UC Irvine
ICS Technical Reports

Title
A data-driven model for parallel interpretation of logic programms [sic]

Permalink
https://escholarship.org/uc/item/058942fd

Author
Bic, Lubomir

Publication Date
1984

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no. 217

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

A Data-Driven Model for Parallel Interpretation of Logic Programms†

by

Lubomir Bic

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

January 1984

Technical Report 217

†This work was supported by the NSF Grant MCS-8117516: The UCI Dataflow Project.

Abstract

The main objective of this paper is to present a model of computation which permits logic programs to be executed on a highly-parallel computer architecture. It demonstrates how logic programs may be converted into collections of dataflow graphs in which resolution is viewed as a process of finding matches between certain graph templates and portions of the dataflow graphs. This graph fitting process is carried out by tokens propagating asynchronously through the dataflow graph; thus computation is entirely data-driven, without the need for any centralized control. It is shown that at the implementation level the proposed model is very similar to a general dataflow system and hence a dataflow architecture could easily be extended to support the proposed model.

CR Categories: C.1.3 [Processor Architectures]: Other Architecture Styles – *Data-flow Architectures*; F.1.2 [Computation by Abstract Devices]: Modes of Computation – *Parallelism*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic – *Logic Programming*

Additional Key Words and Phrases: data-driven computation, parallel logic programming

1. Introduction

Logic programming has been recognized as an effective approach to representing information and describing problems that can be solved by a computer using logical inferences /Kow82/. Furthermore, logic programs do not presuppose a von Neumann architecture and are, therefore, inherently well suited to parallel computations. Several recent research projects have investigated this potential and schemes which permit parallel execution of logic programs have been proposed /Bow82, CoKi83, Con83, EKM82/. While all these approaches differ in many fundamental aspects, the principle common to all of them is to view each predicate $p(x_1, ..., x_n)$ as a node in an AND/OR-tree of possible solutions. Execution then may proceed concurrently along the OR-branches while AND-branches are subject to restrictions resulting from free variables shared among predicates.

In /DeKo79/ Deliani and Kowalski have shown how logic programs may be viewed as an (extended) form of semantic networks. In this approach, each predicate is assumed to be binary, ie. of the form $p(t_1, t_2)$, an thus may be represented as a labeled arc p interconnecting the two nodes t_1 and t_2. Such a representation permits resolution to be viewed as a special pattern matching problem in which networks corresponding to individual clauses are fitted into portions of other graphs.

In the approach presented in this paper we adopt a similar point of view by representing logic programs as collections of graphs and graph templates. The main distinction, however, is the way the pattern matching is carried out. This is based on the principles of asynchronous, data-driven computations /COM82, Den75, TBH82/ in which a graph is not merely a passive representation of a program stored in memory; rather each node is an active agent, supported by an independent processing element, and hence is capable of communicating with other nodes via value tokens traveling asynchronously along the graph arcs. Finding a

given pattern (a graph template) in such a **dataflow graph** is then accomplished by an asynchronous propagation of tokens through the graph. The pattern to be fitted is placed on one or more tokens which are injected into specific nodes of the graph. Each token is replicated into all possible directions, thus searching for the given pattern. Since many processing elements may be engaged in the replication and forwarding of individual tokens, a high degree of parallelism can be achieved.

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 /DeKo79/. † 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 representation systems /Dah82, GaMi78, Min78, War81/, as opposed to mathematically oriented computations.

Overview of paper: After briefly surveying the basic principles of logic programming, we will demonstrate how logic programs are converted into a network form (Section 2). Then the principles of solving goals using token propagation will be introduced (Section 3). In Section 4 we will introduce a linear form of a goal which is easier to search for in the dataflow graph, and a method for transforming arbitrary goals into linear sequences will be presented. Section 5 then defines the operational semantics of the model by specifying the exact procedures executed by each node of the dataflow graph when a token is received. Finally, the architectural requirements for supporting the proposed model will be addressed in Section 6.

† It can be shown that any n-ary relation may be transformed into a sequence of binary relations.

