International Journal of Parallel Programming, Vol. 15, No. 1, 1986

Control and Data Driven Execution of Logic Programs: A Comparison

Simon Kasif[1]

Received October 1985; revised May 1986

In this paper we examine two methods for controlling the execution of parallel logic programs. Specifically, we compare control driven execution of PRISM programs vs. data driven execution of Concurrent Prolog. Given a PRISM program we present several automatic transformations for deriving a Concurrent Prolog program whose execution is isomorphic to the original program. Although in many specific cases we may be able to write very natural specifications based on read-only variable and commit constructs, in general it is difficult to simulate control flow naturally using a transformation based on these constructs. Since control flow primitives are shown to have a simple and efficient implementation it seems that both data-flow and control-flow mechanisms are desirable for a general purpose parallel logic programming language. Subsequently, we propose a simple low level language to implement both PRISM nested control flow and Concurrent Prolog read-only variables. The idea is to convert the control/data dependencies into simple event scripts and then use existing methods to implement these scripts efficiently. Finally, we introduce a data structure that supports an efficient implementation of PRISM nested control flow.

KEY WORDS: Logic Programming; parallelism; data flow; graphs.

1. INTRODUCTION

1.1. Motivation

In conventional programming systems, control is embedded in the program and is difficult to modify without affecting the logic of the

[1] Department of Electrical Engineering and Computer Science, The Johns Hopkins University, Baltimore, Maryland 21218.

algorithm. In a logic programming system, control decisions can be made by the interpreter dynamically depending on the state of the execution, the local context and availability of resources. Thus, a problem solving system specified using logic may incorporate facilities for dynamic decomposition of the problem, recognizing multiple data streams and distributed control. Specifically, several processors may solve the problem simultaneously and then cooperate with the rest of the system to obtain the complete solution.

A primary issue in achieving an effective parallel logic programming system is developing a formalism that will allow:

1. Control of execution of parallel logic programs by specifying a partial order on the execution of program statements. It is desirable that the specification is natural in the sense that sequencing constraints are expressed directly.
2. Automatic detection of concurrency in logic programs. By this we mean that the formalism must allow us to exploit the intrinsic concurrency of the computation, even in cases where it is impossible to detect *a priori* that two tasks may be executed in parallel.
3. Analysis of the relative effectiveness of control constructs in terms of their expressiveness and efficiency.

Control issues in programming languages belong to two categories: execution (process) control and communication control. These problems are sometimes very difficult to treat separately. In this paper however, we shall emphasize the process control component of parallel logic programs. Communication control is studied in Ref. 1. We investigate and compare two prevalent facilities for execution control of parallel logic programs: control and data driven sequencing. Our objective is to investigate the candidate control constructs in terms of their efficiency, difficulty of implementation and closeness to the spirit of logic programming.

We shall assume that the reader is familiar with the basic concepts of logic programming (see Ref. 2). Throughout the paper we shall assume a top-down execution of logic programs. The programming convention we shall use is that variables are denoted by names starting with lower case letters. To simplify the notation we shall avoid formal rigor, and for the purpose of this paper assume the procedural interpretation of logic programs. For completeness, we begin with some remarks pertaining to the elements of the reduction process that need to be controlled.

1.2. Why Sequence Literals

There are two reasons for introducing partial order on the execution of literals in a clause (literal sequencing). Firstly, efficiency considerations may dictate a particular order on the execution of literals. A typical example is the logic program for the computation of Ancestor (given by the following) where the literal F in the second clause should be executed before the literal A.

$$A(x, y) \leftarrow F(x, y).$$
$$A(x, z) \leftarrow A(y, z), F(x, y).$$
$$F(A, B).$$
$$F(B, C).$$
$$F(C, D).$$
$$F(D, E).$$
$$F(E, F).$$

A rather different need for literal sequencing arises whenever logic programming is used to write essentially procedural programs such as programs that manipulate shared data-structures, perform I/O operations or have side-effects. Several self-explanatory examples that belong to this category are given by the following.

GET(gift, box) ← TAKE_OUT(gift, box), THROW_OUT(box)

DO-LAUNDRY(x, x'') ← WASH(x, x'), DRY(x', x'')

Executing procedures GET or DO_LAUNDRY in any other order except left to right makes little or no sense. Consider the following program that tests the membership of an integer in a set of integers in a binary tree representation.

Binary(a, Tree(t1, root, t2)) ← Equal(a, root).

Binary(a, Tree(t1, root, t2)) ← Less(a, root), Binary(a, t1).

Binary(a, Tree(t1, root, t2)) ← Greater(a, root), Binary(a, t2).

Binary Search Program in Logic

An important aspect of the semantics of the Binary Search program is determined by its procedural interpretation, and its only reasonable interpretation is Prolog-like (left-to-right). The number of examples where one uses procedural (imperative) language to specify a partially ordered sequence of events is numerous. In Ref. 3, we describe a general transformation of flowcharts to logic programs. Consider the Algol-style program to exchange the values of two memory locations.

```
1: tmp = x;
2: x = y;
3: y = tmp;
```

Using the transformation given in Ref. 10 we derive the following Prolog program:

$$P1(x, \text{tmp}, y) \leftarrow P2(x, x, y).$$
$$P2(x, \text{tmp}, y) \leftarrow P3(y, \text{tmp}, y).$$
$$P3(x, \text{tmp}, y) \leftarrow P4(x, \text{tmp}, \text{tmp}).$$

It is not easy to associate a meaningful declarative semantics with the Prolog program above. Its procedural semantics as a top down goal reduction is similar to the operational semantics of the Algol program. In many cases such as robot programming languages, the sequencing and synchronization of activities are the central problem in the specification. In these cases it is essential that control be an integral part of the algorithm specification.

1.3. Why Sequence Clauses

The sequencing of the execution of clauses has two distinct rationale illustrated by the following two examples.

The first case arises when it is more likely to find a solution faster using one clause then the other. This was indeed the case in the Ancestor program above. Many instances of these phenomena may be found in the domain of heuristic programming, where a heuristic function dictates the selection of the best node to expand.

The second and perhaps more important case arises when one clause should be tried only if the other had failed. A typical example is a recursive program specified by a base case and an inductive case where the inductive case should be executed iff the base case fails. In principle, it is possible to associate a set of literals with the inductive case that will ensure that this case is executed iff the base case failed. However, it is both inelegant and inefficient to perform the test for the base case twice, when the same effect may be attained by explicit sequencing of the two alternatives. A slightly less intuitive example is given by the following.

