AN AND-PARALLEL EXECUTION MODEL OF LOGIC PROGRAMS

Bernd Schwinn, Gerhard Barth[1)]

ABSTRACT

This paper deals with the most important kinds of parallelism that can occur in logic programs, OR parallelism and AND parallelism. To explore parallelism we use a data flow model, where the rules and facts of the logic programs are represented as graphs. Beginning with a basic model for OR parallel execution of logic programs, where all the subgoals of a rule are persued in sequential order, we give an extended model, where we try to detect and exploit independencies among subgoals during the execution of a rule (dynamic AND parallelism). The prime when extending the basic model was to improve execution time. The sizes of the graphs remain in a tolerable range.

1. Introduction

Programming in logic has evolved during the last years as an alternative for Lisp in the area of knowledge engineering. Yet, the common predicative programming systems, such as Prolog [Clo81] use a strict sequential inferencing mechanism. This method is based on the resolution principle [Fur82] and traverses an AND/OR search tree in preorder while trying to find a solution for a given goal [Kow79]. Because this method is very time consuming, the development of parallel implementations for predicative programming systems has become an increasingly important area of research. Conery has classified the following kinds of parallelism in logic programs [Con 81]:

(1) OR parallelism
   - searching for alternative solutions in parallel

(2) AND parallelism
   - investigating independent subgoals in parallel

(3) Stream parallelism
   - eager evaluation of structured data (streams)

(4) Search parallelism
   - parallel searching in partitioned sets of Horn clauses

This paper is on the first two kinds of parallelism, OR parallelism and AND parallelism. The second section introduces a basic model for parallel execution of logic programs where Horn clauses are represented as data flow graphs. This model allows the efficient implementation of OR parallelism. Section 3 describes the extension of the basic model. By introducing special test- and compare-operators independencies among subgoals may be detected dynamically.

---

1) University of Kaiserslautern, Department of Computer Science, P.O. Box 3049, 6750 Kaiserslautern, West Germany

This leads to an optimal derivation ordering for subgoals. The graph representation of a sample clause is shown in the fourth section. In this context the improvements in running time as well as the sizes of the graphs in the extended model are discussed. Section 5 describes a simulation system for the extended model and sketches the components which a possible parallel architecture should have. The terminology used throughout this paper is that of [Clo81].

2. MODEL FOR OR PARALLEL EXECUTION OF LOGIC PROGRAMS

In common predicative programming systems the clauses are ordered sequentially and are processed in that order to find a solution for a given goal. In parallel execution models, all the possible clauses to derive a subgoal are executed concurrently (OR parallelism). This leads to an enormous speed up. Alternative solutions for the goal can be found in parallel.

To explore OR parallelism in logic programs we use a method which is very similar to that of Umeyama/Tamura [Ume83]. In our system the rules and facts of logic programs are translated into independent data flow graphs. These graphs consist of nodes and edges, where the edges are used to transport tokens between the nodes. The nodes denote simple operations. In our basic model we use five of them. The data flow mechanism allows an operation to be executed as soon as all its operands are available. To activate the graphs we use the dynamic interpretation model [Arv78]. In this model several independent computations in a graph may proceed in parallel.

2.1 COLORED TOKENS

In the data flow model a token T can be regarded as a message between two operations. It consists of the following components:

T = [N, CT, SCT, DST, LD]

where
- N = token number
- CT = context
- SCT = saved context
- DST = destination, specified by (C, U, P)
  - where C = process number
    - U = unit number
    - P = port number
- LD = literal data

The literal data portion of a token consists of a list of the variable bindings so far established.

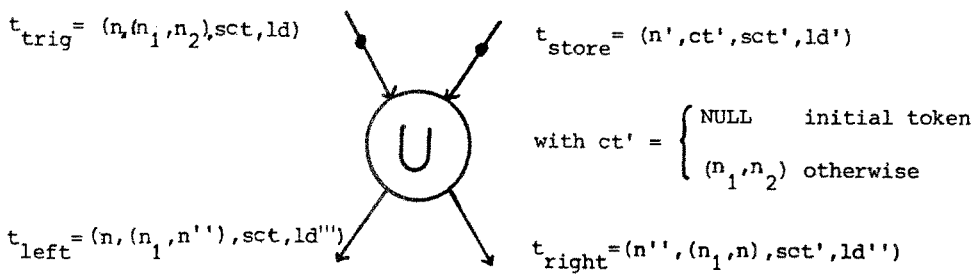
2.2 BASIC OPERATIONS

In our basic model we use the following five simple operations:

U	=	Unification
C	=	Copy
M	=	Merge
A	=	Apply
R	=	Return

Among these the unification (U) is the most important and expensive operation. The other operations are used to copy tokens (C), to merge the data flow (M) or to process a context change when applying other clauses (A) or returning from other clauses (R). The semantics of all basic operations are described next by using colored tokens. To simplify matters the DST-field is not listed explicitly.

(1) U(nification)-operation

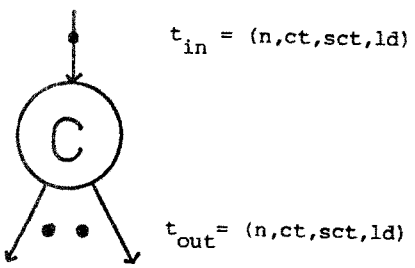


The U-operation has two input and two output ports. The input ports are classified into a store port (right) and a trigger port (left):

a) Store port
Tokens arriving at the store port are simply stored in the unification unit.

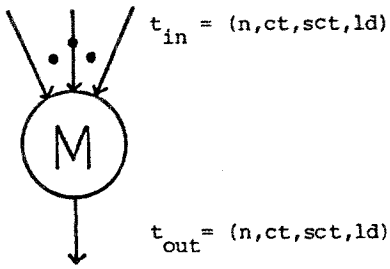
b) Trigger port
When a token arrives at the trigger port a search is carried out for a stored token with the same context. The latter token is copied and the copy is unified with the token from the trigger port. If unification succeeds, the two tokens created within this process are sent to the output ports. Otherwise, the two tokens are discarded.

(2) C(opy)-operation



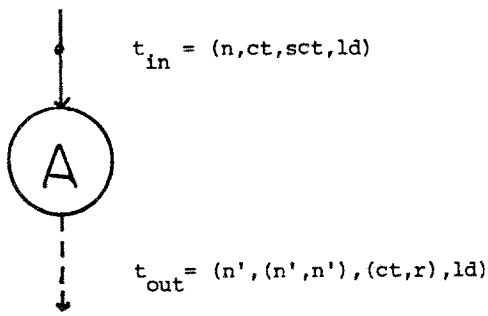
The C-operation has one input port and any number of output ports. A token arriving at the input port is copied to each output port.

(3) M(erge)-operation



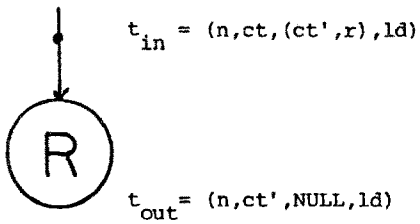
The M-operation has any number of input ports and one output port. Each token arriving at an input port is copied to the output port.

(4) A(pply)-operation



The A-operation has one input port. A token arriving at it is sent to the first node of all graphs for rules and facts with the same predicate name as the subgoal to be derived. Prior to that a context change is performed. The old context (CT) and the return address (physical successor of the A-node in the graph) are saved and a new context is generated for every OR parallel derivation.

(5) R(eturn)-operation



The final node in each graph is an R-node. A token arriving at the input port is sent to the return address in the calling graph after the old context (CT') has been restored. On the highest level, where no return address exists, a solution for the initial goal has been constructed.

2.3 DATA FLOW GRAPHS FOR RULES AND FACTS

After specification of the semantics for the basic operations we now concentrate on the graphs to be constructed for the rules and facts. A rule has the form:

$$p \text{ :- } q_1, q_2, \ldots, q_n.$$

Therefore the graph in Fig. 1 is constructed.