2. Representing Logic Programs as Networks

We assume the reader to be familiar with the principles of logic programming; the following paragraphs survey only briefly the fundamental concepts and introduce an example to be used in subsequent sections.

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.

p_0 is called the **head** or **conclusion**, and p_1 through p_n form the **body** or **conditions** of the clause. A clause with an empty set of conditions is called an **assertion** and is used to represent explicit facts. In Figure 1 a sample program is presented which records the relationships 'mother' and 'father' among individuals as a sequence of assertions (lines 1-5). †

A clause which contains both a head and a body can be interpreted as recording implicit information. For example, the program in Figure 1 records that a 'parent' relationship between two individuals X and Y holds if they are related via the relationships 'mother' (line 6) or 'father' (line 7). Similarly, a 'grandparent' relationship between X and Y holds if two 'parent' relationship, one between X and Z and another between Z and Y, exist (line 8).

A clause with an empty conclusion is interpreted as a request or **goal** which the system tries to solve by unifying it with the head of some other clause. In Figure 1, line 9 contains a goal paraphrased as 'Who are the grandparents of bill?'. The system will try to unify this goal with some other clause, in our example, the

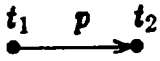
† Throughout this paper, lower case letters are used to denote constants while capitals are used to denote free variables.

clause on line 8. The variable X is bound to the constant 'bill' which generates two new goals, parent(bill,Z) and parent(Z,Y), both of which must be solved in order to satisfy the original goal. This process is repeated until one (or more) solutions are found, or no further unifications are possible.

(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

A literal, as defined above, consists of a predicate name and a sequence of arguments. 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. (As will be discussed later, the arrow heads do not prescribe the direction in which tokens may flow through the graph.)

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.

(**Notation:** 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.)

We will distinguish two types of graphs: An **assertion graph** is constructed from the sequence of all assertions containing only ground terms (ie. 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 occurences 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.

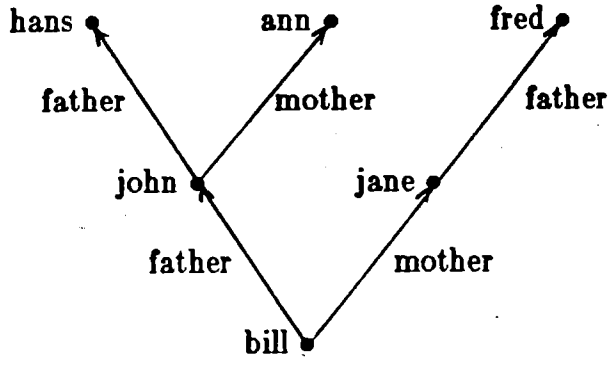


Figure 2

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.

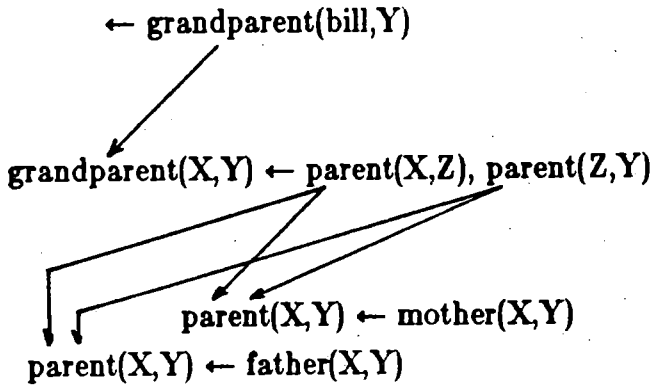


Figure 3

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.)

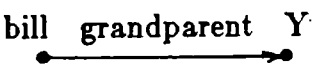


Figure 4a

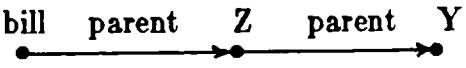


Figure 4b

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 seqences. For example, the graph template of Figure 4b, would be transcribed as bill $\xrightarrow{parent}$ Z $\xrightarrow{parent}$ Y.)

