An Execution Model for Exploiting AND-Parallelism in Logic Programs

Yow-Jian Lin
Vipin Kumar

AI TR86-34 September 1986
An Execution Model for Exploiting
AND-Parallelism in Logic Programs

Yow-Jian Lin
Vipin Kumar

Artificial Intelligence Laboratory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

This paper presents a parallel execution model for exploiting AND-parallelism in pure Horn Clause logic programs. The model is based upon the execution model of Conery and Kibler, and compares favorably with various related execution schemes (including Conery and Kibler’s). We also presents two implementation schemes to realize our model: one which has a coordinator to control the activities of processes solving different literals; and the other one which achieves synchronization by letting processes passing messages to each other in a distributed fashion. Trade-offs between these two schemes are then discussed.

This work was supported by Army Research Office grant #DAAG29-84-K-0060 to the Artificial Intelligence Laboratory at the University of Texas at Austin.
# Table of Contents

1. Introduction .................................................................................................................. 1
2. Assumptions and Definitions ......................................................................................... 3
3. The Execution Model ...................................................................................................... 4
   3.1. The Forward Execution Algorithm ......................................................................... 4
       3.1.1. The Algorithm ............................................................................................ 4
       3.1.2. Example ...................................................................................................... 6
       3.1.3. Comparison .................................................................................................. 7
       The Forward Execution scheme of Chang, et. al. ..................................................... 7
       Restricted AND Parallelism (RAP) of Degroot ....................................................... 9
       The Forward Execution Algorithm of Conery and Kibler ...................................... 10
   3.2. The Backward Execution Algorithm ....................................................................... 12
       3.2.1. The Algorithm ............................................................................................ 12
       3.2.2. Example ...................................................................................................... 13
       3.2.3. Comparison .................................................................................................. 14
       The Backward Execution Algorithm of Conery and Kibler .................................................. 14
       The Semi-Intelligent Backtracking of Chang and Despain ..................................... 15
       The Intelligent Backtracking Schemes of Brunooghe-Pereira-Porto and Cox-Pietrzykowski-Matwin .......................................................... 16
   3.3. How to Find More Than One Solution? .................................................................. 16
4. Implementations ............................................................................................................ 17
   4.1. The Distributed Implementation ............................................................................. 17
   4.2. The Centralized Implementation ........................................................................... 21
   4.3. Discussion .............................................................................................................. 23
5. Concluding Remarks ..................................................................................................... 24
Appendix .......................................................................................................................... 26
References ........................................................................................................................ 40
Figures ............................................................................................................................... 42
An Execution Model for Exploiting AND-Parallelism in Logic Programs

Yow-Jian Lin  
Vipin Kumar  
Artificial Intelligence Laboratory  
Department of Computer Sciences  
The University of Texas at Austin  
Austin, Texas 78712

1. Introduction

Exploiting AND-parallelism in logic programs refers to executing more than one literal of a conjunct at the same time. When literals share variables, solving them independently could lead to excessive computation for most practical problems. Various approaches have therefore been proposed to exploit AND-parallelism in a more limited form [6] [2] [7] [13] [3] [8]. In the generator-consumer approach followed by many researchers [6] [2] [7], if more than one literal in a conjunct share an uninstantiated variable \( V \), then one of these literals is designated as the generator of \( V \) and the remaining literals are designated as the consumers of \( V \). The execution of any consumer of \( V \) is not started until the execution of the generator of \( V \) has finished. One such approach was developed by Conery and Kibler in the context of AND/OR process model [6]. Their execution model consists of two main parts: a forward execution algorithm which selects generators of shared variables and schedules the execution of various literals; and a backward execution algorithm which chooses backtracking literal when some literal fails and coordinates the execution of other affected literals. Unfortunately, as we have shown in [11], their backward execution algorithm is incorrect; i.e., it can miss some solutions while performing backtracking (Woo and Choe later presented similar result in [14]). Their forward execution algorithm uses an ordering algorithm to identify the generator of every uninstantiated variable. Each time a non-ground binding is generated, the ordering algorithm is applied again to identify a new set of gen-
operators. Since the ordering algorithm may have to be repeatedly executed while interpreting a logic program, the run-time overhead of setting-up AND-parallelism in their execution model can be very high.

In order to avoid the problem of excess run-time overhead, several schemes have been presented [2] [7]. A scheme proposed by Chang, et. al. [2] makes use of static data-dependency analysis. The execution graphs and backtracking paths are all derived at compile time, based on the worst case activation of each predicate. The restricted AND-parallelism (RAP), proposed by Degroot [7], creates one execution graph expression for each clause at compile time. It then dynamically generates parallel execution sequences based on these execution graph expressions. However, since PLM makes use of compile-time worst case analysis and RAP utilizes limited execution graph expressions, these two schemes, unlike Conery and Kibler's scheme, can not always explore AND-parallelism to the fullest extent allowable in the generator-consumer approach.

We have developed a parallel execution model which is a variant of Conery and Kibler's scheme, and appears to be more suitable for exploiting AND-parallelism in pure Horn Clause logic programs. Like their execution scheme, our model also exploits AND-parallelism in the framework of generator-consumer approach, and consists of a forward execution and a backward execution algorithm. Our forward execution algorithm can achieve the same degree of AND-parallelism as it is achieved in Conery and Kibler's model but with less run-time overhead. Our backward execution algorithm backtracks intelligently within clause, i.e., when a literal $P_i$ of a clause fails, backtracking can be done directly to a literal which is not necessarily the left neighbor of $P_i$ in the clause. We prove that our algorithm is correct (i.e., no solution is missed while performing intelligent backtracking). Preliminary versions of our forward execution algorithm and backward execution algorithm were presented in [10] and [11] respectively.
Section 2 defines certain terms used in the paper. Section 3 presents an abstract description of our execution model. Examples and comparison with other related research are also provided. Section 4 presents details of two possible implementation schemes, and discusses the trade-offs between them. Section 5 contains concluding remarks.

2. Assumptions and Definitions

In our execution model, we are dealing with pure Horn Clause logic programs. Unlike Conery and Kibler's execution model which uses a run-time ordering algorithm to identify the generator of each uninstantiated variable, our model makes use of a linear sequence of literals for each clause. These linear sequences are constructed at compile time using problem-specific knowledge such as variable annotations, programmer-suggested ordering, etc (See Appendix A for the details of our ordering algorithm). The ordering of literals is only meant to increase the execution efficiency. The semantics of the logic program is not changed by the ordering. Throughout the discussion in this paper, we assume that, for each clause, the literals in the body have been ordered, and $P_i$ represents the $i$-th literal in the linear sequence. In addition, $P_0$ stands for the head literal.

In addition to the linear sequence which specifies a total order among literals in the same clause body, we also define a partial order between literals of a clause as follows. A literal $P_i$ is a child of $P_j$ (and $P_j$ is a parent of $P_i$) if $P_i$ consumes bindings generated by $P_j$. $P_i$ is a successor of $P_j$ (and $P_j$ is a predecessor of $P_i$) if

i) $P_i$ is a child of $P_j$, or

ii) $P_i$ is a child of $P_k$ and $P_k$ is a successor of $P_j$.

A literal $P_i$ is a leaf literal if $P_i$ has no child. It is obvious that the partial ordering can be depicted as a data-dependency graph.

A literal can be in one of the three states: GATHER, EXECUTING and SOLVED. When a literal $P_i$ is in the GATHER state, it is waiting to receive bindings for the variables of which $P_i$ is
a consumer. When $P_i$ is in the EXECUTING state, it is trying to find a solution based upon its current bindings. $P_i$ is in the SOLVED state if it has succeeded in its execution. To exploit AND-parallelism, each literal is executed by a different process. For brevity, we will often use "a literal $P_i$$"$ to mean "the process that executes $P_i$$"$.

3. The Execution Model

This execution model specifies how the execution of literals in a clause body should proceed to exploit AND-parallelism. Our execution model consists of the following two algorithms: a forward execution algorithm which is executed after the head literal has been unified with a given goal; and a backward execution algorithm which is executed when some literal fails. The following presents abstract description of these two algorithms. Different implementation schemes will be discussed in Section 4.

3.1. The Forward Execution Algorithm

3.1.1. The Algorithm

Our forward execution algorithm selects generators for uninstantiated variables and decides when to start the execution of each literal. Among the literals which share a variable $V$, the leftmost literal in the linear sequence is selected as the generator of $V$. The remaining literals become the consumers of $V$. A literal $P_i$ becomes executable (i.e., the corresponding process can actually begin to solve $P_i$) when it is the generator of all the uninstantiated variables in its own binding environment. The choice of leftmost literal (as generator) guarantees absence of deadlock (i.e., a situation in which none of the unexecuted literals can proceed).

During the execution of a clause, there exists a token (a list of literals denoted as $F$-list($V$)) for each variable $V$. $F$-list($V$) consists of those literals in the clause body which are the consumers of $V$. Literals $P_j$ in each $F$-list are sorted according to the ascending order of $j$. At run time,
tokens are generated and passed around from one literal to another according to a simple procedure \( S_0 \). Conceptually, if a token \( F(V) \) resides in a literal \( P_i \), then \( P_i \) is designated as the generator of \( V \).

When a clause with a non-null body is used to solve a literal, a process, corresponding to the head literal \( P_0 \) of the clause, is created. Tokens for different variables \( V \) in the head literal are initialized (to contain every literal in the clause body which has \( V \)) and given to \( P_0 \). The token for each free variable \( V \) (i.e., a variable that does not appear in the head literal) in the body is created and assigned to the leftmost literal of the clause containing \( V \). Note that for a given linear sequence of literals, these tokens can be computed at compile time. After \( P_0 \) has been successfully unified with a given goal, Procedure \( S_0 \) is executed to pass tokens to some literals in the clause body, and the forward execution begins as follows.