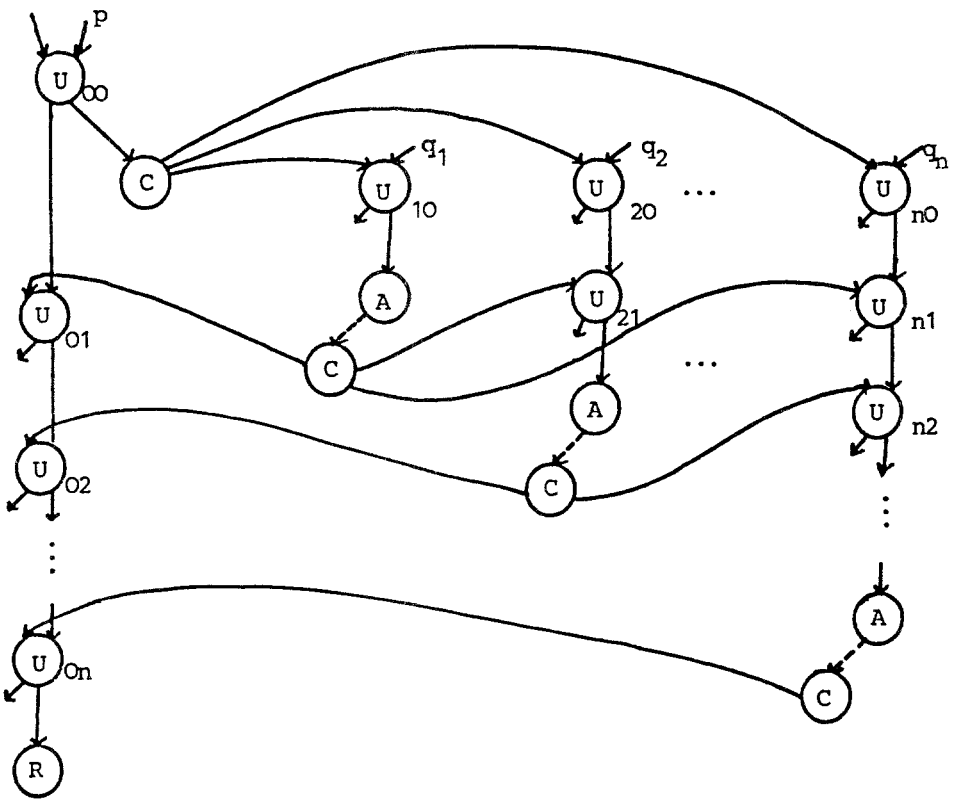


Fig. 1:

After the unification with head p (U_{00}) the unifications for the subgoals appearing in the rule can be transformed in parallel ($U_{10},\ldots,U_{n0}$). On the other side the derivation of a subgoal q_i must wait until the results of the sugoals $q_1, \ldots, q_{i-1}$ have been verified so that the results of those solutions can be consumed ($U_{i1}, \ldots, U_{i(i-1)}$). On the left hand side of the graph the derivation results of the subgoals are assembled ($U_{01}, \ldots, U_{0n}$) to form the result of the entire rule, which is passed to the final R-node.

The graph for a fact p. consists of a U-node for the head and a R-node to return the result:

The OR parallelism in this model is realized by simultaneously passing a token to each rule or fact that could be used to derive a subgoal. Verification of subgoals in this model is performed sequentially from left to right, no AND parallelism has yet been realized.

3. MODEL FOR AND PARALLEL EXECUTION OF LOGIC PROGRAMS

Systems with AND parallelism try to exploit independencies of subgoals, so that the execution of the subgoals can start as early as possible. There are two different ways to detect independencies of arguments:

(1) Marking of variables within the clauses declares certain subgoals as producers or consumers of values for the variable. This method was chosen in the predicative programming languages Parlog [Cla83], Concurrent Prolog [Sha83] and Epilog [Wis82]. In these systems the programmer himself must detect and organize the AND parallelism. Another problem is that the general nature of clauses makes it very difficult to detect all possible independencies of subgoals in this way.

(2) The not-annotated AND parallelism needs no extension of the logic programming language. The programmer needs not care about the AND parallel execution within the clauses, he can concentrate on the complete and consistent formulation of his knowledge. The rules and facts are transformed into an intermediate code, where special test-operations are used to detect the real dependencies among the subgoals, for every given goal.

In our model we chose a dynamic method to realize AND parallelism. This means that the independencies are explored when derivation actually uses a rule and not when the rule is defined.