3. Solving Goals Using Token Propagation

3.1. 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 occurence 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 pattern.

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. † 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.

3.2. 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 $p_1(T_1, T_2)$, $p_2(T_2, T_3)$, ..., $p_{n-1}(T_{n-1}, T_n)$. Such a sequence corresponds to the following graph template $T_1 \xrightarrow{p_1} T_2 \xrightarrow{p_2} \ldots \; T_{n-1} \xrightarrow{p_{n-1}} T_n$ and shall be referred to as **linear form**. Note that the first term of each literal matches the second term of the preceeding literal which implies the following important property: Each time a literal p_i is solved, it binds the term T_{i+1}, which is the first term of the next literal p_{i+1}. Hence all literals of the linear sequence, except the first, will have at least one term bound when the sequence is processed from left to right.

Assuming that none of the subgoals p_i has a pointer to other clauses, the processing of the linear sequence then corresponds to the following *graph fitting* problem:

† 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 drasticly reduce the number of injected tokens.

determine possible bindings for all terms T_i 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. (As was the case with individual subgoals, only one such node t_1 will exist if T_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 t_2 to T_2 and, if successful, will continue the propagation of the token along all arcs matching the name p_2. An analogous step is performed by any node 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 (ie., T_i 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 path, is returned by the node t_i to the sender of the received token.

(b) The last node T_n 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).

3.3. 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 5, where p is the goal to be solved.

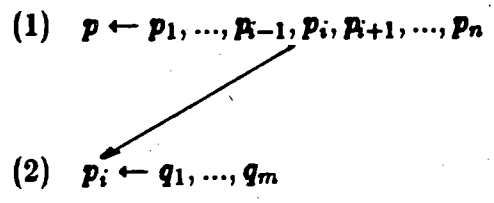


Figure 5

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

(a) The original sequence $p_1, ..., p_n$, and

(b) The sequence $p_1, ..., p_{i-1}, q_1, ..., q_m, p_{i+1}, ..., p_n$, 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 $p_1, ..., p_n$ is placed on a token and starts expanding from an injection node t_1 into all possible directions as described in Section 3.2. 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 $q_1, ..., q_m$ and a copy of the yet unused portion of the current sequence $p_{i+1}, ..., p_n$, 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

$p_1, ..., p_{i-1}, p_i, p_{i+1}, ..., p_n$ and $p_1, ..., p_{i-1}, q_1, ..., q_m, p_{i+1}, ..., p_n$, respectively.

The scheme described thus far is complete in that it finds all solutions to a given goal by applying the resolution principle to all relevant clauses constituting the program. It requires, however, that all clauses be in the linear form as defined in Section 3.2. Furthermore, the leftmost term T_1 should be bound, if possible, in order to reduce the number of injection points to one. The next section is devoted to the problem of transforming arbitrary sequences of literals into such linear sequences.

4. Transformation of Clauses into Linear Sequences

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. We will take the following approach: First the original sequence of literals is divided 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 the number of clusters comprising a sequence $p_1, ..., p_n$ determines the number of activities that can be started concurrently for the given sequence. Such activities are AND-parallel, ie., a solution to the sequence $p_1, ..., p_n$ exists only if *each* cluster yields at least one solution.

Since clusters may be solved independently, they will be carried (and fitted into the assertion graph) by separate sets of tokens. Hence we can concentrate on the problem of transforming clusters (as opposed to arbitrary sequences of literals) into

linear forms. This transformation is based on the idea of finding an Euler path † through the corresponding graph template, (ie. a path which traverses all arcs exactly once). Furthermore, the transformation will attempt to construct the linear sequence such that the leftmost node is a bound variable. This will reduce the number of injection points to one as discussed in Section 3.1. The only time this will not be possible is when none of the variables constituting the cluster is bound.