Initially (i.e., before \( P_0 \) is unified with a given goal), all literals \( P_i \) \((1 \leq i \leq n)\) are in the GATHER state. A literal \( P_i \) is switched from the GATHER state to the EXECUTING state and becomes executable when it receives tokens for all the uninstantiated variables in its binding environment. When a literal \( P_i \), running in the EXECUTING state, succeeds in its execution, it goes to the SOLVED state and suspends its execution.\(^1\) If \( P_i \) has generated bindings for some variables \( V \), then the binding environment of the consumers of \( V \) is updated, and Procedure \( S_0 \) is executed to create and distribute tokens for newly generated variables. If \( P_i \) fails while running in the EXECUTING state, then the backward execution algorithm (see Section 3.2) is invoked.

The execution of the clause succeeds when all the literals in the clause body are in the SOLVED state.

Note that in the description as above, we have not mentioned when the process for each

\(^1\) Note that if \( P_i \) is a generator of any variable and OR-parallelism is being exploited, then \( P_i \) could keep on producing new solutions. To simplify discussion in this paper we have assumed that no OR-parallelism is being exploited. Our execution model works even when OR-parallelism is exploited (see Lin’s dissertation [12] for details).
literal in the clause body should be created. We also have not specified who is responsible for
deciding whether a literal is executable, and who would update the binding environment of con-
sumers of V after a literal produces a binding for V. These details are implementation-dependent,
and will be discussed in Section 4.

Procedure $S_0$
Suppose $P_j$ has generated bindings for some variable $V_i$.
If, for every such $V_i$,
i) the binding $T_i$ generated by $P_j$ is ground, or
ii) $F$-list($V_i$) is null,
then the procedure exits.
Otherwise, for each new variable $X$ produced by $P_j$, a temporary $F$-list, $FT(X)$, is first
created and initialized to contain all the literals $P_k$, where $P_k$ is in any $F$-list($V_i$) such that $X$
appears in the binding for $V_i$. A new token $F$-list($X$) = tail($FT(X)$) is then created for $X$ and
given to head($FT(X)$).

3.1.2. Example

Depending on various binding conditions, different parallel execution sequences can be
explored by this algorithm. An example of executing a given clause

$$p_0(X,Y,Z) :- p_1(Y,W), p_2(X,W), p_3(Z), p_4(W,Z), p_5(Y)$$

is shown in Figure 1, where an arc with label $X$ means that the token of variable $X$ has been sent
along the direction of arc. A literal which has a dotted variable $V$ means that the literal holds the
token of $V$. For the given ordering of literals in the clause body (Figure 1a), this example shows
two possible sequences of parallel execution corresponding to unification with two different
goals. At compile time, tokens $F$-list($X$)=[p2], $F$-list($Y$)=[p1,p5], $F$-list($Z$)=[p3,p4], are initially
given to the process $p_0$, and $F$-list($W$)=[p2,p4] is given to $p_1$.

In the first case, after unification with a given goal $p_0(A,B,C)$, the head literal $p_0$ generated
bindings $A/X$, $B/Y$, $C/Z$. According to Procedure $S_0$, token $F$-list($A$)=[] is passed to literal $p_2$, as
$FT(A)=[p2]$. Similarly, token $F$-list($B$)=[p5] is passed to $p_1$, and token $F$-list($C$)=[p4] is passed
to $p_3$. At this time both literals $p_1$ and $p_3$ have received tokens for all of their variables in the
current binding environment, and can be executed in parallel (Figure 1b). Suppose that p1 generates bindings $x_0/B$, $w_0/W$, and p3 generates binding $z_0/C$. Since all new bindings are ground, no token has to be passed around. Both literals p4 and p5 are executable, as there are no uninstantiated variables in their current binding environments. Literal p2 is also executable, since it has received token for A from p0 previously. Therefore p2, p4, and p5 can all be solved in parallel (Figure 1c). The data-dependency graph corresponding to the first execution sequence is shown in Figure 1d.

In the second case, p0 generates bindings $V/X$, $V/Y$, and $z_0/Z$ for a given goal p0($V,V,z_0$). Since FT($V$) = F-list($X$) + F-list($Y$) = [p1,p2,p3], a token F-list($V$) = [p2,p5] is passed to p1. This time literals p1 and p3 can be executed in parallel (Figure 1e). After the execution of p1, binding $S/W$ and $v_0/V$ is generated, and p2 and p5 can start executing in parallel (Figure 1f). After p2 generates binding $s_0/S$, p4 becomes executable (Figure 1g). The data-dependency graph corresponding to the second execution sequence is shown in Figure 1h.

3.1.3. Comparison

In the following, we compare our forward execution scheme with other related forward execution schemes. All these schemes are based upon the generator-consumer approach but have different methods for deciding when a literal becomes executable.

The Forward Execution Scheme of Chang, et. al.

To decide whether a literal is executable, the scheme by Chang, et. al. [2] uses a static data-dependency graph which is generated at compile time based upon the worst case activation.\footnote{The worst case activation, as defined in [2], refers to the binding condition where each variable has the worst type of its possible bindings. Among three types of bindings, NGD(non-ground and dependent) is worse than NGI(non-ground and independent), and NGI is worse than G(ground).}

A literal $P_i$ is "data-dependent" on another literal $P_j$ (i.e. $P_i$ is a child of $P_j$ in their data-dependency graph) if, in the worst case, $P_i$ should wait for bindings generated by $P_j$. A literal
can become executable only after all of its parents in the static graph have succeeded in their execution. Since the worst case scenario may not happen, some data-dependency relations shown in the static graph do not have to exist at run time. Therefore a literal \( P_i \) may be ready to be executed (i.e., \( P_i \) is the generator of all the uninstantiated variables in its binding environment) long before this scheme makes it executable. In our algorithm, a literal becomes executable as soon as it is ready to be executed at run time. Therefore, our algorithm is able to exploit parallelism in cases where their scheme would fail to do so. This is illustrated in the following.

Suppose that the worst activation mode of \( p0 \) is

\[ p0(NGD,NGD,NGD), \]

and the derived activation modes and exit modes\(^3\) (based on the worst activation mode of \( p0 \)) of \( p1 \) to \( p5 \) are

<table>
<thead>
<tr>
<th>activation mode</th>
<th>exit mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p1(NGI,NGI) )</td>
<td>( p1(G,NGI) )</td>
</tr>
<tr>
<td>( p2(NGI,NGI) )</td>
<td>( p2(NGL,G) )</td>
</tr>
<tr>
<td>( p3(NGI) )</td>
<td>( p3(G) )</td>
</tr>
<tr>
<td>( p4(G,G) )</td>
<td>( p4(G,G) )</td>
</tr>
<tr>
<td>( p5(G) )</td>
<td>( p5(G) )</td>
</tr>
</tbody>
</table>

then Figure 2 shows the static data-dependency graph used by their scheme. Suppose at run time the clause is called to solve \( p0(V,V,z_0) \). Our forward execution algorithm will achieve the dynamic data-dependency graph shown in Figure 1h, whereas Chang's approach would still use the static graph of Figure 2. From both graphs, it is clear that when actual activation is better than the worst case activation, Chang's approach fails to exploit all the possible AND-parallelism.

The construction of static data-dependency graphs in Chang's approach is also worthy of discussion. The programmer first declares initial activation mode of each predicate. Using an

\(^3\) The derived exit mode of a predicate \( P \) is the binding condition of variables in \( P \) after the execution of \( P \), based on a given activation mode. The derived activation mode of \( P \) is the activation mode derived from the activation mode of the head literal and the derived exit modes of previously solved literals.
iterative procedure, the compiler then derives the worst case activation of each predicate. The data-dependency graph of each clause is then generated based on the worst case activation of its head literal. In order to provide reasonable initial activation mode of each predicate, the programmer must be very much aware of all kinds of binding conditions which can happen at run time. If the programmer is not aware of these conditions, or if a variety of binding conditions can happen, then it is quite possible for the static data-dependency graph to degenerate into a linear list. This would mean that no AND-parallelism would be exploited. An example is shown in Appendix B.

The primary strength of their scheme is that the run-time checking is simpler; i.e., to decide whether a literal \( P_i \) is executable, it only needs to check if all the parents of \( P_i \) in the static data-dependency graph have succeeded in their execution.

**Restricted AND Parallelism (RAP) of Degroot**

To decide whether a literal is executable, RAP checks the bindings of a set of variables to see if a group of literals (which consume those bindings) are still sharing any variable. If they do not share variables, then literals in the group are all executable. Otherwise only one of them is executable, and similar checking may be performed afterward. Using an execution graph expression for each clause, Degroot’s approach can explore different data-dependency graphs dynamically. However, as discussed in [7], due to the predetermined execution sequence of substatements it may not be able to start processing a subgoal as soon as the subgoal is ready to be executed.