At this point we will study some cases where dependencies of subgoals arise:

1. If we have a rule of the form

   $p(X,Y,\ldots)$:- $q_1(X)$, $q_2(Y)$, ...

   then in most cases the subgoals $q_1(X)$ and $q_2(Y)$ are independent and can be executed concurrently. On the other hand, a dependency among these subgoals can arise when a goal of the form p(A,A,...) has to be derived. This means that it should be possible to decide at execution time where those variable bindings occur.

2. If we have a rule of the form

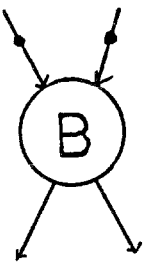
   $p(\ldots, X, \ldots) \;:-\; q_1(X), q_2(X), q_3(X), \ldots$

   then the subgoals $q_1(X)$, $q_2(X)$ and $q_3(X)$ are normally dependent on each other and should be derived one after another to produce only one value for variable X. In cases where execution of subgoal $q_1(X)$ produces a value for X, the subgoals $q_2(X)$ and $q_3(X)$ can be derived concurrently. If the rule is activated with a constant value for X, all three subgoals could be executed in parallel. Therefore a mechanism is needed to detect the point where a constant value has been produced for a variable, so that the verification of dependent subgoals can be started earlier.

This leads to an extension of the basic model:

(6) B(inding-test)-operation

t_{in} = (n,ct,sct,ld)



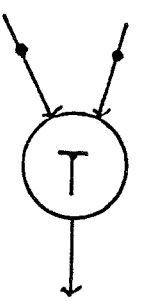
(l_1, l_2)

t_{out} = (n,ct,sct,ld)

The B-operation has two input ports and two output ports. For a token arriving at the left input port a test is performed to see if a variable of list l_1 is bound to a variable of list l_2. The lists l_1 and l_2 reside permanently at the right input port. If a variable binding occurs for the input token, the token is sent to the right output port. Otherwise, it is sent to the left output port.

(7) T(est)-operation

t_{in} = (n,ct,sct,ld)



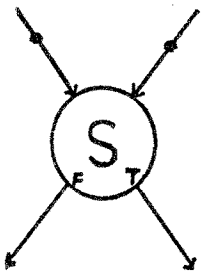
l

t_{bool} = (n,ct,sct, $\frac{\text{TRUE}}{\text{FALSE}}$)

The T-operation has two input ports and a boolean output port. For a token arriving at the left input port a test is done if the variables of list l are bound to constant values. In this case TRUE is sent to the boolean output port, otherwise FALSE. The list l resides permanently at the right input port.

(8) S(witch)-operation

$t_{in} = (n,ct,sct,ld)$



$t_{bool} = (n',ct,sct', \overset{TRUE}{\underset{FALSE}{/}})$

$t_{out} = t_{in}$

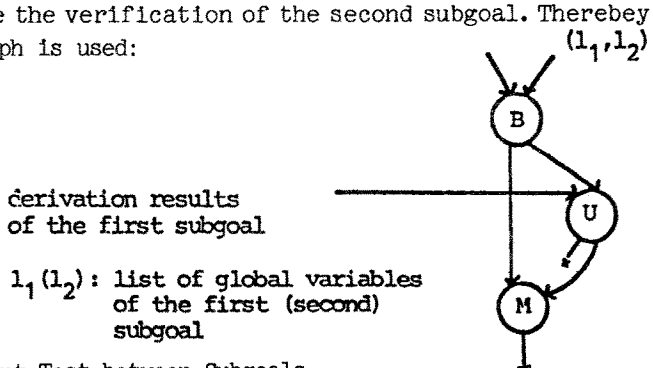
The S-operation has two input and two output ports. Depending on the value of t_{bool} the token t_{in} is passed to the right (true) or to the left (false) output port.

Independencies of subgoals can be revealed by these three operations. While constructing the graph, pairs of subgoals are examined, whereby the following cases are distinguished:

(a) Unconditioned Dependency of Subgoals
Hereby the two subgoals have a temporary[1] in common that has not appeared before the first of these subgoals. In this situation the second subgoal needs the bindings initiated by the first, no dependency test is necessary.