We can distinguish the following three cases when transforming a cluster into such a linear sequence:

1. All nodes of the cluster have an even local degree, (where local degree is defined as the number of arcs connected to that node.) In this case an Euler path is guaranteed to exist /Mar71/; traversing this path then yields a linear sequence comprising all literals of the cluster. Furthermore, the Euler path is circular, hence we can begin the traversal at any point within the cluster. If at least one node of the cluster is a bound variable we can choose it as the starting point thus constructing the desired linear sequence in which the leftmost node is bound. Figure 6a shows the graph template corresponding to the sequence $p_1(A, b)$, $p_2(b, C)$, $p_3(A, d)$, $p_4(C, d)$. It contains a circular Euler path and hence we can construct the following linear sequence with b as the leftmost node:
$b \xrightarrow{p_2} C \xrightarrow{p_4} d \xleftarrow{p_3} A \xrightarrow{p_1} b.$
(Note that the arc p_3 will be traversed against the arrow head.)

2. There exist two nodes with odd local degree. In this case an Euler path connecting these two nodes is guaranteed to exist /Mar71/. Traversing this path yields a linear sequence comprising all literals of the cluster. If one of the nodes with odd local degree is bound, the sequence is in the desired form. Otherwise the following modification is performed: The path is broken at one of the bound

† Also referred to as Eulerian Chain in the literature.

nodes along the path and the portion to the left of that node is reversed. Thus we obtain a sequence of two paths, each beginning with the same bound node. Note that the only time this transformation is not possible is when the entire cluster consists of only free variables. Figure 6b illustrates the situation. An Euler path connects the two odd local degree nodes A and E, both of which are free variables. By breaking the path at one of the bound nodes, b, we obtain the following two paths:

$$b \xleftarrow{p_1} A \xrightarrow{p_5} E \Longrightarrow b \xrightarrow{p_2} C \xrightarrow{p_4} d \xleftarrow{p_3} A$$

The double arc connecting the two paths indicates that this transition is not the traversal of an arc, rather it denotes a 'transfer' or 'jump' to b. Since the target node b is bound, such a jump is analogous to injecting a token into the node b. Hence injecting a token and performing a jump may be implemented using the same mechanism for routing a token to a given node, as will be discussed further in Section 6.

Note: When traversing an Euler path it may be necessary to visit some nodes more than once which results in multiple occurences of terms within the linear sequence. If such a term is a free variable, all occurences will have to be bound to the same term during execution. For example, in the above sequence the variable A occurs twice. When the token carrying this template reaches a node a which binds itself to the first occurence of A, all other occurences of A within that sequence must be bound to a as well before forwarding the token to other nodes.

3. The cluster contains more than two nodes with odd local degree. Since no Euler path exists in this case, we must find several disjoint paths and connect these using the jump-construct introduced in point 2 above. It can be shown that n/2 edge-disjoint paths are necessary to travers a cluster, where n is the

number of nodes with odd local degree /Mar71/. Using the same approach as in 2, each of these paths may be constructed such that it begins with a bound node, unless all nodes along that path are free variables. Note that all paths, except the first, will have at least one node bound. This is because each path has an intersection X with at least one other path (otherwise the cluster would not be a connected graph); when a path is traversed, all of its nodes are bound and hence the path to be traversed next will have at least one bound node – the intersection node X. Figure 6c shows a cluster which can be traversed in three paths connected via the jump-construct as follows:

$$b \xleftarrow{p_1} A \xrightarrow{p_5} E \Longrightarrow b \xrightarrow{p_2} C \xrightarrow{p_4} d \xleftarrow{p_3} A \Longrightarrow C \xrightarrow{p_6} F$$

The first two have been obtained by breaking a single path at the bound node b as in the previous case. Note that the third path begins with a free variable C. The same variable C, however, appears on the second path, and hence will be bound before the jump-construct to C is reached.