Suppose the following execution graph expression is produced by the compiler:
(GPAR(Y,Z)  
  SEQ  p1(Y,W)  
  (IF IPAR(Z,X)  
    (GPAR(W)  p2(X,W)  p4(W,Z))  
    (SEQ  p2(X,W)  p4(W,Z)))  
  (IPAR(Z,Y)  p3(Z)  p5(Y)))

If we make the same assumption about bindings as we did in the second part of our example in Section 3.1.2, then Degroot’s approach will achieve the data-dependency graph of Figure 3. As we can see, the execution of p3(Z) is postponed even if it is ready to be executed from the very beginning. The execution of p5(Y) is also postponed due to the predetermined execution sequence of substatements. Therefore in Degroot’s approach, AND-parallelism can not be exploited to the fullest extent either. Note also that sometimes type checking could be redundant in Degroot’s approach. For example, if GPAR(Y,Z) is true, then IPAR(Z,Y) is surely true. However since IPAR(Z,Y) has to be there to cover the case when GPAR(Y,Z) is false, the redundant type checking is unavoidable. Overall the run-time overhead in Degroot’s approach does not appear to be less than the overhead in our approach, even though Degroot’s approach exploits only a limited amount of AND-parallelism.

It is important to point out that, for any given clause, it is possible to create an execution graph expression in Degroot’s approach which would be able to exploit all the AND-parallelism allowable in the generator-consumer approach. Unfortunately, such expression would be too complicated, and would involve excessive redundant run-time checking. Even if RAP is used to exploit only a limited amount of AND-parallelism, constructing a compiler which can produce reasonable execution graph expressions still appears to be a nontrivial task due to combinatorial combinations of binding conditions.

The Forward Execution Algorithm of Conery and Kibler

Compared with Conery and Kibler’s approach, our scheme as well as their scheme can explore AND-parallelism to the fullest extent allowable in the generator-consumer approach, as
independent literals can always be executed in parallel. Moreover, if all the bindings produced by the generators are ground, then both schemes would achieve the same data-dependency graph for each clause (provided that the linear order of literals used by both schemes are the same). When generators can produce non-ground bindings, their scheme must execute the ordering algorithm every time a non-ground binding is generated, hence the data-dependency graphs achieved by both schemes could be different. Although repeated execution of the ordering algorithm may reorder the literals, it does not guarantee that a better data-dependency graph would be achieved. As we can see in Figure 4, during the execution of this particular example, the ordering algorithm must be executed three times (after the head literal p0(X,Y,Z) is unified with the goal p0(V,V,Z) and after the execution of both p1 and p2) by their scheme, but the resulting data-dependency graph is identical to the graph produced by our approach. When the heuristic rules used by their scheme are fallible, the new graph could even be worse. Since our token passing scheme is simpler than the ordering algorithm of Conery and Kibler, the run-time overhead of our algorithm in general is less than the overhead of their forward execution algorithm.

In some cases, dynamically changing the linear sequence of literals could lead to better performance. However, when the choice of alternative linear sequences depends only on the activation mode of the head literal, techniques such as control alternatives used in IC-Prolog [4] would allow our scheme to use a different optimal sequence for each possible activation. Since our scheme does not change the linear sequence when the execution of literals in the clause body starts, the only case in which Conery and Kibler's forward execution scheme (as well as Degroot's RAP) could do better is when the linear sequence must be reordered with respect to the bindings produced by a generator in the clause body. However, these two schemes may still fail to do so by not using proper heuristic rules (in Conery and Kibler's forward execution scheme) or not having smart compiler to transfer clauses to proper execution graph expressions (in RAP).
3.2. The Backward Execution Algorithm

3.2.1. The Algorithm

Our backward execution algorithm is executed when some literal $P_i$ fails (i.e., when $P_i$ cannot find any more solutions while executing in the EXECUTING state). This algorithm consists of two parts:

i) Select a proper backtracking literal $P_j$ that may possibly cure the failure of $P_i$;

ii) Appropriately change the status of literals whose computation is directly or indirectly dependent upon the previous bindings generated by $P_j$.

To select a backtracking literal, each literal $P_i$ dynamically maintains a list of literals denoted as B-list($P_i$). B-list($P_i$) consists of those literals which may be able to cure the failure of $P_i$ (if $P_i$ fails) by producing new solutions. The literals $P_k$ in each B-list are sorted according to the descending order of $k$. Initially, B-list($P_i$) is null when a process is first created to solve $P_i$. Each time $P_i$ consumes a binding generated by another literal $P_k$ (including the head literal) and $P_k$ is not in B-list($P_i$), $P_i$ adds $P_k$ to B-list($P_i$). Clearly, when $P_i$ is switched from the GATHER state to the EXECUTING state, B-list($P_i$) contains all the literals $P_k$ which have generated bindings to be consumed by $P_i$.

When a literal $P_i$ fails, $P_j=$head(B-list($P_i$)) is selected as the backtrack literal. If $P_j = P_0$, then the execution of the clause fails. Otherwise, a redo request is sent to $P_j$ asking $P_j$ to generate a new solution. The tail of B-list($P_i$) is also passed to $P_j$ and merged into B-list($P_j$) so that if $P_j$ is unable to cure the failure of $P_i$, backtracking may be done to other literals in B-list($P_i$).

In Appendix C, we prove that, at the time $P_i$ fails, B-list($P_i$) contains enough backtracking candidates to ensure that no solution will be missed during backtracking.

After receiving a redo request, $P_j$ goes to the EXECUTING state and resumes execution to try to produce a new set of bindings. Before doing this, all the bindings that $P_j$ has previously supplied to other literals $P_m$ must be canceled, as these bindings are now outdated. The literals
\(P_m\) whose bindings are canceled then go to the GATHER state. But before doing that they should cancel bindings that they have previously supplied to other literals. Now consider a literal \(P_k\) such that \(k > j\) and some successor (not necessarily child) of \(P_k\) has been canceled due to the failure that caused redoing of \(P_j\). For each such \(P_k\), if \(P_k\) has supplied more than one set of bindings to its successors, then \(P_k\) should be reset to resupply all the previous bindings, as these old bindings may be acceptable now. This can be done by transferring \(P_k\) to the EXECUTING state, as if \(P_k\) has just made the transition from the GATHER state to the EXECUTING state for the first time. Before making the transition, \(P_k\) needs to cancel the bindings it has supplied to other literals (i.e., it has to cancel all of its children). If a literal has to be both reset and canceled, then cancel overrides reset. An iterative procedure that can be used to find all literals \(P_k\) that need to be reset is given in Appendix D. This procedure is rather complex, hence we adopt a simple scheme in which all literals \(P_k, k > j\) and \(P_k\) has successors, are reset.

The above backward execution algorithm can be implemented in various ways. Two schemes will be discussed in Section 4. To be correct, any implementation should maintain the following invariant:

Let \(P_k\) be a literal that has most recently been canceled due to the redo operation on \(P_i\). Suppose \(P_k\) has started executing after having gone through the GATHER state. At this time, all the bindings used by \(P_k\) were

i) either produced by literals that were not affected (i.e., were not canceled or reset) by redo operation on \(P_i\) or any other redos previous to the redo operation on \(P_i\);

ii) or produced after these literals had been appropriately canceled or reset due to these redos.

Furthermore, if \(P_k\) is using any binding supplied by \(P_i\), then this binding should have been produced by \(P_i\) after receiving the redo request.

Note that redo operation on \(P_i\) is previous to redo operation on \(P_j\) if \(P_i\) fails in its redoing and causes redo operation on \(P_j\). The "previous" relation among redo operations is transitive.

3.2.2. Example
The following is an example of how our backward execution algorithm works. Assume that we are solving $P_6(A,B,C)$ with the following set of clauses.

$$P_6(A,B,C) \leftarrow P_1(A), P_2(A,B), P_3(A,C), P_4(C), P_5(B,C).$$

$P_1(a1)$.
$P_2(a1, b1)$.
$P_2(a1, b2)$.
$P_3(a1, c1)$.
$P_3(a1, c2)$.
$P_4(c1)$.
$P_4(b1, c2)$.
$P_3(b2, c1)$.

Suppose $P_1, P_2, P_3, P_4$ have all succeeded. $P_5$ is about to fail. Figure 5a shows the data-dependency graph as well as the B-list and state of each literal at this moment. When $P_5$ fails, $P_3 = \text{head}(\text{B-list}(P_5))$ is selected as the backtrack literal, and $[P_2, P_0] = \text{tail}(\text{B-list}(P_5))$ is merged into B-list($P_3$). $P_3$ is asked to redo, and $P_4$ and $P_5$ are canceled. Figure 5b shows the status after backtracking has occurred due to the failure of $P_5$.

$P_3$ succeeds in producing another binding. This time $P_4$ fails. Figure 5c shows the status right before the failure of $P_4$, and Figure 5d shows the status after backtracking has occurred due to the failure of $P_4$. Backtracking is done to $P_3$ again.

$P_3$ fails. $P_2$ is correctly selected as the backtrack literal. $P_5$ is canceled, which then causes $P_3$ to be reset and $P_4$ to be canceled. Figure 5e shows the current status. The execution then proceeds gracefully, and the final solution is $a1/A, b2/B, c1/C$.

3.2.3. Comparison

The Backward Execution Algorithm of Conery and Kibler

The backward execution algorithm presented by Conery and Kibler [5] [6] is incorrect because, in some cases, it may miss solutions while performing backtracking. The problem with their algorithm is that it uses only one failure context list to record the history of failure. When a new process is created for a failed literal $P_i$, $P_i$ is removed from the failure context, and the
history about the failure of $P_i$ is lost.

Consider the example given in Section 3.2.2. When $P_5$ fails, the failure context becomes \{P_5\}, and $P_3$ is asked to redo. When a new process is created for $P_5$ after $P_3$ has succeeded again, $P_5$ is removed from the failure context, and the failure of $P_5$ is lost. Later on after $P_4$ has failed and caused the failure of $P_5$, the failure context becomes \{P_4, P_3\}. According to Conery and Kibler's approach, $P_1$, instead of $P_2$, is selected as the backtracking literal. Since there is no other alternative for solving $P_1$, their approach fails to find any solution for $P_9(A,B,C)$.