$\leftarrow$MAXR(0, max).

MAXR(x, max) $\leftarrow$ $R(y)$, Greater(y, x), MAXR(y, max).

MAXR(x, x).

Finding The Maximum of a Relation

This program finds the maximum element of a relation R whose length is unknown that consists of a set of positive integers i, $0 \leqslant i$. It is easy to see that MAXR works iff the second clause is executed iff the first one failed. The specification of the MAXR problem without resorting to sequencing constraints is rather complicated.

These observations have been recognized by the logic programming community since the emergence of the discipline and raised many interesting solutions. In this paper we shall focus on the literal sequencing component of parallel logic programs. Specifically, we shall investigate and compare control and data driven execution of logic programs.

2. MEANS FOR ACHIEVING CONTROL IN PARALLEL LOGIC PROGRAMS

Roughly speaking, control of existing logic programming languages belongs to two categories: control-flow driven and data-flow driven. As the name control flow driven computation suggests, the computation of a program is guided by some select procedure that at each point of the computation determines the next set of executable instructions. Once the set of executable instructions is defined, it is immediately executed. Two examples of logic control driven logic programming languages are Prolog and PRISM. Prolog is perhaps the best known logic programming language and was designed to run efficiently on a sequential computer. PRISM(4) is a parallel system developed for a highly parallel multiprocessor ZMOB.(5)

Data flow driven systems select ALL the statements in a program for possible execution. However, they add an EXAMINE step to check for the firing condition of each statement. Typically, the firing criterion is the availability of data that serve as the arguments for the statement. Logic programming languages that incorporate data-driven execution are Concurrent Prolog,(6) Parlog,(7) Epilog,(8) and GHC,(9) IC-Prolog(10). Parlog and Epilog have control flow primitives as well.

In the following sections we shall examine the control facilities available in current logic programming systems and compare their relative efficiency and applicability.

3. PRISM

PRISM[4] is a logic programming language that provides the user with control facilities to specify a partial order on the execution of literals and clauses in a program. This partial order expresses the dependencies among the subgoals within a goal (a procedure body may be considered to be a goal), or within alternative procedures for solving the same goal. Since PRISM was originally designed as a problem solving system tailored for Artificial Intelligence applications it has many control facilities that are irrelevant to the comparisons performed in this paper. We shall focus our attention on the primitives that are necessary for the analysis that follows (see Refs. 4 and 11 for a full description).

A convenient way to represent PRISM structured control specification is by using the notion of a control group defined recursively as follows:

1. A control group is either an S-group or a P-group, where S and P denote sequential and parallel execution respectively.
2. A P-group is a set of elements that are either atoms or control groups.
3. An S-group is a set of elements that are either atoms or control groups.

We use parenthesis ( ) and brackets [ ] to specify sequential execution concurrent execution respectively. For instance the $[P, (Q.R), S]$ represents a P-group that consists of the atom P, the S-group (Q, R) and the atom S. The semantics of this specification is that the elements of a P-group may be executed in parallel, whereas the elements of an S-group must be executed sequentially.

The clause selection notation is symmetric with the notation for atom selection. A simple schematic description of PRISM interpreter in PRISM is given by.

```
[
PRISM(P) ← (Clause(P ← Body), PRISM(Body)).
PRISM([P, Q]) ← [PRISM(P), PRISM(Q)].
PRISM((P, Q)) ← (PRISM(P), PRISM(Q)).
]
```

A PRISM Interpreter in PRISM

Whenever the control specification is not given PRISM assumes default ordering determined at the initiation of the system. We shall assume parallel sequencing as a default.

PRISM has been fully implemented and the preliminary experimental results are reported in Ref. 12.

4. DATA DRIVEN LOGIC PROGRAMMING

Consider a logic program for the computation of factorial and its data flow graph as given in Fig. 1. It is evident that the data flow graph provides valuable control information that could be useful in sequencing the literals. Several early ideas based on this theme were originally proposed in Refs. 10, 13 and 14. (References on Data driven computation go back to 1970.) These ideas spurred a significant amount of work and indeed proved fertile since today there exist a large class of languages and models based on data driven execution of logic programs. Since Concurrent Prolog is widely known, we shall restrict our discussion to the constructs found in this language.

4.1. Concurrent Prolog

The fundamental difference between Prolog and Concurrent Prolog[6] is based on a different operational interpretation of logic programs found in each. In Prolog the interpretation of

$$P \leftarrow P_1, \ldots, P_n$$
$$P \leftarrow Q_1, \ldots, Q_m$$

is to solve P solve P_1, then P_2 and so on. If the first clause that matches P fails the second one is tried. In Concurrent Prolog to solve a goal $\leftarrow P$ the system tries (in parallel) to match the goal against the heads of both clauses. A commitment is then made to the clause whose head matches the goal and whose guard succeeded (see text below) first irrespective of whether there were additional matches. Hence, only one clause is being effectively executed (no OR-parallelism). The system proceeds by a parallel invocation of all the processes in that procedure body. As a result, conjunctive atoms in Concurrent Prolog are interpreted as communicating

Fact(0,1).

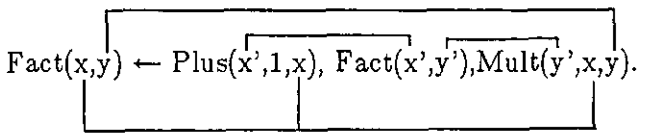

Fig. 1. A data flow graph for a factorial program.

parallel processes. There are two constructs that allow synchronization o Concurrent Prolog programs: the commit operator and read-onl variables. Commit used to specify the Dijkstra guarded command seman tics. For example the clause:

$$P \leftarrow G \mid Q$$

specifies that process P' is reduced to process Q if P' is unifiable with P and the guard G terminates successfully. The use of this construct in the context of logic programming was originally in Ref. 15. A read-only variable denoted syntactically by $x?$, allows the programmer to suspend the execution of an atom in which it occurs. The atom will be activated when the variable is bound to a nonvariable term during the execution of another atom where x also occurs. This concept is a generalization of data flow sequencing in functional languages.

In our opinion Concurrent Prolog is an elegant language, and it meets many of the design goals of its creators. However, (similarly to Prolog) due to the loss of the clean model theoretic semantics of logic programs, a Concurrent Prolog program may no longer be seen as a set of logic formulae that specify the semantics of the problem. This is not a criticism since the designers of these languages made a conscious decision to prove the effectiveness of the logic programming paradigm by sacrificing logical completeness in favor of efficiency. As a result, in many cases the semantics of Concurrent Prolog/Parlog programs may not be understood only by studying the declarative semantics of the clauses. This is due to the fact that both commit and read-only variable may cause incompleteness.