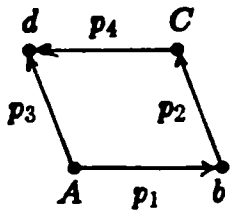


Figure 6a

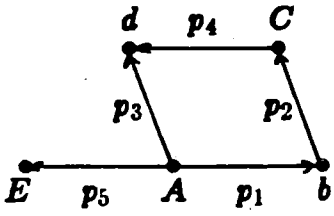


Figure 6b

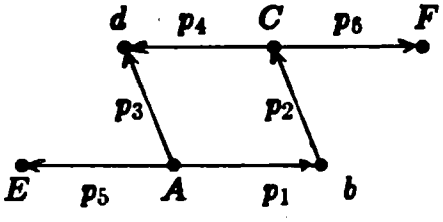


Figure 6c

From the above it follows that any sequence of literals may be converted into one or more linear sequences, some of which may be connected via the jump-construct. Furthermore, the leftmost term of any linear sequence will be a free variable only if no bound variable occured in the entire cluster.

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 /AGP78/: 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 $T_i \xrightarrow{p_i} T_{i+1} \xrightarrow{p_{i+1}} \ldots \xrightarrow{p_{n-1}} T_n$ and the activity name $a_1.a_2. \ldots .a_i$, which we shall abbreviate as $\bar{a}$. As described in Section 3.2, 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 3.3. These activities will be assigned the names $\bar{a}.(p+1), \bar{a}.(p+2), \ldots, \bar{a}.(p+k)$, 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 $\bar{a}$:

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

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 /AGP78/. 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 3. 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: a

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

   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 activity name $\bar{a}$ to sender S,
  discard token T
 else bind t_i to T_i (appending the pair $\langle T_j, t_j \rangle$ to the list L)
  if T_i is the last node (T_n) of the template
  then return a reply token (carrying the list L and the activity
   name $\bar{a}$) to sender S,
   discard token T
  else form a new token T' with graph template $T_{i+1} \xrightarrow{p_{i+1}} \ldots \xrightarrow{p_{n-1}} T_n$
   and the list of bindings L (including the new pair $\langle T_j, t_j \rangle$),
   replicate T' along all arcs that match p_i; the activity names
   of these tokens will be $\bar{a}.1, \ldots, \bar{a}.p$ (see section 5.1),
   record the new activity names as pending activities;
   if p_i points to other clauses
   then for each such clause do
    fetch the clause,
    form l linear clusters as described in Section 4,
    place each cluster on a token and send it to the node that
    matches the leftmost node of the cluster,
    record l new subactivities $\bar{a}.[(p+j).1], \ldots, \bar{a}.[(p+j).l]$,
    (where $1 \leq j \leq k$).

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 $\bar{a}.1, \ldots, \bar{a}.p$
   then send reply token (with activity name $\bar{a}$) to sender S;
   else (ie. when the activity name is within $\bar{a}.(p+1), \ldots, \bar{a}.(p+k)$)
    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 subactivity as completed;
  if all subactivities within the activity $\bar{a}.j$ are marked
  then for each combination of bindings (one from each subactivity)
    produce a reply token (carrying that combination of bindings
    and the activity 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 activity name $\bar{a}$) to sender S.

6. Architectural Issues

In this section we examine the requirements that must be satisfied by a computer architecture in order to exploit the potential parallelism offered by the proposed model.

We consider an architecture consisting of a large number of asynchronously operating processing elements (PEs), each equiped with a certain amount of local memory. The architecture must satisfy the following fundamental requirements:

1. The assertion graph must be mapped onto the collection of PEs during execution such that each node can receive, process, and emit tokens. This can be accomplished by using a global mapping function which, given a node of the assertion graph, yields a number from 1 to n, where n is the number of PEs. The node is then assigned to the PE corresponding to the selected number. Hence each PE is 'multiplexed' among all nodes mapped onto that PE. This requirements is analogous to the problem of mapping a general dataflow graph onto a parallel architecture and a number of possible schemes have been proposed and investigated /GoTh80/.

2. Nodes must be able to exchange tokens with one another along the (logical) graph arcs. This is accomplished by associating with each node t a list of all

those nodes to which t is (logically) connected. Sending a token along such an arc then involves calculating the PE number of the destination node and letting the token propagate via neighboring PEs to its final destination. Similar to requirement 1 above, any dataflow architecture must be capable of supporting such an exchange of tokens among graph nodes and hence the same principles apply to the system presented in this paper.