(b) Unconditioned Independency of Subgoals
This occurs if the subgoals do not share variables nor do both of them contain global variables (except temporaries that occur the first time). These two subgoals are independent and can be executed concurrently without any further tests.

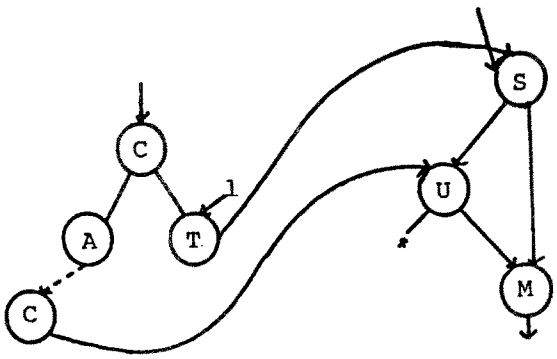
(c) Binding-Test between Subgoals
If the two subgoals have no common variables, but variables appear in both termlists (except new temporaries), then a binding-test is done before the verification of the second subgoal. Therebey the following subgraph is used:



(d) Constant-Test between Subgoals
If case a) does not apply, common variables (global or temporaries) exist but not both termlists contain other variables, then a constant-test for the common variables has to be performed. Thereby the following subgraph is used:

---

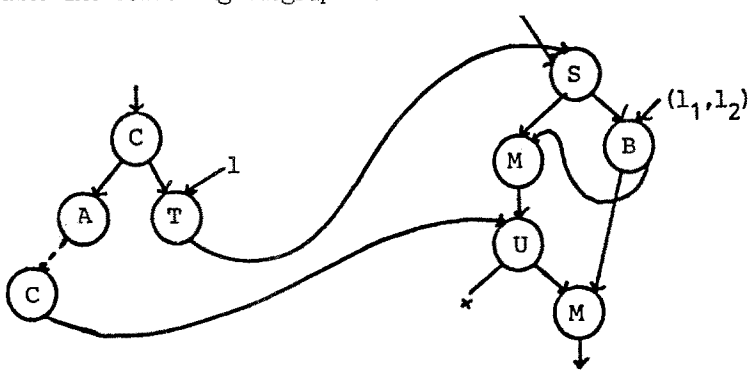
1) Temporaries are variables that do not appear in the head of a rule



l: list of common variables

(e) Constant/Binding-Test between Subgoals

If there are common variables and both termlists contain further variables, a test must be performed to see if the common variables have constant values before the first subgoal is derived. If this is true, a binding of other global variables can lead to dependency of the subgoals. The following subgraph realizes the test:



l: list of common variables
l_1 (l_2): list of global variables of the first (second) subgoal not appearing in the second (first)

4. EXAMPLE

An example illustrates the modifications of the graphs to perform dependency-tests at execution time. Improvements with respect to the basic model will be pointed out. Suppose the following rule is given:

$$p(X_1,X_2,X_3) :- q_1(X_2,T_1),\ q_2(X_3,T_2),\ q_3(X_1,X_2,T_1,T_2),\ q_4(X_2,X_3,T_2).$$

This rule contains the global variables X_1, X_2 and X_3 and the temporaries T_1 and T_2. The first two subgoals seem to be independent. Yet, variable bindings can create a dependency between these subgoals. For example, when goal p(A,B,B) has to be derived. Because of the common temporaries the third subgoal must be verified later than the first and second subgoal. The last subgoal must be persued after the second. It is possibly independent of the first and third subgoal. This is true if X_2 is initialized with a constant value and constant values for X_3 and T_2 are derived by the second subgoal. In our extended model the graph in Figure 2 is produced for this rule.

In this relatively complicated example, cases a), c), d), e) from above apply.