4.2. Communication and Synchronization

Communication in data-driven languages is based on the principle of suspending a consumer until all its input arguments have been bound. Concurrent Prolog offers the most flexible communication facility in its group: read-only variable.

A read-only variable, distinguished syntactically by associating a "?" on its right is unifiable with a term y iff y is a variable. If the unification of a literal that contains a read-only variable $x?$ with a procedure head fails due to the read-only annotation the execution of this literal is suspended until the variable x is bound to a nonvariable term. Read-only variable were introduced in IC-Prolog[10] and are also facilitated in Epilog.[8]

There are several problems with the read-only variable construct as the only synchronization facility (see Refs. 1, 16, and 17. A key problem arises due to the fact that there is no way to communicate to a process except for explicit variable sharing. Thus, in order to define more elaborate control constructs, state variables must be introduced that affect the declarative semantics of the program. We shall elaborate on this point in Section 5.

5. COMPARISON

In this section we shall study the utility of data flow and control flow sequencing in the context of logic programming. It is well recognized that data flow synchronization is applicable in the evaluation of functional languages and arithmetic expressions, whereas control flow is superior in the domain of general purpose programs that manipulate shared complex data structures and perform many I/O operations.[8] Since logic programming is a general framework that allows both kinds of programming styles, the question which is the "right" control is inevitable.

This problem may be answered by examining the cost of simulating one method with the other. If the transformation from a programming style *A* to a style *B* is simple and may be supported efficiently, then this will serve as a conclusive evidence in favor of *B*.

5.1. Control Flow to Data Flow Translation

In this section we shall investigate the methods of transforming PRISM programs into Concurrent Prolog programs. This problem has been independently studied in the following Refs. 1, 19, and 20. Since we would like to focus on the sequencing of atoms (as supposed to sequencing the execution of clauses), we informally define the notion of a deterministic PRISM program as follows. A PRISM program Pr is deterministic if it satisfies either of following conditions.

C1. All procedures in Pr have only one definition.

C2. If a procedure *P* in Pr has several definitions, then the procedures heads in the alternative definitions of the procedure *P* do not unify.

C3. If a procedure *P* in Pr has two or more definitions whose heads unify, then all these matching definitions must have a commit in the body of the procedure definition. When the first *k*-atoms in the matching definitions are mutually exclusive e.g., Less(x, 5) and Greater(x, 5), we omit the commit operator in PRISM clauses.

These conditions guarantee that the execution of the PRISM program may be simulated correctly in Concurrent Prolog. Given a PRISM logic program with a partial order defined on the literals we derive an equivalent

Concurrent Prolog program. The two programs should be equivalent in the following sense. If we have enough computing resources to execute all the executable nodes then at each point of the execution the sets of executable nodes in the two programs are equivalent. To simplify the presentation we shall appeal to the reader's intuition and shall not give a rigorous definition of such an equivalence.

5.2. Transformation Using Commit

Consider the PRISM program that performs a binary search given by the following.

Binary(a, Tree($t1$, root, $t2$)) ← Equal(a, root).

Binary(a, Tree($t1$, root, $t2$)) ← (Less(a, root), Binary(a, $t1$)).

Binary(a, Tree($t1$, root, $t2$)) ← (Greater(a, root), Binary(a, $t2$)).

A Binary Search Program In PRISM

Surely, in this simple case the program may be replaced with the following Concurrent Prolog program.

Binary(a, Tree($t1$, root, $t2$)) ← Equal(a, root)|.

Binary(a, Tree($t1$, root, $t2$)) ← Less(a, root) | Binary(a, $t1$).

Binary(a, Tree($t1$, root, $t2$)) ← Greater(a, root) | Binary(a, $t2$).

Binary Search in Concurrent Prolog

The sequencing of atoms in the second clause of the Concurrent Prolog program is attained by the commit construct, that forces the guard Less(a, root) to be evaluated before the recursive call to Binary. We shall show that sequencing based on the commit operator may not be that easy in general.

Consider a binary tree, used to store a set of pairs ⟨index, word⟩, where index is an integer and word is an arbitrary size symbolic string over some finite alphabet. We represent such an object in logic by a term pair(i, w). Now, consider a PRISM program that searches for the object pair(i, w) stored in a binary tree. The ordering of atoms in the following program guarantees that an expensive comparison on the right portion of each object is not performed before a successful comparison of the left part.

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← (Equal(i, i'),
Equal, (w, w')).

Binary(pair(i, w), tree($t1$, pair(i', w), $t2$)) ← (Equal(i, i'),
Less(w, w'), Binary(pair(i, w), $t1$)).

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← (Less(i, i'),
Binary(pair(i, w), $t1$)).

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← (Equal(i, i'),
Greater(w, W$'$), Binary(pair(i, w), $t2$)).

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← (Greater(i, i'),
Binary(pair(i, w), $t2$)).

Binary Search for pairs in PRISM

If we attempt to perform a similar transformation we derive the following Concurrent Prolog program.

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← Equal, (i, i'),
Equal, (w, w')|.

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← Equal, (i, i'),
Less(w, w') | Binary(pair(i, w), $t1$).

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← Less(i, i')|
Binary(pair(i, w), $t1$).

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← Equal, (i, i'),
Greater(w, w') | Binary(pair(i, w), $t2$).

Binary(pair(i, w), tree($t1$, pair(i', w'), $t2$)) ← Greater(i, i)|
Binary(pair(i, w), $t2$).

Binary Search for pairs in Concurrent Prolog

The Concurrent Prolog program does not allow us to sequence the comparisons of the left and right portions of the compared objects. The problem is, that the commit construct forces the execution to commit to the branch where the construct was executed. In clauses 1, 2, and 4 we cannot commit before both comparisons in the guards were executed successfully, and therefore, we cannot have the commit between the guards. Thus, "commit" cannot be used exclusively as a sequencing construct, since it has an additional side-effect of terminating all sibling computations.