3. A token carrying a graph template may contain pointers to other clauses in the goal structure. General dataflow systems are capable of solving an analogous problem: tokens must carry pointers to large data structures kept in a common memory and shared among different tokens /ArTh80/. In our case the situation is further simplified by the fact that the goal structure, while being shared, need not be modified during execution.

4. It must be possible to inject a token into any node of the assertion graph. This includes the injection of initial tokens from outside of the system, as well as the implementation of the jump-construct introduced in Section 4, which requires a token to travel to some other node of the graph. Both cases are analogous to the problem of sending a token from one node to another along a logical arc (requirement 2 above) and may be solved using the same mechanisms: given the destination node name, the corresponding PE holding that node may be determined by applying the global mapping function (discussed under requirement 1 above) to that node name. The token is then routed to that PE via the physical connections of the architecture.

From the above discussion it follows that the fundamental architectural requirements of the proposed model are already satisfied by any general dataflow architecture and that only minor modifications would be necessary to adapt such an architecture to support the proposed logic programming model.

7. 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, ie., groups of literals within a clause which do not share free variables, may be processed concurrently by separate tokens.

Simultaneous execution of independent programs: By using different activity name sets, many programs, eg. 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 /Ais81/.

References

/AGP78/ 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

/ArTh80/ Arvind, R E Thomas, I-Structures: An Efficient Data Type for Functional Languages, Tech. Rep. TM-178, Lab. for Computer Science, MIT, Cambridge, Mass., Sept. 1980

/Ais81/ H Aiso, Fifth Generation Computer Architecture, Proc. Int'l Conf. Fifth Generation Computer Systems, Oct. 1981

/Bow82/ K A Bowen, Concurrent Execution of Logic, Proc. 1st Int'l Logic Programming Conf., Marseilles, Sept. 82

/CoKi83/ J Conery, D Kibler, AND Parallelism in Logic Programs, IJCAI 1983

/COM82/ COMPUTER, Special Issue on Dataflow Systems, 15,2, Feb. 1982

/Con83/ J Conery, The AND/OR Model for Parallel Interpretation of Logic Programs, PhD Thesis, Dept. of ICS, Univ. of California, Irvine, 1983

/Dah82/ V Dahl, On Database Systems Development Through Logic, ACM TODS, Vol.7, No.1, March 82

/DeKo79/ A Deliyanni, R A Kowalski, Logic and Semantic Networks, CACM 22,2, March 79

/Den75/ J B Dennis, First Version of a Dataflow Procedure Language, Mac Tech. Memorandum 61, M.I.T., Cambridge, 1975

/EKM82/ N Eisinger, S Kasif, J Minker, Logic Programming: A Parallel Approach, Proc. First Int'l Logic Programming Conf., Faculte des Sciences de Luminy, Marseille, Sept. 1982

/GaMi78/ H Gallaire, J Minker (Eds.), *Logic and Data Bases*, Plenum, N.Y. 1978

/GoTh80/ K P Gostelow, R E Thomas, Performance of a Simulated Dataflow Computer, IEEE TC, Vol. C-29,10, Oct. 1980

/Kow82/ R A Kowalski, Logic Programming for the Fifth Generation, Proc. Int'l Conf. on Fifth Generation Systems, SPL Internationals 1982

/Mar71/ C W Marshall, *Applied Graph Theory*, Wiley-Interscience, 1971

/Min78/ J K Minker, An Experimental Relational Database System Based on Logic, in *Logic and Databases* (H Gallair, J K Minker, Eds.), Plenum Pub., 1978

/TBH82/ P C Treleaven, T R Brownbridge, R C Hopkins, Data-Driven and Demand-Driven Computer Architecture, ACM Computing Surveys, 14,1, March 1982

/War81/ D Warren, Efficient Processing of Interactive Relational Database Queries Expressed in Logic, Proc. 7th Int'l Conf. VLDB, Cannes, 1981