar{a}.j$ are marked
     then for each combination of bindings
       produce a reply token (carrying that

combination and the act. name $\bar{a}$),
return the token to sender S;
terminate the activity $\bar{a}.j$.
if all activities $\bar{a}.1, ..., \bar{a}.(p+k)$ have been terminated then return eos-token (with act. name $\bar{a}$) to sender S.

6. CONCLUSIONS

The aim of this paper was to present a model of computation which would permit logic programs to be executed on a highly parallel computer architecture. The approach was based on the idea of transforming logic programs into collections of dataflow graphs and graph templates and to let resolution be carried out by asynchronously propagating tokens through the graphs. The main advantage of this approach is a high-degree of potential parallelism, exploitable at the following three levels:

OR-parallelism: If more than one clause is unifiable with a given goal, each may be processed independently by separate sets of tokens injected into the graph.

AND-parallelism: Clusters, i.e., groups of literals within a clause which do not share free variables, may be processed concurrently by separate tokens. Furthermore, when clusters are transformed into trees, each branch the the tree may be processed concurrently, thus further increasing potential parallelism.

Simultaneous execution of programs: By using different activity name sets, many programs, in particular database queries, may be processed concurrently thus further increasing the throughput of the system.

In terms of the necessary architectural support required, the proposed model bears a strong similarity to a general dataflow system, primarily due to the underlying data-driven principles of operation. Hence this paper offers further support for the claim that dataflow machines could be extended to inference machines through the use of logic programming /Aiso 1981/.

REFERENCES

Arvind, K P Gostelow, W Plouffe, An Asynchronous Programming Language and Computing Machine, Advances in Computing Science and Technology (ed. Ray Yeh), Prentice-Hall publ. 1978

A V Aho, J E Hopcroft, J D Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publ., 1976

Arvind, R A Iannucci, A Critique of Multiprocessing von Neumann Style, Proc. 10th Annual Int'l Symposium on Computer Architecture, SIGARCH Vol.11, No.3, June 1983

H Aiso, Fifth Generation Computer Architecture, Proc. Int'l Conf. Fifth Generation Computer Systems, Oct. 1981

L Bic, Execution of Logic Programs on a Dataflow Architecture, Proc. 11th Annual Int'l Symp. on Computer Architecture, Ann Arbor, Mich., 1984

COMPUTER, Special Issue on Dataflow Systems, 15,2, Feb. 1982

V Dahl, On Database Systems Development Through Logic, ACM TODS, Vol.7, No.1, March 82

A Deliyanni, R A Kowalski, Logic and Semantic Networks, CACM 22,2, March 79

H Gallaire, J Minker (Eds.), *Logic and Data Bases*, Plenum, N.Y. 1978

J K Minker, An Experimental Relational Database System Based on Logic, in *Logic and Databases* (H Gallair, J K Minker, Eds.), Plenum Pub., 1978

P C Treleaven, T R Brownbridge, R C Hopkins, Data-Driven and Demand-Driven Computer Architecture, ACM Computing Surveys, 14,1, March 1982

D Warren, Efficient Processing of Interactive Relational Database Queries Expressed in Logic, Proc. 7th Int'l Conf. VLDB, Cannes, 1981