We now analyze this transformation slightly more formally. First we prove a technical lemma that will simplify our analysis. Specifically, we shall show that for every PRISM program there is an equivalent PRISM program in 2-normal form. It is straight forward to show that for each logic program Pr there is an equivalent program Pr$'$ in 2-normal form. The new program Pr′ is constructed by a transformation similar to the Chomsky normal form transformation,[21] which must be modified to accommodate for substitutions. The transformation here is analogous, with the exception that we must preserve control annotations in PRISM. Since we would like to focus on the control component, whenever no confusion may arise we omit the variables in the clause. We first illustrate our transformation with an example. Consider the goal:

$$\leftarrow (P, Q, [R, (S1, S2)], T, V)$$

This goal may be replaced with an equivalent PRISM program:

$$\leftarrow (P, \langle (Q, [R, (S1, S2)], T, V) \rangle)$$
$$\langle (Q, [R, (S1, S2)], T, V) \rangle \leftarrow (Q, \langle ([R, (S1, S2)], T, V) \rangle)$$
$$\langle ([R, (S1, S2)], T, V) \rangle \leftarrow (\langle [R, (S1, S2)] \rangle, \langle (T, V) \rangle)$$
$$\langle [R, (S1, S2)] \rangle \leftarrow [R, \langle (S1, S2) \rangle]$$
$$\langle (T, V) \rangle \leftarrow (T, V)$$
$$\langle (S1, S2) \rangle \leftarrow (S1, S2)$$

The expression $\langle (Q, [R, (S1, S2)], T, V) \rangle$ is just a new predicate name (whose variables are the same as the variables in the atoms Q, R, $S1$, $S2$, T and V), and so are the rest of the expressions enclosed in $\langle\ \rangle$.

Lemma 1. For every PRISM program there is an equivalent PRISM program in 2-normal form.

Proof. The proof is by induction on the length of the clause, i.e., the number of atoms in the body of the clause (denoted informally by $\| \|$).

For assertions our claim is trivially correct.

Assume then, that each PRISM clause of length $n-1$ has an equivalent PRISM program in 2-normal form. Let C be a PRISM clause and $|C| = n$. C must be either a P-group or an S-group. Without loss of generality assume C is an S-group (the proof for P-group is analogous), that is, C is of the form

$$A \leftarrow (B1, B2, \ldots, Bk).,$$

where $B1, B2, \ldots, Bk$ are P-groups and $k \leqslant n$. We replace C with the following PRISM program

$$A \leftarrow (\langle B1 \rangle, \langle B2, \ldots, Bk \rangle).$$
$$\langle B1 \rangle \leftarrow B1.$$
$$\langle B2, \ldots, Bn \rangle \leftarrow (B2, \ldots, Bk).$$

The first clause is already in 2-normal form, the second and the third clause are of length $n-1$, and by the induction hypothesis may be placed in 2-normal form.

As a result of this transformation we may assume that all the clauses in the program are of the form:

$$\langle (L1, L2)\rangle \leftarrow (L1, L2).,$$

$$\langle [L1, L2]\rangle \leftarrow [L1, L2].,$$

or

$$L\leftarrow. \quad \blacksquare$$

Corollary 2. Each deterministic PRISM program can be transformed to Concurrent Prolog using commit.

Proof. If the program satisfies conditions $C1$ and $C2$ a call to a procedure never invokes more than one procedure body. Thus for PRISM programs that satisfy conditions $C1$ and $C2$ the proof is straight forward by observing that any procedure of the form $P\leftarrow(Q, R)$ can be transformed to $P\leftarrow Q\,|\,R$.

But now let us consider a simple deterministic program that satisfies $C3$ but not $C1$ and $C2$.

$$P\leftarrow(Q1, R1)|.$$

$$P\leftarrow(Q2, R2)|.$$

The transformation using commit cannot not be applied to this program as before since positioning the commit operator after $Q1$ will cause aborting the execution of the second clause if $Q1$ succeeds. A simple trick resolves this problem. We convert this program to the following program.

$$P\leftarrow P1|.$$

$$P\leftarrow P2|.$$

$$P1\leftarrow(Q1\,|\,R1).$$

$$P2\leftarrow(Q2\,|\,R2).\ \square.$$

If we allow read-only variables in the guards (there are no read-only variables in PRISM) an interesting pathological scenario may occur as described in Ref. 19. Consider the following simple program:

$$\leftarrow P(x, y), Q(x, y)$$

$$P(A, y)\leftarrow P1(y?)\,|\,P2(y)$$

$$Q(x, B)\leftarrow Q1(x?)\,|\,Q2(x) \quad \blacksquare$$

In Concurrent Prolog the commit operator does not allow bindings to be passed to the caller until a commitment is made. Although x is bound to "A" and y is bound to "B" these bindings are not passed to $Q1$ and $P1$ respectively and they both enter a deadlock.

5.3. Transformation Using Read-Only Variables

In this section we present two distinct procedures to obtain a mechanical transformation for converting PRISM programs into Concurrent Prolog programs using read-only variables. The first method relies on the notion of state variables to obtain a Concurrent Prolog program from a PRISM program. The basic idea of the transformation is simple. With each event that needs to be monitored we associate a variable, called state variable, that will be bound at the occurrence of the event. Specifically, the events in PRISM that need to be represented are completions of procedures. Thus, if we want to sequence the execution of a goal $\leftarrow P(a), S(a)$, we convert the unary predicates $P(a)$ and $S(a)$ to binary predicates and convert the original goal to the goal $\leftarrow P(a, s), S(a, s?)$. We also have to ensure that the variable s in P will get bound to a nonvariable term only at the completion of P.

The transformation based on the state-variables method is given in the next example. To simplify the discussion we represent the terms m the clauses by a single variable x.

1. Each parallel procedure of the form:

   $$\langle [L1, L2] \rangle(x) \leftarrow [L1(x), L2(x)]$$

   is replaced with the procedure

   $$\langle [L1, L2] \rangle(x, T, \text{end } 12) \leftarrow L1(x, s, \text{end } 1), L2(x, s, \text{end } 2),$$
   $$\text{Complete}(\text{end } 1?, \text{end } 2?, \text{end } 12)$$

   At the completion of $L1$ and $L2$ the variables end 1 and end 2 (respectively) will be bound. This, in turn, will activate procedure Complete, which will result in the binding of end 12.

2. Each sequential procedure of the form:

   $$\langle (L1, L2) \rangle(x) \leftarrow (L1(x), L2(x)).$$

   is replaced with the procedure

   $$\langle (L1, L2) \rangle(x, T, \text{end } 12) \leftarrow L1(x, s, \text{end } 1), L2(x, \text{end } 1?, \text{end } 2),$$
   $$\text{Complete}(\text{end } 1?, \text{end } 2?, \text{end } 12).$$

The variable end 1 will not get bound until $L1$ has completed execution, and consequently $L2$ will be suspended until such time.