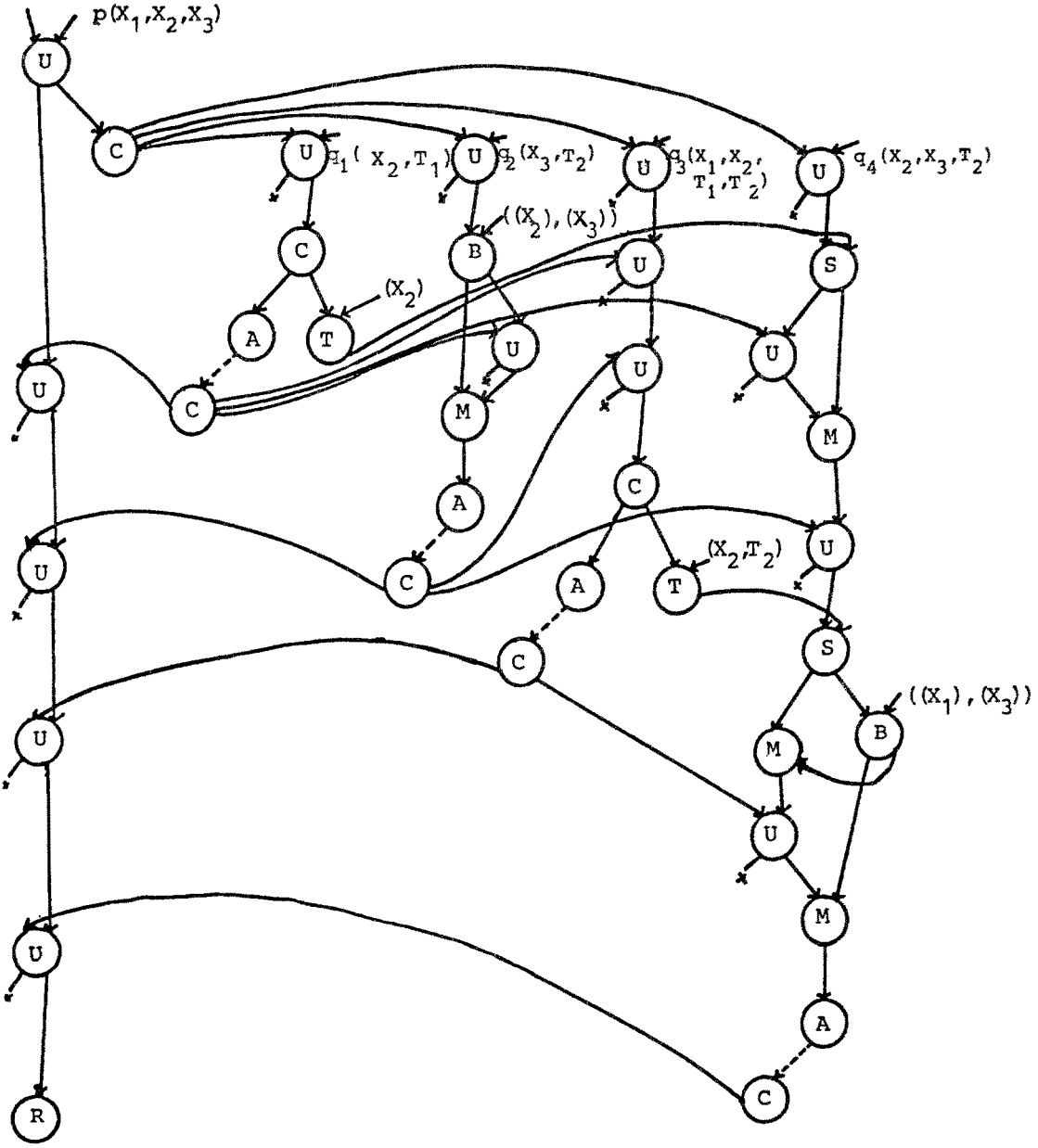
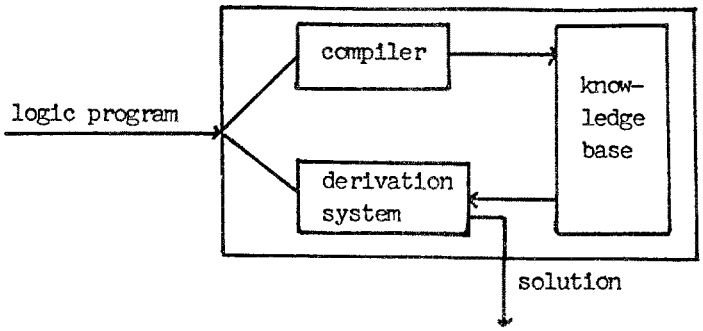


Fig. 2:
Since it is realistic to assume that the T, S and B operations are much less time consuming than unification, it is obvious that our implementation achieves considerable savings in time. In the above example, we can earn up to 50% savings in time. The graph of the extended model contains 37 nodes, as compared to the 25 nodes in the basic model. Normally, the rate of growth is much less. Particularly, if unconditioned independencies exist, the graphs in the extended model can become even smaller than those in the basic model.