The Semi-Intelligent Backtracking of Chang and Despain

Chang and Despain [1] identify two backtrack literals (Type I and Type II) for each literal, based on the same static data-dependency graph constructed for their forward execution scheme. Type I backtrack literal is chosen for backtracking if $P_i$ fails without producing any solutions; Type II backtrack literal is used if $P_i$ fails because all the solutions generated by $P_i$ were not accepted by the successors of $P_i$ in the static data-dependency graph.

Note that our scheme does not use a data dependency graph to choose backtracking literals. However, the transfer of bindings between literals implicitly constructs a data dependency graph (if a literal $P_i$ receives a binding from $P_j$, then $P_i$ becomes a child of $P_j$ in this graph). If the run time graph of our approach happens to be identical to the static graph generated in [1], then Type I backtracking literals would be chosen identically in both approaches; but the choice of Type II backtrack literal in Chang's approach can still be worse (but never better) than our approach. If the compile time graph generated by Chang's approach is not identical to our run time graph (their graph is constructed for the worst case scenario that may not happen in a real case), then their scheme can do worse even for Type I backtracking. In general, if possible activations of a clause are not known (or can be diverse), then the statically constructed data-dependency graph of the clause may be too restrictive and may even degenerate into a linear ordering, making backtracking completely depth-first.
The Intelligent Backtracking Schemes of Bruynooghe-Pereira-Porto and Cox-Pietrzykowski-Matwin

A number of intelligent backtracking schemes (related to each other) have been developed by these authors. All these schemes analyze the conflicts that arise during the unification process to decide the backtrack point. Since the backtrack point in these approaches can be anywhere in the proof tree, these approaches can perform across-the-clause backtracking intelligently.

In AND/OR process model, the proof tree is distributed at various AND-processes, and each AND-process has information about only one clause. Hence it is not clear how these schemes could be adopted to perform intelligent backtracking in parallel execution models based upon the AND/OR process model. In [9] we present an extension of our backtracking scheme that can perform across-the-clause backtracking intelligently and is applicable to execution models based upon the AND/OR process model.

3.3. How to Find More Than One Solution?

In order to find more than one solution for a clause, we assume that for each clause, there is a dummy literal $P^d$ appearing at the end of the linear sequence of literals such that $P^d$ has all the variables as its arguments that appear in the head literal. When $P^d$ has tokens for all of its uninstantiated variables, then, unlike other literals, it directly goes to the SOLVED state.\(^4\) When all the literals of the clause are in the SOLVED state, then the bindings of the variables in $P^d$ constitute a solution of the clause.

When another solution is requested for a given clause, we simply make the dummy literal $P^d$ fail and let $P^d$ start the backtracking session. This would automatically restart the execution of the clause. If there are no more solutions, then the execution will terminate with failure. Other-

---

\(^4\) $P^d$ being in SOLVED state does not necessarily mean that the execution of the clause has succeeded, as some other literals may still be in the EXECUTING state.
erwise, a new solution will be found when all the literals of the clause are in the SOLVED state again.

4. Implementations

Conceptually our forward and backward execution algorithms are distributed, but they can be implemented both in distributed and centralized ways. This section introduces two possible implementations of our execution model. In the first implementation, execution status and other relevant information for performing forward and backward execution are distributed at the processes executing the literals of the clause. The processes communicate directly with each other via exchanging messages. In the second implementation, all the information is kept in a coordinator, and all the communication messages are channeled through the coordinator. In both approaches, we assume that the communication channels are reliable: i.e., messages are not lost.

4.1. The Distributed Implementation

In this implementation, a process is created for each literal \( P_i \) in the clause. This process keeps track of the state, the binding environment, and the B-list of \( P_i \) as well as the tokens residing at \( P_i \). Initially, all processes \( P_i \) (except \( P_0 \)) are in the GATHER state, and \( \text{B-list}(P_i) = [] \). Tokens for the variables of the clause are created and given to appropriate processes at the time of process creation. After the head literal \( P_0 \) unifies with a subgoal, it executes \( S_0 \) to create new tokens. The newly generated bindings and tokens are then passed by \( P_0 \) to all the relevant literals \( P_i \). A process \( P_i \) moves from the GATHER state to the EXECUTING state when it has tokens for all the uninstantiated variables in its own binding environment. While executing if \( P_i \) succeeds, then \( P_i \) also runs \( S_0 \) to create new tokens, and distributes these tokens and generated bindings to relevant processes. After that \( P_i \) goes to the SOLVED state. When a process \( P_j \) receives bindings from \( P_i \), it includes \( P_i \) in \( \text{B-list}(P_j) \).
Recall from Section 3.3 that the execution of the clause terminates when all literals are in the SOLVED state. At this time bindings of variables in $P^d$ constitute a solution to the clause. Since $P_0$ would need to know this solution to be able to communicate it to the subgoal that invoked the clause, we allow $P^d$ to be executed by the process for $P_0$. One way of finding that every literal is in the SOLVED state is to let each literal send a SUCCESS message to $P^d$ after it has reached the SOLVED state. When $P^d$ has received a SUCCESS message from every literal, a new solution to the clause has been found. Actually, if $P^d$ is a consumer of some bindings produced by a literal $P_j$, then it is not necessary for $P_j$ to send a separate SUCCESS message to $P^d$, as upon receiving bindings from $P_j$, $P^d$ knows that $P_j$ has succeeded. Thus a SUCCESS message needs to be sent from $P_j$ to $P^d$ only if $P_j$ does not send any bindings to $P^d$.

The description given so far works smoothly for deterministic programs. But in non-deterministic logic programs, since the literals can fail, we need some mechanism to ensure the invariant stated in Section 3.2.1. One possible method, described in the following, is to associate the concept of time with each binding generated.

Let a local clock $LC(P_i)$, owned by each process $P_i$, denote the number of REDOs which $P_i$ has received but not ignored since it last moved from the GATHER state to the EXECUTING state. A global clock $GC$ (corresponding to the execution of a clause) can be constructed as the concatenation of local clocks of all the processes solving literals in that clause. That is, $GC = LC(P_1); LC(P_2); \ldots; LC(P_{n-1}); LC(P_n)$. Since the concatenation follows the linear sequence of literals, it should be obvious that $GC$ is monotonically increasing during the execution of a clause, and hence provides us the notion of time.

Whenever $P_i$ succeeds in its execution, the success of $P_i$ is based on the bindings produced by $P_1, \ldots, P_{i-1}$. Whenever any literal in $P_1, \ldots, P_{i-1}$ needs to redo, bindings received by $P_i$ (hence the success of $P_i$) can become invalid. If each literal $P_i$ has up-to-date information about its partial global clock $PGC(P_i) = LC(P_1); LC(P_2); \ldots; LC(P_{i-1}); LC(P_i)$, then whenever any literal $P_j$,
\(1 \leq j \leq n-1\), needs to redo, \(P_i\) can check to see whether some of its bindings have become invalid. But in a distributed implementation, it is impossible for \(P_i\) to have up-to-date information about the local clock of other literals. Hence we let each process maintain its own (possibly outdated) view of the partial global clock \(\text{PGC}(P_i)\). Any time a process \(P_i\) needs to redo, it increments \(\text{LC}(P_i)\) and informs all literals \(P_{i+1}, \ldots, P_n\) about it so that they can update their partial clocks and take any necessary action. In a distributed environment, messages can take arbitrary time to go from one process to another. Therefore it is possible for a message to become outdated by the time it reaches its destination process. To allow the receiving process to find out whether a binding or a message is outdated, each binding or message going out of \(P_i\) is stamped with the current partial global clock of \(P_i\). Time ordering between two partial global clocks \(\text{PGC}(P_i)\) and \(\text{PGC}(P_j)\) is determined by the lexicographic ordering between first \(m(= \min\{i, j\})\) components of \(\text{PGC}(P_i)\) and \(\text{PGC}(P_j)\) as follows:

If there exists a component \(\text{LC}(P_k)\) such that \(k = \min\{x \mid \text{LC}(P_x)\text{ is different from }\text{LC}(P_x)\text{ in }\text{PGC}(P_j), 1 \leq x \leq m\}\),

then

If \(\text{LC}(P_k)\) in \(\text{PGC}(P_j)\) is smaller than \(\text{LC}(P_k)\) in \(\text{PGC}(P_i)\),

then \(\text{PGC}(P_j)\) is older than \(\text{PGC}(P_i)\).

Otherwise, \(\text{PGC}(P_j)\) is more-recent than \(\text{PGC}(P_i)\).

Otherwise, there is no time-order between \(\text{PGC}(P_i)\) and \(\text{PGC}(P_j)\).