3. Finally, all the assertions in the PRISM program of the form:

$$L1(x).$$

are replaced with the assertion

$$L1(x, T, T).$$

The variables s, end 1, end 2 and end 12 are state variables and T is a new constant that does not belong to the Herbrand universe of the program. Procedure Complete is defined by the assertion:

$$\text{Complete}(T, T, T) \leftarrow$$

Intuitively, we have converted every n-ary predicate P in the program to a $n+2$ predicate. The two new variables monitor the execution of the atom in which they occur. Specifically, the $n+1$th variable monitors the beginning of the execution of the atom, and the $n+2$th variable monitors its completion.

Proposition 3. Let Pr be a deterministic PRISM program. The transformation previously given derives a Concurrent Prolog Program Pr′ whose execution is isomorphic to the original PRISM program.

Proof. The proof may be obtained by induction on the length of the refutation and is omitted.

The restriction to deterministic PRISM programs in Proposition 2 is necessary because the PRISM interpreter supports the parallel execution of OR-nodes, which is not facilitated in the current version of Concurrent Prolog.

Example. Consider the following clause that typically occurs in a merge-sort program.

(1) Sort(x, y) ← (Split(x, $x1$, $x2$), [Sort($x1$, $y1$), Sort($x2$, $y2$)], Merge($y1$, $y2$, y))

The semantics of this clause is self explanatory. Without going to the intermediate level of 2-normal form we could obtain the clause

(2) Sort(x, y, T, end) ← Split(x, $x1$, $x2$, s, end 1),

Sort($x1$, $y1$, end 1?, end 2),

Sort($x2$, $y2$, end 1?, end 3),

We observe that in this example (1) has a simple representation in Concurrent Prolog:

(3) $\mathrm{Sort}(x, y) \leftarrow \mathrm{Split}(x, x1, x2),$
$\mathrm{Sort}(x1?, y1),$
$\mathrm{Sort}(x2?, y2),$
$\mathrm{Merge}(y1?, y2?, y).$

The program in (3) is not operationally equivalent to (1), since (3) allows stream parallelism (pipelining). There is little doubt that (1) and (3) are equally natural representations, and (2) is a very cumbersome specification. We object to the use of state variables for sequencing because the programs do not preserve the original clean semantics which is affected by the introduction of state variables. Additionally, we shall show in Section 6 that control flow sequencing is amenable to a more efficient implementation.

The second method to transform PRISM programs to Concurrent Prolog programs is based on the simulation approach (see Refs. 1, 19, and 22). This method simply defines a meta-interpreter that simulates the execution of the desired control regime with given control facilities. The transformation based on the meta-interpreter method is depicted by

$I(p, T, s) \leftarrow \mathrm{Clause}(p \leftarrow \mathrm{body}) \mid I(\mathrm{body}, \_, s).$

$I([p, q], T, s) \leftarrow I(p, \_, s1), I(q, \_, s2), \mathrm{Complete}(s1?, s2?, s).$

$I((p, q), T, s) \leftarrow (p, \_, s1), I(q, s1?, s2), \mathrm{Complete}(s1?, s2?, s).$

$I([\ ], T, T).$

$\mathrm{Complete}(T, T, T).$

PRISM to Concurrent Prolog Transformation

The variables s, $s1$, and $s2$ are state variables and T is a new constant. The notation "_" denotes an arbitrary variable which is different from the remaining variables occurring in the clause. The correctness of this transformation may be proved by induction on the length of the execution.

There are several obvious problems with the practicality of the PRISM-Concurrent Prolog transformations presented. If the transformation were to be done by the user, the declarative semantics of the program is affected by the introduction of state variables. If the transformation were to be carried out by the system automatically, it seems to require a substantial penalty both in memory and time. In Section 6, we discuss an efficient implementation of PRISM control constructs, and the cost of the transformation will become evident. It should be mentioned that the use of partial evaluation methods may dramatically improve the effectiveness of meta-interpreters such as this one.

6. Efficiency Consideration in Parallel Reduction of Logic Programs

In this section we investigate several efficient schemes to implement control and data flow sequencing for logic programs. We shall assume that the control flow sequencing is specified using PRISM syntax, and data flow synchronization is attained by read-only variables. As before, when no confusion arises we omit the variables in the programs. A simple way to implement both schemes is by using a more general concept of an event. For control flow sequencing, the primitive event is defined as the completion of a procedure. For data driven languages, a primitive event is defined to be the binding of a nonvariable term to a read-only variable. We define a group event recursively to be:

- a primitive event
- a conjunction of group events.

We shall use a simplified language to describe events and group events. In PRISM, for each parallel procedure of the form: $S \leftarrow [P, Q]$ we define the following event script

$$S = \mathrm{AND}(P, Q)$$

that is, event S is defined as a conjunction of event P and event Q. Intuitively, the completion of S is equated with the completion of both P and Q. For each sequential procedure of the form

$$S \leftarrow (P, Q)$$

we write

$$S = Q$$

and

$$\text{on } P \text{ do } Q.$$

That is, the completion of S is equated with the completion of Q. The invocation of Q is dependent on the completion of P. Note that we do not need to define the start of a procedure as an event.

Now, the implementation of control/data flow sequencing is straight forward by using the standard notion of event queues. Event scripts, such as the previous one, are created at compile time. Also, for all the event scripts of the form

$$\text{on } P \text{ do } Q$$

we create an event queue that is indexed by the event P and stores pointers to all the procedures that depend on P for execution. When an event P occurs a message must be sent (in some way or another) to all the procedures that were stored in the event queue P. Analogously, data flow sequencing may be specified by attaching the script

$$x = \text{NON-VARIABLE } (x) \text{ and}$$

$$\text{on } x \text{ do } Q$$

for each clause of the form $S \leftarrow P(x), Q(x?)$. In this case the event queues are indexed with variable names. An event queue indexed by x stores all the procedures in which the variable x occurs as a read-only variable. For a thorough discussion of methods for efficient execution of general event languages the reader is referred to Ref. 23. There are a few important implementation points that should be mentioned, and are discussed in the following sections.