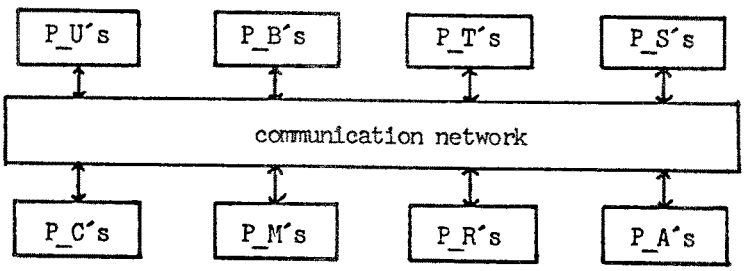
5. FURTHER INVESTIGATIONS

A simulation system for our model is just being implemented. This system consists of a compiler to generate data flow graphs for rules and facts and an interpreter to perform the derivation of goals. The interpreter is similar to the U-interpreter of Arvind, Gostelow [Arv82].

simulation system:



In our simulation system the user may select a compiler for the basic model or one for the extended model. Thus, the performances of these two models may be compared. Furthermore, empirical estimates about the number of required processors P_X, where X denotes the 8 different basic operations, may be thereby obtained.



Presently, in our execution model dependencies among subgoals are resolved strictly from left to right. In many applications this can lead to performing a lot of calculations that later in the derivation are detected to be useless. Therefore, the actual parameters of a rule should determine the sequence in which dependent subgoals should be considered. The realization of this idea in the extended model will be our next goal. The investigation of the other kinds of parallelism, stream parallelism and search parallelism, will be done in the near future.

REFERENCES

[Arv78] Arvind, Gostelow K.P., Plouffe W.: An Asynchronous Programming Language and Computing Machine, DCS Report 114A, University of California, Irvine, Dec. 1978.

[Arv82] Arvind, Gostelow K.P.: The U-Interpreter, IEEE Computer, pp. 42-49, Feb. 1982.

[Cla81] Clark K.L., Gregory S.: A Relational Language for Parallel Programming, Research Report of Imperial College of Science and Technology, Dec 81/16, July 1981.

[Cla83] Clark K.L., Gregory S.: PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, Imperial College, March 1983.

[Clo81] Clocksin W.F., Mellish C.S.: Programming in Prolog, Springer Verlag, Berlin 1981.

[Con81] Conery J.S., Kibler D.F.: Parallel Interpretation of Logic Programs, Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, pp. 163-167, October 1981.

[DeG84] De Groot D.: Restricted AND Parallelism, Proceedings of the International Conference on Fifth Gerenation Computer Systems, ICOT, 1984.

[Den74] Dennis J.B.: First Version of a Data-Flow Procedure Language, Lecture Notes in Computer Science, Vol. 19, pp. 362-376, Springer-Verlag, 1974.

[Fur82] Furukawa K., Nitta K., Matsumoto Y.: Prolog Interpreter Based on Concurrent Progamming, Proceedings of the First International Logic Programming Conference, Marseille, France, pp. 38-41, September 1982.

[Ito85] Ito N., Schimizu H.: Data Flow Based Execution Mechanisms of Parallel and Concurrent Prolog, New Generation Computing 3, pp. 15-41, 1985.

[Kow79] Kowalski R.: Logic for Problem Solving, North Holland, 1979.

[Sha83] Shapiro E.Y.: A Subset of Concurrent Prolog and its Interpreter, ICOT Techn. Report, TR 003, Feb. 1983.

[Ume83] Umeyama S., Tamura K.: A Parallel Execution Model of Logic Programs, The 10th Annual International Symposium on Computer Architecture, ACM, pp. 349-355, June 1983.

[Wis82] Wise M.: A Parallel Prolog: The Construction of a Data Driven Model, University of New South Wales, Australia, 1982.