Whenever \(P_i\) receives a message from \(P_j\) and the time-stamp of the message (\(\text{PGC}(P_j)\) at the time message was sent) is older than \(\text{PGC}(P_i)\), then the message is outdated and can be ignored by \(P_i\). Otherwise, if \(\text{PGC}(P_j)\) is more-recent than \(\text{PGC}(P_i)\), then \(P_i\) updates its \(\text{PGC}(P_i)\) by first substituting the first \(j\) components of \(\text{PGC}(P_i)\) with \(\text{PGC}(P_j)\) and then resetting components \(j+1\) to \(i-1\) to 0. \(P_i\) then throws away all the outdated bindings and tokens it has previously received. Note that if \(\text{PGC}(P_j)\) is more-recent than \(\text{PGC}(P_i)\), then only those bindings and tokens are outdated which \(P_i\) has previously received from any process \(P_m\) where \(m \geq k\) and \(k =\)

---

5 In this case, \(j\) is always smaller than \(i\). The reason is that the only message \(P_i\) can receive from any \(P_j\) (where \(j > i\)) is a REDO message, and such a message can never have time-stamp that is more-recent than the clock of \(P_i\).
\( \min \{x \mid \text{LC}(P_x) \text{ in } \text{PGC}(P_j) \text{ is different from } \text{LC}(P_x) \text{ in } \text{PGC}(P_i), 1 \leq x \leq \min(i,j) \} \). Therefore \( P_i \) only has to keep the sender of bindings and tokens it is consuming, but not the time-stamps of these bindings and tokens. The following describes how each process responds to different kinds of messages received during the execution.

**\( P_i \) receives bindings (and tokens) from \( P_j \).**

If \( \text{PGC}(P_j) \), sent along with the bindings, is older than \( \text{PGC}(P_i) \), then the message is outdated and can be ignored. Otherwise, \( P_i \) keeps the bindings and tokens, and does the following:

1. If \( \text{PGC}(P_j) \) is more-recent than \( \text{PGC}(P_i) \), then \( P_i \) resets \( \text{LC}(P_i) \) to 0, updates \( \text{PGC}(P_i) \), and throws away outdated bindings and tokens.
2. Reset B-list(\( P_i \)) to contain only the senders of those bindings that are still consumed by \( P_i \).
3. If \( P_i \) has tokens for all of its un uninstantiated variables, then it goes to the EXECUTING state. Otherwise it goes to the GATHER state.

**\( P_i \) receives a REDO(Y) message from \( P_j \), where Y= \text{tail}(\text{B-list}(P_j)).**

If \( P_i \) is the head literal, the execution of the clause fails. If \( \text{PGC}(P_j) \), sent along with REDO message, is older than \( \text{PGC}(P_i) \), then the REDO is ignored. Otherwise, \( P_i \), being in the SOLVED state, does the following:

1. Merge \( Y \) to B-list(\( P_i \)). Remove duplicate literals from the new B-list(\( P_i \)).
2. Increment \( \text{LC}(P_i) \) by one, and send INFORM messages along with its current \( \text{PGC}(P_i) \) to all the literals \( P_k \), where \( k > i \).
3. Switch to the EXECUTING state, and resume execution to find a new set of bindings for the variables it generates.

**\( P_i \) receives an INFORM message from \( P_j \).**

If \( \text{PGC}(P_j) \), sent along with the message, is not more-recent than \( \text{PGC}(P_i) \), then the message is ignored. Otherwise \( P_i \) does the following:

1. Reset \( \text{LC}(P_i) \) to 0, update \( \text{PGC}(P_i) \), and throws away outdated bindings and tokens.
2. Reset B-list(\( P_i \)) to contain only the senders of those bindings that are still consumed by \( P_i \).
3. If \( P_i \) has tokens for all of its uninstantiated variables, then it goes to the EXECUTING state. Otherwise it goes to the GATHER state.

Note that in this implementation, an INFORM message can have the effect of either "cancel" or "reset", depending on whether some bindings were thrown away while \( P_i \) was processing that INFORM message. Newly arrived bindings and tokens can also have the effect of cancel if \( P_i \) has to throw away bindings and tokens which were received previously. Also note that if
LC(\(P_i\)) = 0 before INFORM message arrives, and if the arrival of INFORM message does not result in discarding any existing bindings, then \(P_i\) does not have to change its state. That is, if \(P_i\) is in the EXECUTING state or the SOLVED state, then it could stay in that state, as by moving to the beginning of the EXECUTING state, \(P_i\) is guaranteed to repeat the previous computation. If \(P_i\) is in the SOLVED state and has sent some bindings to other literals, then it must re-send these new bindings, as the previous version could be considered outdated by the consumer literals. Hence the third step performed by \(P_i\) after receiving an INFORM message (that is more-recent than PGC(\(P_i\))) can be rewritten as follows:

3. If \(P_i\) does not have tokens for all of its uninstantiated variables, then \(P_i\) goes to the GATHER state.
   Otherwise, \((P_i\) must be in the EXECUTING or SOLVED state before the arrival of INFORM message)
   if \(LC(\(P_i\)) = 0\) before the arrival of INFORM message,
   then \(P_i\) does not change its execution status.
   If \(P_i\) was in the SOLVED state before receiving the INFORM message, then \(P_i\) re-sends bindings and tokens to its children with the new PGC(\(P_i\)).
   else \(P_i\) resets itself, i.e., it goes to the EXECUTING state and resumes execution from the beginning, as if it has just made the transition from the GATHER state for the first time.

Clearly the invariant of Section 3.2.1 is satisfied, as after \(P_i\) is canceled (by receiving either an INFORM message or a more-recent binging) due to the redo operation on \(P_j\), all the bindings received by \(P_i\) from the literals \(P_k\) \((k > j)\) will become outdated if \(P_k\) had not been appropriately canceled or reset before sending the bindings. This implementation also handles multiple failures naturally. See Appendix E for an illustration of this implementation.

4.2. The Centralized Implementation

In this implementation, the head literal \(P_0\) serves as a coordinator for synchronizing the execution of literals \(P_i\) in the clause body. \(P_0\) maintains all the information about \(P_i\) (such as the state of \(P_i\), B-list(\(P_i\)), tokens residing at \(P_i\)). \(P_0\) also keeps a common binding environment for
all the literals in the clause. A directed data-dependency graph, uniquely determined by the
tokens residing at various literals, can be constructed as follows: if a literal $P_i$ is in the SOLVED
state, then for every token F-list(V) residing at $P_i$, all literals in F-list(V) are children of $P_i$ in the
graph.

During the execution of the clause, $P_0$ may send any of the following messages to a literal
$P_i$: a START message is to initiate the execution of $P_i$; a REDO message is to ask $P_i$ to generate
a new set of bindings; a CANCEL message is to cancel the execution of $P_i$; and a RESET mes-
 sage is to restart the execution of $P_i$ from very beginning. If the execution of a literal $P_i$ fails, $P_i$
sends a FAIL message to $P_0$. Otherwise, $P_i$ sends a SUCCESS message to $P_0$.

Initially, all literals $P_i$ are in the GATHER state and B-list($P_i$) = []. After the head literal is
unified with the goal, $P_0$ creates the initial binding environment and runs Procedure $S_0$ to create
and distribute new tokens. If $P_j$ is in some token residing at $P_0$, then B-list($P_j$) = [$P_0$]. $P_0$ then
 repeats the following steps until all literals of the clause are in the SOLVED state.

1. For every literal $P_j$, if all of its predecessors in the data-dependency graph are in the
SOLVED state (equivalently, if $P_j$ has tokens for all the uninstantiated variables in its bind-
ing environment), then
   i) Advance $P_j$ to the EXECUTING state;
   ii) Create a process with a unique tag to solve $P_j$;
   iii) Send a START message to that process.
2. Wait until the process for any literal $P_i$, running in the EXECUTING state, sends back a
SUCCESS or FAIL message.
   If the process for $P_i$ sends back a SUCCESS message, then
   i) Change the state of $P_i$ to SOLVED.
   ii) If $P_i$ also sends back newly generated bindings, then
      a) Update binding environment of the clause;
      b) Run Procedure $S_0$ to create and distribute new tokens;
      c) Add $P_i$ to B-list($P_j$), if $P_j$ consumes bindings generated by $P_i$.
   Otherwise,
      i) Send REDO message to $P_k$ = head(B-list($P_i$)), and merge tail(B-list($P_i$)) to B-list($P_k$).
      ii) Determine the set of literals to be reset or canceled (based on the data-dependency
          graph corresponding to the current set of tokens), and send them RESET or CANCEL
          messages, respectively. Any further messages from canceled process are ignored.
iii) If \( P_m \) has been redone, canceled, or reset, then remove bindings produced by \( P_m \) from the common binding environment and eliminate tokens sent by \( P_m \) to other literals.

iv) If \( P_m \) has been reset or canceled, then reinitialize B-list(\( P_m \)) to contain only those literals \( P_l \) where \( P_l \) is in the SOLVED state and has at least one token which contains \( P_m \).

4.3. Discussion

Comparing with the centralized implementation, the distributed implementation in general requires about the same number of communication messages. Deciding ordering between timestamps is similar in time complexity to deciding whether a message comes from a canceled process or currently valid process. However, in the distributed implementation, the task of checking the ordering and updating the execution status is distributed at various literals, which avoids the potential processing bottleneck that exists at the coordinator (process \( P_0 \)) of the centralized implementation. The mechanism for deciding which literals should be canceled or reset is also decentralized in the distributed implementation, whereas in the centralized implementation a sequential procedure has to be executed at \( P_0 \). Moreover, in the distributed implementation, multiple failures can be handled simultaneously (see snapshot <5> of Appendix E), whereas in the centralized implementation, they can be handled only one after another. Hence the distributed implementation increases the chance of by-passing failure branches in the search space.

The distributed implementation creates processes for every literal in the clause body all at once. If the execution of a clause should fail due to some uncurable failure in the upper part of corresponding data-dependency graph, then some processes created for the literals in the lower part of data-dependency graph are wasted. Furthermore, each process needs to maintain a separate (partial) binding environment, causing space overhead. On the other hand, in the centralized implementation a redo operation can result in complete elimination of certain processes, whereas in the distributed implementation a redo operation only causes elimination of outdated bindings at relevant processes. Thus in the distributed implementation, the process need to be created only once.
Overall it appears that if the underlying architecture has a large number of processors as well as a fast communication channel, then the distributed approach can be more suitable for exploiting AND-parallelism of logic programs.