6.1. The Counter Method

Consider the goal $([P, Q, R], S)$. The invocation of procedure S is dependent on the conjunctive event AND(P, Q, R). A simple way to implement this construct is to associate a counter with the event AND(P, Q, R). The counter is initially set to 3, and is decremented by one as each procedure P, Q, or R completes its execution. When the procedure P is reduced, for example to $(P1, P2, P3)$ we start a new event script

$$P = \text{AND}(P1, P2, P3)$$

which will cause another to be created. In this method, a virtual tree of counters is created. Whenever some counter C in the tree becomes zero, it decrements the father-counter by 1.

An alternative method is at each reduction step to increment the counter by $n-1$, where n is the number of atoms in the new node that was created. In this example, we increment the counter for AND(P, Q, R) by 2.

Intuitively, the value of the counter is equivalent to the number of active atoms that must complete excution before S may be activated.

The two alternatives exhibit a common trade-off: time vs. space. The common counter method is most economical in terms of memory cells used. However, the counter may ultimately become a bottleneck since during the course of the execution it becomes accessible to an increasingly growing number of procedures. The counter-tree method uses $O(N)$ counters where N is the number of conjunctive events spanned by the program. However, the number of procedures accessing each counter is bounded. For example, consider a counter-tree, which is a complete binary tree of depth $n+1$. In the common counter method this tree is represented by a single counter whose value is 2^n. Now, assume that all the leaf counters of the tree are zero. In the tree method it takes $O(n)$ parallel time to update the root counter. In the common counter it will take 2^n s steps. From this discussion it is obvious that if our target implementation is a concurrent interpreter on a sequential machine then the common counter method is the method of choice.

Now, we can show why the method of simulating control flow with read-only variables seems to suffer from the disadvantages of both methods outlined in the previous sections. To verify this claim consider the conjunctive goal

$$\leftarrow([P, Q, R], S)$$

which creates the event script AND(P, Q, R). The reduction of P to ($P1$, $P2$, $P3$) is represented by creating the script $P =$ AND($P1$, $P2$, $P3$). This reduction process may be implemented by either the common counter method or the counter tree method as previously discussed. In the framework of read-only variables, (using the state variable transformation) we derive the clause

$$P(\text{end\_}P),\ Q(\text{end\_}Q),\ R(\text{end\_}R),\ S(\text{end\_}P?, \text{end\_}Q?, \text{end\_}R?)$$

So instead of one counter we used 3 variables. When P is reduced to ($P1$, $P2$, $P3$), we derive the clause

$$P1(\text{end }1),\ P2(\text{end }2),\ P3(\text{end }3),\ \text{Complete}(\text{end }1?, \text{end }2?, \text{end }3?, \text{end\_}P)$$

This step is analogous to the counter tree method, where the Complete predicates act as counters. However, during the execution we may have many simultaneous calls for Complete( ). Thus, inless the code of procedure Complete is distributed to all processors it becomes a bottleneck, like the counter in the common tree method. Additionally, the state variable method requires a time-space overhead since these variables are needlessly involved in the unification process. Thus, the simulation of control flow with data flow in the context of logic programming may cause inefficiency.

The situation becomes more involved when the procedures are nondeterministic and/or when shared variables are involved, though neither case presents a conceptual difficulty.

The former case when there are no shared variables among the conjunctive goals is solved by extending the event language with the script

$$P = \mathrm{OR}(P1, P2, \ldots, Pn)$$

where the *Pi*'s are the nondeterministic definitions of *P*.

In the case where the conjunctive goals share variables, the completion of the event $[P1, P2]$ is defined to be the completion of the events $P1$ and $P2$ with consistent bindings. Recalling the operational semantics of logic programs the completion of $[P1(x), P2(x)]$ is defined as:

$$[P1(x), P2(x)] = \mathrm{AND}(\mathrm{AND}(P1(x1), P2(x2)), \mathrm{UNIF}(x1, x2))$$

where $x1$ and $x2$ are the two new variables created for the variable x and UNIF is a function that guarantees the successful unification of $x1$ and $x2$. That is, the completion of $[P1(x), P2(x)]$ is defined as the conjunction of the event of completion of $P1$ and $P2$ with the event of fincing a set of consistent bindings for $P1$ and $P2$.

The approach outlined here is general and may be compiled to any general purpose machine. We must note that in case the interpreter is to be implemented on a typical Von-Neumann machine, the implementation of PRISM nested control flow may be accomplished by the same scheme that supports the implementation of fork-join in conventional languages. This is done by attaching GOTOs at the end of each parallel task.

Similar methods are possible for the implementation of read-only variable synchronization. In particular, with each variable that has a read-only annotation associated with it, we attach an event queue that contains all the atoms awaiting the binding of the variable. The event queue represents an address list of targets to which the binding for the variable should be sent. Additionally, with each atom we associate a counter that is initially set to the number of read-only variables in the atom. The atom is executable when the counter becomes zero.

6.2. Static Control Constructs

In this section we shall investigate the problem of choosing an efficient data-structure for the representation of PRISM clauses on a computer. In principle, the syntax of PRISM allows us to define control dependencies among the atoms of a clause. These dependencies may be represented as a control graph. The nodes of the graph are defined by the atoms of a clause, and the arcs describe the control dependencies among the atoms. For instance, the clause

$$\begin{aligned}\mathrm{Sort}(x, y) \leftarrow (&\mathrm{Split}(x, x1, x2), [\mathrm{Sort}(x1, y1), \mathrm{Sort}(x2, y2)],\\ &\mathrm{Merge}(y1, y2, y))\end{aligned}$$

is represented by the graph in Fig. 2. Many properties of this class of graphs, called series-parallel graphs, are studied in the literature. We shall also refer to such graphs as nested control graphs. In the context of this section we shall focus on their efficient representation on a shared memory parallel computer. Generally speaking, most graphs are effectively represented using an adjacency matrix, However, in our framework during the execution of the program the control graphs are subject to many insertions and deletions. Whenever a node is reduced, it replaced by another graph. When the node is solved it is simply deleted from the graph. We must first verify that during the execution PRISM goals maintain nested control flow. Otherwise, it is not meaningful to devise an efficient data structure for representing series-parallel graphs if this structure is not preserved in the dynamic graphs.

Lemma 4. During the execution PRISM logic programs maintain their nested control flow structure.

Proof. By induction on the length of the execution one can prove hat the nested control flow structure is preserved under insertions and leletions of nested control flow structures. The proof is omitted.