5. Concluding Remarks

We have presented an execution model for exploiting AND-parallelism in logic programs which is a variant of the execution model of Conery and Kibler [6]. Compared with Conery and Kibler's approach, our scheme has less run-time overhead, and still achieves the same degree of AND-parallelism. Chang, et. al. [2] and Degroot [7] have also proposed schemes which reduce the run-time overhead presented in Conery and Kibler's scheme. However, these schemes tend to limit the degree of AND-parallelism that can be exploited.

Unlike Conery [6] and Chang's [2] approaches, our execution model does not require an explicit data-dependency graph to select generators and consumers for various literals and to find which literals can be executed. Instead, each literal, by using the information it receives from other literals, can decide whether it is executable. Actually a dynamic data-dependency graph is maintained implicitly by our forward and backward execution algorithm. This graph grows when new bindings and tokens are created in the forward execution, and shrinks when outdated bindings and tokens are discarded in the backward execution.

As discussed in Section 4, our execution model can be implemented both in centralized and distributed manner. Conery and Kibler's execution model can only be implemented in a centralized manner, as the ordering algorithm used for creating a data-dependency graph has to be executed by a central process. To be able to run the ordering algorithm, the central process also needs to keep track of the bindings of all the variables of the clause. Furthermore, all the fail, cancel, reset, redo messages as well as new bindings are channeled through the central process. This may create a bottleneck at the processor executing the central process and hence limit parallelism. In the distributed implementation of our execution model, process executing different
literals of the clause together share the burden of receiving and sending messages. The number of messages generated in both approaches appear to be about the same.

The backward execution of our model is intelligent in the sense that if a literal $P_i$ fails, then it backtracks to a literal $P_j$ such that $P_j$ may be able to cure the failure of $P_i$. In a naive backtracking scheme, such as used in PROLOG, the failure of $P_i$ would always cause backtracking to $P_{i-1}$ even if $P_{i-1}$ is completely unrelated to $P_i$. Furthermore, our backward execution algorithm is guaranteed to not miss any solution while backtracking.

We are currently investigating various ways of mapping our execution model to a physical architecture.
Appendix A

The Ordering Algorithm

At compile time we use problem-specific information to construct linear sequences of literals in the body, one for each clause. Rules for ordering literals can be classified into four levels by their strength of strictness.

1. **Variable annotations**  
   When two or more literals share variable $V$, literals with read-only annotated $V$ should always appear after other literals where $V$ is not annotated.

2. **Programmer-suggested ordering**  
   Using domain knowledge, the programmer may be able to suggest a good order for solving literals in a conjunct. This kind of information could also be generated via simulation.

3. **Heuristic rules**  
   Some heuristics such as the connection rule or the number of unbound variables, as described in Conery's dissertation [5] can suggest ordering between literals.

4. **Default choice**  
   The left to right ordering of the literals in a clause provides the default ordering.

If we consider each literal as a node, then the ordering algorithm is just adding directional arcs from one node to the other. The final acyclic graph automatically represents a total ordering on the literals. The algorithm is as follows:

1. For each partial order $Pi < Pj$ specified in level 1, add an arc from $Pi$ to $Pj$.
2. For $l = 2$ to 3, add arcs in turn according to the partial orders specified in level $l$ as follows: for each partial order $Pi < Pj$, add an arc from $Pi$ to $Pj$ if adding this arc does not form a cycle in the existing graph.
3. For any two literals $Pi$ and $Pj$, if $Pi$ appears before $Pj$ in the default ordering, then add an arc from $Pi$ to $Pj$ if adding this arc does not form a cycle in the existing graph.

Note that the ordering of literals may form a cycle due to the annotations of different shared variables. We assume that this will not happen. An example of ordering literals in the body of a given clause:

is shown in the following. If the cycle situation should happen in Step 1, a more complicated algorithm should be applied at run time (prior to the execution of literals in the clause body) to generate a linear sequence based on the bindings generated through unification.

Suppose we are applying ordering algorithm on the following clause

\[ p_0(X,Y,Z) : p_1(X,W), p_2(Y,W), p_3(W?,Z), p_4(Z), p_5(Y). \]

Partial orders in level 1 are \( p_1 < p_3 \) and \( p_2 < p_3 \)
Partial orders in level 2 are \( p_2 < p_5 \) and \( p_2 < p_1 \)
Partial orders in level 3 are \( p_4 < p_2 \) and \( p_5 < p_2 \)
Partial orders in level 4 are \( p_1 < p_2 < p_3 < p_4 < p_5 \)

(W is an input variable of \( p_3 \).
(user suggested).
(heuristic rule: number of variables).
(left to right).

1. Initial
   \[
   \begin{array}{ll}
   p_1(X,W) & p_2(Y,W) \\
   p_5(Y) & p_3(W?,Z) \\
   p_4(Z) &
   \end{array}
   \]

2. Add arcs for \( p_1 < p_3 \) and \( p_2 < p_3 \) in level 1
   \[
   \begin{array}{ll}
   p_1(X,W) & p_2(Y,W) \\
   p_5(Y) & p_3(W?,Z) \\
   p_4(Z) &
   \end{array}
   \]

3a. Add arc for \( p_2 < p_5 \) in level 2
   \[
   \begin{array}{ll}
   p_1(X,W) & p_2(Y,W) \\
   p_5(Y) & p_3(W?,Z) \\
   p_4(Z) &
   \end{array}
   \]

3b. Add arc for \( p_2 < p_1 \) in level 2
   \[
   \begin{array}{ll}
   p_1(X,W) & p_2(Y,W) \\
   p_5(Y) & p_3(W?,Z) \\
   p_4(Z) &
   \end{array}
   \]

4. Add arc for \( p_4 < p_3 \) in level 3.
   Arc for \( p_5 < p_2 \) is discarded.
   \[
   \begin{array}{ll}
   p_1(X,W) & p_2(Y,W) \\
   p_5(Y) & p_3(W?,Z) \\
   p_4(Z) &
   \end{array}
   \]

5. Add arcs according to Step 3 of the algorithm
   \[
   \begin{array}{ll}
   p_1(X,W) & p_2(Y,W) \\
   p_5(Y) & p_3(W?,Z) \\
   p_4(Z) &
   \end{array}
   \]

6. The total ordering
   \[
   \begin{array}{ll}
   p_2(Y,W) & p_1(X,W) & p_4(Z) & p_3(W?,Z) & p_5(Y) \\
   \end{array}
   \]
Appendix B

Suppose the following clause is given:

\[ p_0(X,Y,Z) :- p_1(X), p_2(Y), p_3(Z). \]

If the set of possible activation modes of \( p_0 \) contains only \( p_0(NGI,NGD,NGD) \), \( p_0(NGD,NGD,G) \), \( p_0(NGI,G,NGI) \), and \( p_0(G,G,G) \), then in some moment at run time at least two of those literals in the clause body can be executed in parallel. However, the worst activation mode of \( p_0 \), according to the scheme of Chang, et.al., is \( p_0(NGD,NGD,NGD) \). If the exit mode of \( p_1 \), with respect to an activation mode \( p_1(NGI) \), is \( p_1(NGI) \), then the static data-dependency graph becomes a linear list. This means the given clause can only be run sequentially.
Appendix C

At the time a literal $P_i$ fails, let $X_i = \{ L | L$ is a predecessor of $P_i \} \cup \{ L | L$ is a predecessor of $M$ and the failure of $M$ has directly or indirectly caused a redo operation on $P_i$ since $P_i$ was most recently changed from the GATHER state to the EXECUTING state $\}$. Clearly, the only way to correct the current failure of $P_i$ is to re-solve some literals in $X_i$. But it is not clear beforehand as to which literals in $X_i$ should be re-solved to make $P_i$ succeed again. To ensure that all possible combinations of the bindings generated by the literals in $X_i$ can be tried, we make use of the linear ordering on the literals. If $P_i$ fails, then we always backtrack to a literal $P_m$ such that $m = \max \{ k | P_k$ can possibly cure the failure of $P_i$ $\}$. This is reminiscent of the nested loop model discussed in [6]. The following lemma says that $\{ P_m | P_m \in X_i \text{ and } m < i \}$ is precisely the set of literals which can possibly cure the failure of $P_i$.

**Lemma 1.** If $P_i$ fails, then $P_m$ can possibly cure the failure of $P_i$ if and only if $P_m \in X_i$ and $m < i$.

**Proof:** The if part is trivial. Let us focus on the other direction. Assume that literals are divided into two sets. Set $S_1$ consists of every literal which has directly or indirectly caused the failure of $P_i$; and set $S_2$ consists of the rest of the literals. If $P_m$ is not a member of $X_i$, then re-solving $P_m$ will not affect the success or failure of any literal in $S_1$. Therefore $P_i$ would keep failing for exactly the same reasons for which literals in $S_1$ have caused its failure before. Hence $P_m$ must be a member of $X_i$.

Assume that $m > i$. Then $P_m$ is not a predecessor of $P_i$. Let $P_m$ be a predecessor of $P_i$, where the failure of $P_i$ has directly or indirectly caused the failure of $P_i$. In order to possibly cure the failure of $P_i$, $P_m$ should be able to prevent $P_j$ from causing the failure of $P_i$. However $P_m$ should have failed to do so, otherwise $P_j$ would not have caused the failure of $P_i$.

Let $bi = \max \{ k | P_k \in X_i \text{ and } k < i \}$. From Lemma 1 and the preceding discussion, it is clear that the failure of $P_i$ should result in backtracking to $P_{bi}$. The following lemmas help us prove that our algorithm does precisely that.

---

6 Failure of M indirectly causes a redo operation on $P_i$ if failure of L directly causes a redo operation on $P_i$, and the failure of M directly or indirectly causes a redo operation on L.
Let $S_i = \{L|L \text{ is a parent of } P_i\} \cup \{L|L \text{ is a parent of } M \text{ and the failure of } M \text{ has directly or indirectly caused a redo operation on } P_i \text{ since } P_i \text{ was most recently changed from the GATHER state to the EXECUTING state}\}. \text{ Lemma 2 says that } b_i \text{ is equal to } \max\{k|P_k \in S_i \text{ and } k < i\}.

**Lemma 2.** $b_i = \max\{k|P_k \in S_i \text{ and } k < i\}$.

**Proof:** First we prove that $P_{bi} \in S_i$. By definition, the set $S_i$ is the union of the parents of $P_i$, and those literals which are parents of some $P_j$ such $P_j$ has directly or indirectly caused the failure of $P_i$. Suppose $P_{bi}$ is not a parent of any such $P_j$, and $P_{bi}$ is not a parent of $P_i$. Then since $P_{bi} \in X_i$, $P_{bi}$ must be a predecessor of some $P_m$ which is either a parent of a certain $P_j$, or a parent of $P_i$. Therefore $P_m$ can possibly cure the failure of $P_i$. Hence by Lemma 1, $P_m \in X_i$ and $m < i$. However, $b_i < m$ because $P_{bi}$ is a predecessor of $P_m$. Thus, there is a $P_m \in X_i$ such that $b_i < m < i$, which is contrary to the definition of $b_i$.

Now we prove that $b_i = \max\{k|P_k \in S_i \text{ and } k < i\}$. From the definition of $X_i$ and $S_i$, it follows that $\{P_k|P_k \in S_i \text{ and } k < i\} \subseteq \{P_k|P_k \in X_i \text{ and } k < i\}$. Hence $b_i = \max\{k|P_k \in X_i \text{ and } k < i\} \geq \max\{k|P_k \in S_i \text{ and } k < i\}$. Therefore $b_i < k < i$; hence $b_i \leq \max\{k|P_k \in S_i \text{ and } k < i\}$. It follows that $b_i = \max\{k|P_k \in S_i \text{ and } k < i\}$.

**Lemma 3.** At the time $P_i$ fails, $\{P_j|P_j \in B\text{-list}(P_i)\} = \{P_j|P_j \in S_i \text{ and } j < i\}$.

**Proof:** By induction on the failure level of the failed literal. The failure level of a literal $L$ is defined as follows.

1. If no redo operations have been done on $L$ since the most recent instant when $L$ was changed from the GATHER state to the EXECUTING state, then the failure level of $L$ is $0$.
2. Otherwise, the failure level of $L = n+1$, where $n = \max\{|f(Q)| \text{ failure of } Q \text{ has directly invoked a redo operation on } L \text{ since } L \text{ started executing most recently, and } f(Q) \text{ was the failure level of } Q \text{ at the time } Q \text{ failed}\}$.

**Base Case:** Failure level of $P_i = 0$.

Since no redo operation has been done on $P_i$, $\{P_j|P_j \in B\text{-list}(P_i)\} = S_i = \{P_j|P_j \text{ is a parent of } P_i\}$. Furthermore, each $j$ is smaller than $i$, as $P_j$ is a predecessor of $P_i$.

**Induction Step:** Suppose the theorem holds for every failed literal whose failure level is less than or equal to $r$, and let $P_i$ is a failed literal whose failure level is $r+1$.

From the time $P_i$ was most recently transferred to the EXECUTING state, assume that the failure of $P_{i1}, \ldots, P_{ik}, \ldots, P_{in}$ have directly invoked redo operations on $P_i$. Note that each $P_{ik} (1 \leq k \leq n)$ is either a child of $P_i$ or a predecessor of a successor of $P_i$ such that $ik > i$. By definition $S_i$ is the union of $S_{ik}, ik = 1$ to $in$, and the set of parents of $P_i$. Therefore, any literal $P_j$ in $\{P_j|P_j \in S_i \text{ and } j < i\}$ should be either a parent of $P_i$ or in $S_{ik}$ for some $1 \leq k \leq n$. If $P_j$ is a parent of $P_i$, then $P_j$ is surely in $B\text{-list}(P_i)$. If $P_j$ is not a parent of $P_i$, then (because $j < i < ik$) $P_j$ should appear in $B\text{-list}(P_{ik})$ by the induction hypothesis. Hence, it would have been merged into $B\text{-list}(P_i)$ by our backward execution algorithm while handling the failure of $P_{ik}$. Therefore $B\text{-list}(P_i)$ contains every literal in $\{P_j|P_j \in S_i \text{ and } j < i\}$.
If \( P_m \) is in B-list\( (P_i) \), then either \( P_m \) is a parent of \( P_i \) or \( P_m \) is in B-list\( (P_{ik}) \) for some \( ik \) (and was merged into B-list\( (P_i) \) after the failure of \( P_{ik} \)). If \( P_m \) is a parent of \( P_i \), then clearly \( P_m \in S_i \) and \( m < i \). Otherwise, \( P_m \) is in B-list\( (P_{ik}) \), and is also in \( S_{ik} \) (and therefore in \( S_i \)) by the induction hypothesis. Furthermore, it must be true that \( m < i \), as otherwise in our backward execution algorithm the failure of \( P_{ik} \) would have caused a redo operation on \( P_m \) instead of \( P_i \). This proves that \( \{ P_j \mid P_j \in S_i \text{ and } j < i \} \) contains every literal in B-list\( (P_i) \).

\( \square \)

Note that the B-list of each literal is always ordered such that if B-list\( (P_i) = [P_j \mid Y] \), then \( j = \max \{ k \mid P_k \in \text{B-list}(P_i) \} \). Hence, from Lemma 2 and Lemma 3 it follows that in our algorithm, the failure of \( P_i \) causes backtracking to \( P_{bi} \).
Appendix D

The following is an iterative procedure that can be used to find all literals $P_k$ that need to be reset. Suppose that $P_j$ is asked to redo.

1. Let CS contains literals which should be canceled, and RS contains literals which should be reset. Initially, CS = $\{P_m | P_m$ is a successor of $P_j\}$ and RS = $\{\}$.
2. Let GS = $\{P_k | k > j$, $P_k$ has children, and $P_k$ is not a child of any literal $P_s$ ($s > j$)$\}$. Let GS' = $\{\}$.
3. While GS is not empty
   a. Remove the leftmost literal $P_l$ from GS and then do one of the following:
      (1) If none of the successors of $P_l$ is in CS, then put $P_l$ in GS'.
      (2) If some of the successors of $P_l$ is in CS, and $P_l$ has generated only one set of bindings, then for every child $P_m$ of $P_l$, if $P_m$ is not in CS, put $P_m$ in GS.
      (3) If some of the successors of $P_l$ is in CS, and $P_l$ has generated more than one set of bindings, then
         i) Put $P_l$ in RS;
         ii) Remove from GS any literal which is a successor of $P_l$;
         iii) Add to CS all the successors of $P_l$ which was not in CS previously;
         iv) If any literal has been added to CS in step iii), then move every literal in GS' back to GS.

When the procedure terminates, RS contains all the literal which should be reset, and CS contains literals whose bindings must be canceled.
Appendix E

The following is an example of how the distributed implementation of our execution model works.

Suppose that we are solving $P_0(X,Y)$ with the following set of clauses.

$$P_0(A,B) \leftarrow P_1(A), P_2(B), P_3(A,C), P_4(A,B,C), P_5(B).$$

$P_1(a1).$

$P_1(a2).$

$P_2(b1).$

$P_2(b2).$

$P_3(a2, c1).$

$P_4(a2, b2, c1).$

$P_5(b2).$

In each table, column Received refers to the bindings and tokens a literal has received from other literals. VN stands for variable name. If subcolumn From is empty, it means the token is received at the time of process creation. Literal $P^4$ is the dummy literal mentioned in Section 3.3. Messages are represented in the format [sender, content, time-stamp].

Suppose that the head literal has unified with the given goal, and the processes for all the literals in the clause body have been created.
\[<1>\] \(P_1\) and \(P_2\) have started executing after having received tokens and bindings from \(P_0\). Suppose \(P_1\) has succeeded and sent binding \(a_1/X\) to \(P_3\), \(P_4\), and \(P^d\) by sending message \([P_1, a_1/X, 0]\) to them. The table shows the status after \(P_3\) has received binding from \(P_1\) and started executing. The following messages are still on the way to the destination.

\([P_1, a_1/X, 0]\) to \(P_4\).
\([P_1, a_1/X, 0]\) to \(P^d\).