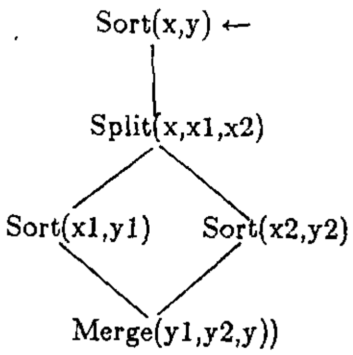

Fig. 2. A series-parallel graph for a sorting program.

A desirable property of a control construct from the efficiency point of view, is the ability of the interpreter to determine the set of executable procedures (nodes in our formalism) without excessive overhead. We shall refer to a control construct as a *static* control construct if the set of executable nodes may be determined in constant time. Intuitively, this definition implies that most processing may be done in compile time + additional small overhead during execution. A typical example of a construct that is not static is a selection strategy based on the number of instantiated variables.[4] In some cases the choice of a data structure determines whether the construct is static or not. For example, if we choose the adjacency matrix representation for control graphs the set of executable nodes cannot be determined in constant time, since it involves scanning the matrix. An alternative representation for this class of graphs is a linked list of control groups. With each control group we associate a descriptor, *s* or *p* depending on whether it is a parallel or a sequential group. For simplicity, assume the descriptor is the first element of each group. If a group consists of a single atom we shall omit the descriptor. For example, the Sort procedure may be represented as

$$\text{Sort}(x, y) \leftarrow (s\,\text{Split}(\ ), (p\,\text{Sort}(\ ), \text{Sort}(\ )), \text{Merge}(\ ))$$

In this representation we still cannot determine the set of executable atoms in constant time. This follows from the following examples.

$$(p(s\,P1, P2), (s\,P3, P4))$$

$$(p(s\,(p\,P1\ P2), P3), P6)$$

In the first example the executable atoms are *P*1 and *P*3, but the detection of this set involves scanning the list horizontally, i.e., across the list. In the second example *P*1, *P*2 and *P*6 are executable, but the detection of *P*1 involves scanning the list vertically, i.e., down the list. To alleviate this problem, with each control group we associate an auxiliary list of pointers to the immediately executable atoms in the group. Once these atoms have been executed we replace them with the group they precede in the linked list representation. The linked list is assumed to be a doubly linked list (to allow constant time insertions and deletions). In the next proposition we show that the doubly linked list with the auxiliary list of immediately executable atoms is adequate to support constant time detection of executable atoms.

Proposition 5. PRISM nested control flow is a static control construct.

Proof. At compile time a control flow graph that represents the control dependencies may be compiled. With each clause we associate a directed control graph that represents these dependencies and a list of atoms that have no other atoms in the control graph that precede them. In graph theoretic terminology these nodes are called sources. We shall refer to this list as an OPEN list. The control graph is assumed to be represented as a doubly linked list of control groups. We shall prove our claim by induction on the length of the execution t.

For $t=0$ (immediately preceding the beginning of processing) the list of executable nodes is equivalent to OPEN.

For $t>0$, we shall prove that OPEN either contains the list of executable nodes or may be updated in constant time. Clearly, the OPEN list cannot be perfectly up-to-date at any instance during the execution, since during any update some new events may happen. We therefore must prove that any time during the execution the OPEN list may be updated in constant time. If some node in OPEN has been replaced with a nonempty procedure, then the only new nodes that could be added to OPEN are nodes in the body of the procedure. This body is represented as a control group. and therefore has a list of immediately executable atoms associated with it. We just add this list to OPEN.

Thus, we have to show that once some procedure completes its control successors may be added in constant time. We shall assume the check for completion of an atom L is achieved by the counter associated with the event of its completion as explained previously. We recall that the counter is incremented for each new subgoal that the atom L spans, and decremented for each completion of such subgoal. Similarly, the completion of control groups

$$(P1, P2,..., Pn)$$

and

$$[P1, P2,..., Pn]$$

are associated with the events scripts

$$(P1, P2,..., Pn) = Pn$$

and

$$[P1, P2,..., Pn] = \text{AND}(P1, P2,..., Pn)$$

respectively. Consequently, the completion of these compound events may also be tested using a counter. Thus, the test of each event may be accomplished in constant time after the occurrence of the event itself.

Whenever an event occurs all the atoms that await its completion must be added to the OPEN list. If there is only one successor to the event, the successor may be added to the OPEN list in constant time. The problem arises when there are many successors to an event. However, the successor list for any event must be some control group G. This follows from the fact that PRISM execution preserves the nested control flow structure (Lemma 3). Thus, we may add the whole list of immediately executable atoms in G to OPEN. ∎

The next lemma shows that the data structure proposed in this section is not adequate support sequencing based on read-only variables.

Lemma 6. The read-only variable does not preserve the nested control flow structure.

Proof. The proof is by construction. Consider the Concurrent Prolog goal $\leftarrow P(x, y), Q(x?, y?)$ and the program

$$P(x, y) \leftarrow P1(x), P2(y).$$

$$Q(x, y) \leftarrow Q1(x, y), Q2(y).$$

The execution of this program derives the following goal $\leftarrow P1(x), P2(y), Q1(x?, y?), Q2(y?)$ which induces the control graph given in Fig. 3. which does not preserve a nested control flow structure. ∎

6.3. Scheduling Partially Ordered Logic Programs

One of the most important problems in the domain of concurrent system programming is related to the scheduling of parallel tasks to computing resources, e.g., allocating processes to processors. In general, the scheduling problem is complex and involves solving difficult combinatorial optimization problems. In this section we briefly compare the relative difficulty of scheduling PRISM tasks vs. Concurrent Prolog processes. Intuitively, one suspects that the complexity of scheduling is proportional to the structural complexity of the control structures that may be created by the control constructs available in the language. This section supports this intuitive conjecture for PRISM and Concurrent Prolog. In fact, we show that the complexity of scheduling for Concurrent Prolog is exponentially more difficult.

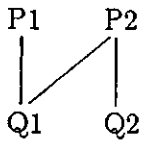

Fig. 3. A control graph not representable in PRISM.

Lemma 7. Let T be a set of tasks of duration (length) one unit each, and $\leqslant$ be a partial order on T. Given a set of m homogeneous multiprocessors the problem of finding an optimal schedule for T that does not violate the precedence constraints is NP-hard.

Proof. This follows from the NP-completeness of the following problem. Given an integer k, $0 \leqslant k$, find a schedule for T for which the completion time is less than k.[24]

We observe that given n partially ordered tasks, we can define a Concurrent Prolog goal that contains n literals with an equivalent partial order among the literals. This is accomplished by the following simple construction.

1. Set GOAL to nil.
2. For each pair of partially ordered tasks $A \leqslant B$ do:
   if A or B do not appear in GOAL we add $A(x)$, $B(x?)$ to GOAL.
   if A or B appear in GOAL, we modify GOAL by creating a new variable x and adding it to A and B. For instance if $A(x)$, $B(y)$ was already in GOAL we modify GOAL to be $A(x, z)$, $B(y, z?)$,....

Example. Consider the partial order

$$\{A \leqslant B, B \leqslant C, A \leqslant D\}.$$

The goal created by this procedure is

$$\leftarrow A(x, y), B(x?, z), C(z?), D(y?)$$

We note that the scheduling problem defined here has a polynomial solution if the partial order is restricted to be series-parallel.[25]

7. SUMMARY

In this paper we examined two methods for controlling the execution of parallel logic programs. We briefly summarize the main results reported in the paper.

We investigated control flow and data flow driven execution of logic programs. It is clear that data-flow primitives such as the read-only variable are more flexible since they allow arbitrary partial orders to be specified (see Section 6.3). Additionally, they allow to specify flexible communication schemes such as stream-parallelism and communicating partially instantiated data-structures. However, the relative complexity of scheduling an arbitrary partial order vs. a structured one favors the latter in terms of efficiency of implementation. Intuitively, we also feel that structured control constructs derive more readable and verifiable programs.

The main conclusion that may be drawn from the analysis presented in Section 5 is that in the framework of logic programming it is difficult to simulate control flow naturally by a general transformation using commit or read-only variables. In specific cases we may be able to write very natural specifications based on read-only variables. This statement is also supported by Ref. 19. Thus, it seems that both data-flow and control flow facilities are desirable for a general purpose parallel logic programming language.

We proposed a simple low level language to implement both PRISM nested control flow and Concurrent Prolog read-only variables. The main idea was to convert the control/data dependencies into simple event scripts and then use existing methods to implement these scripts efficiently. Subsequently, we proposed a data structure that allows the efficient implementation of PRISM nested control flow.

As mentioned before, this paper focused on control-driven and data-driven execution of parallel logic programs. In Ref. 1, we examined other facilities for execution control such as communication control and event based sequencing.

ACKNOWLEDGMENTS

I thank Professor Jack Minker for his support and advice while this work was in progress. Thanks are due to Madhur Kohli, Rich Piazza, and Deepak Sherlekar for helpful discussions. The constructive comments of the referees contributed greatly to the final form and content of the paper. This research was conducted as part of the design and development of PRISM, while the author was at the University of Maryland, College Park. The support of U.S. Air Force Office of Scientific Research under Contract AFOSR-82-0303 is gratefully acknowledged.

REFERENCES

1. S. Kasif, Analysis of Parallelism in Logic Programs, Ph. D. Thesis, Computer Science Dept., University of Maryland., (1984).
2. R. A. Kowalski, *Logic for Problem Solving*, North-Holland, (1979).
3. S. Kasif, A Note on Translating Flowchart and Recursive Schemas to Prolog Schemas, TR-1273, Department of Computer Science, University of Maryland, College Park, Maryland, (May 1983).
4. S. Kasif, M. Kohli, and J. Minker, PRISM—A Parallel Inference System Based on Logic, Technical Report TR-1243, Computer Science Department, University of Maryland, (February 1983).
5. C. Rieger, R. Trigg, and R. Bane, ZMOB: A New Computing Engine for AI, *Proc. of IJCAI-81*, Vancouver (also University of Maryland CS TR-1028), (August 1981).
6. E. Y. Shapiro, A Subset of Concurrent Prolog and Its Interpreter, TR-003, *ICOT*, (1983).

7. K. Clark and S. Gregory, PARLOG: Parallel Programming in Logic, *ACM TOPLAS* 8(1) (January 1986).
8. M. J. Wise A Parallel Prolog: The Construction of A Data Driven Model, *ACM Symp. on LISP and Functional Programming*, Pittsburgh, (August 1982).
9. K. Ueda, Guarded Horn Clauses, TR-103, ICOT, Japan, (1985).
10. K. L. Clark and F. McCabe, IC-Prolog Language Features, in *Logic Programming*, K. L. Clark and S. A. Tarnlund, eds. Academic Press, London, pp. 253–267 (1982).
11. S. Kasif, M. Kohli, and J. Minker, Control Facilities of PRISM—A Parallel Inference System Based on Logic, *Proc. of the Int. Joint Conf. on Artificial Intelligence*, (August 1983).
12. Mark, Giuliano, Madhur Kohli, Jack Minker, Arcot Rajasekar, and Deepak Sherlekar, Parallel Logic Programming in PRISM: Initial Experimental Work, Technical Report, Department of Computer Science, University of Maryland, College Park, (1985).
13. Paul H. Morris, A Dataflow Interpreter for Logic Programs, *Proc. of the First Workshop on Logic Programming*, pp. 148–159, (1980).
14. M. H. van Emden and G. J. de Lucena, Predicate Logic as a Programming Language for Parallel Programming, in *Logic Programming*, K. L. Clark and S. A. Tarnlund, eds., Academic Press, (1982).
15. K. L. Clark and S. Gregory, A Relational Programming Language for Parallel Programming, Research Report, Imperial College, (July 1981).
16. K. Ueda, Concurrent Prolog Re-examined, TR-102, *ICOT*, Japan, (1984).
17. V. A. Saraswat, Problems with Concurrent Prolog, CMU Technical Report, (June 1985).
18. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, Data-Driven and Demand-Driven Computer Architecture, *Computing Surveys* 14(1): 93–143 (March 1982).
19. A. J. Kusalik, Process Serialization in a Concurrent Prolog, *New Generation Computing* 2(3): 289–298 (1984).
20. S. Gregory, Design, Application and Implementation of a Parallel Logic Programming Language, Ph. D. Thesis, Imperial College, London, (1985).
21. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading Massachusetts, (1979).
22. L. M. Pereira and L. F. Monteiro, The Semantics of Parallelism and Coroutining in Logic Programming, Report 2/80, Departmento de Informatica, Universidade Nova de Lisboa, (January 1978).
23. A. Reuveni, The Event Based Language and Its Multiple Processor Implementations, Ph. D. Thesis, MIT/LCS/TR-226, (January 1980).
24. J. D. Ullman, NP-Complete Scheduling Problems, *JCSS* 10:384–393, (1975).
25. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to NP-completeness*, Freeman and Company, San Francisco, (1979).