<table>
<thead>
<tr>
<th>Literal</th>
<th>State</th>
<th>PGC</th>
<th>B-list</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1(A))</td>
<td>SOLVED</td>
<td>0</td>
<td>([P_0])</td>
<td>(P_0) A X (P_0) X (P_5, P_4, P^d)</td>
</tr>
<tr>
<td>(P_2(B))</td>
<td>EXECUTING</td>
<td>0:0</td>
<td>([P_0])</td>
<td>(P_0) B Y (P_0) Y (P_4, P_5, P^d)</td>
</tr>
<tr>
<td>(P_3(A,C))</td>
<td>EXECUTING</td>
<td>0:0:0</td>
<td>([P_1, P_0])</td>
<td>(P_0) A X (P_1) X a1 (C) ([P_2])</td>
</tr>
<tr>
<td>(P_4(A,B,C))</td>
<td>GATHER</td>
<td>0:0:0:0</td>
<td>([P_0])</td>
<td>(P_0) A X (P_0) B Y (P_0) B Y</td>
</tr>
<tr>
<td>(P_5(C))</td>
<td>GATHER</td>
<td>0:0:0:0:0</td>
<td>([P_0])</td>
<td>(P_0) A X (P_0) B Y</td>
</tr>
<tr>
<td>(P^d(A,B))</td>
<td>GATHER</td>
<td>0:0:0:0:0:0</td>
<td>([P_0])</td>
<td>(P_0) A X (P_0) B Y</td>
</tr>
</tbody>
</table>
$P_3$ fails. It sends a REDO message $[P_3, \text{REDO}([P_0]), 0-0]$ to $P_1 = \text{head(B-list}(P_3))$.
Suppose $P_1$ has received $[P_3, \text{REDO}([P_0]), 0-0]$ and sent INFORM messages $[P_1, \text{INFORM, 1}]$ to $P_2, P_3, P_4, P_5, \text{and } P^d$.
The table shows the status after $P_2$ and $P_3$ have received and processed $[P_1, \text{INFORM, 1}]$.
The following messages are still on the way to the destination.

$[P_1, a1/X, 0]$ and $[P_1, \text{INFORM, 1}]$ to $P_4$.
$[P_1, \text{INFORM, 1}]$ to $P_5$.
$[P_1, a1/X, 0]$ and $[P_1, \text{INFORM, 1}]$ to $P^d$.

<table>
<thead>
<tr>
<th>Literal</th>
<th>State</th>
<th>PGC</th>
<th>B-list</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1(A)$</td>
<td>EXECUTING</td>
<td>1</td>
<td>$[P_0]$</td>
<td>$P_0 \ A \ X$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_0 \ X$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$[P_3, P_4, P^d]$</td>
</tr>
<tr>
<td>$P_2(B)$</td>
<td>EXECUTING</td>
<td>1-0</td>
<td>$[P_0]$</td>
<td>$P_0 \ B \ Y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_0 \ Y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$[P_4, P_5, P^d]$</td>
</tr>
<tr>
<td>$P_3(A,C)$</td>
<td>GATHER</td>
<td>1-0-0</td>
<td>$[P_0]$</td>
<td>$P_0 \ A \ X$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$C \ [P_4]$</td>
</tr>
<tr>
<td>$P_4(A,B,C)$</td>
<td>GATHER</td>
<td>0-0-0-0</td>
<td>$[P_0]$</td>
<td>$P_0 \ A \ X$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_0 \ B \ Y$</td>
</tr>
<tr>
<td>$P_5(B)$</td>
<td>GATHER</td>
<td>0-0-0-0-0</td>
<td>$[P_0]$</td>
<td>$P_0 \ B \ Y$</td>
</tr>
<tr>
<td>$P^d(A,B)$</td>
<td>GATHER</td>
<td>0-0-0-0-0-0</td>
<td>$[P_0]$</td>
<td>$P_0 \ A \ X$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_0 \ B \ Y$</td>
</tr>
</tbody>
</table>
Since the communication channels are not necessarily FIFO, \( P_4 \) and \( P^d \) can receive \([P_1, INFORM, 1]\) before receiving \([P_1, a1/X, 0]\).

The table shows the status after \( P_4, P_5, \) and \( P^d \) have processed \([P_1, INFORM, 1]\).

The following messages are still on the way to the destination.

\([P_1, a1/X, 0]\) to \( P_4 \).

\([P_1, a1/X, 0]\) to \( P^d \).

These two messages can be neglected from now on, as they will be outdated and ignored at the time they reach the destination.

<table>
<thead>
<tr>
<th>Literal</th>
<th>State</th>
<th>PGC</th>
<th>B-list</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>From</td>
</tr>
<tr>
<td>( P_1(A) )</td>
<td>EXECUTING</td>
<td>1</td>
<td>([P_0])</td>
<td>( P_0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( P_0 )</td>
</tr>
<tr>
<td>( P_2(B) )</td>
<td>EXECUTING</td>
<td>1-0</td>
<td>([P_0])</td>
<td>( P_0 )</td>
</tr>
<tr>
<td>( P_3(A,C) )</td>
<td>GATHER</td>
<td>1-0-0</td>
<td>([P_0])</td>
<td>( P_0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( P_0 )</td>
</tr>
<tr>
<td>( P_4(A,B,C) )</td>
<td>GATHER</td>
<td>1-0-0-0</td>
<td>([P_0])</td>
<td>( P_0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( P_0 )</td>
</tr>
<tr>
<td>( P_5(B) )</td>
<td>GATHER</td>
<td>1-0-0-0-0</td>
<td>([P_0])</td>
<td>( P_0 )</td>
</tr>
<tr>
<td>( P^d(A,B) )</td>
<td>GATHER</td>
<td>1-0-0-0-0-0</td>
<td>([P_0])</td>
<td>( P_0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( P_0 )</td>
</tr>
</tbody>
</table>
$P_1$ succeeds again, and sends $[P_1, a2/X, 1]$ to $P_3$, $P_4$, and $P^d$. $P_2$ also succeeds. It sends $[P_2, b1/Y, 1:0]$ to $P_4$, $P_5$, and $P^d$.

Suppose these messages have all been received, and $P_3$ has succeeded in its execution and sent $[P_3, c1/C, 1:0:0]$ to $P_4$. $P_3$ also sends $[P_3, SUCCESS, 1:0:0]$ to $P^d$.

The table shows the status after $P_4$ has received $[P_3, c1/C, 1:0:0]$ and advanced to the EXECUTING state.

No message is still on the way to the destination.

<table>
<thead>
<tr>
<th>Literal</th>
<th>State</th>
<th>PGC</th>
<th>B-list</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1(A)$</td>
<td>SOLVED</td>
<td>1</td>
<td>$[P_0]$</td>
<td>$P_0$ A X</td>
</tr>
<tr>
<td>$P_2(B)$</td>
<td>SOLVED</td>
<td>1:0</td>
<td>$[P_0]$</td>
<td>$P_0$ X $[P_3, P_4, P^d]$</td>
</tr>
<tr>
<td>$P_3(A,C)$</td>
<td>SOLVED</td>
<td>1:0:0</td>
<td>$[P_1, P_0]$</td>
<td>$P_0$ A X</td>
</tr>
<tr>
<td>$P_4(A,B,C)$</td>
<td>EXECUTING</td>
<td>1:0:0:0</td>
<td>$[P_3, P_2, P_1, P_0]$</td>
<td>$P_0$ B Y $P_2$ Y b1</td>
</tr>
<tr>
<td>$P_5(B)$</td>
<td>EXECUTING</td>
<td>1:0:0:0</td>
<td>$[P_2, P_0]$</td>
<td>$P_0$ B Y $P_2$ Y b1</td>
</tr>
<tr>
<td>$P^d(A,B)$</td>
<td>GATHER</td>
<td>1:0:0:0:0</td>
<td>$[P_2, P_1, P_0]$</td>
<td>$P_0$ A X $P_1$ X a2 $P_0$ B Y $P_2$ Y b1 $P_3$ SUCCESS</td>
</tr>
</tbody>
</table>
Both $P_4$ and $P_5$ fail. $P_4$ sends $[P_4, \text{REDO, 1\text{-}0\text{-}0\text{-}0\text{-}0}]$ to $P_3$. $P_5$ sends $[P_5, \text{REDO, 1\text{-}0\text{-}0\text{-}0\text{-}0}]$ to $P_2$.

Suppose $P_2$ has received $[P_5, \text{REDO, 1\text{-}0\text{-}0\text{-}0\text{-}0}]$ and sent $[P_2, \text{INFORM, 1\text{-}1}]$ to $P_3$, $P_4$, $P_5$, and $P^d$.

The table shows the status after $P_3$, $P_4$, $P_5$, and $P^d$ have all processed $[P_2, \text{INFORM, 1\text{-}1}]$. Note that both $P_4$ and $P_5$ are "canceled", as some of their bindings were thrown away while they were processing $[P_2, \text{INFORM, 1\text{-}1}]$. Also note that $P_3$ did not change its execution status. $P_3$ just updated PGC($P_3$) and re-sent bindings to its children with the new PGC($P_3$). Thus, $P_3$ sent $[P_3, \text{c1\text{/}C, 1\text{-}1\text{-}0}]$ to $P_4$ and $[P_3, \text{SUCCESS, 1\text{-}1\text{-}0}]$ to $P^d$.

The following messages are still on the way to the destination.

- $[P_4, \text{REDO, 1\text{-}0\text{-}0\text{-}0}]$ to $P_3$ (which will be ignored).
- $[P_3, \text{c1\text{/}C, 1\text{-}1\text{-}0}]$ to $P_4$.
- $[P_3, \text{SUCCESS, 1\text{-}1\text{-}0}]$ to $P^d$.

<table>
<thead>
<tr>
<th>Literal</th>
<th>State</th>
<th>PGC</th>
<th>B-list</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_1(A)$</td>
<td>SOLVED</td>
<td>1</td>
<td>$[P_0]$</td>
<td>$P_0$ A X $P_0$ X $[P_3, P_4, P^d]$</td>
</tr>
<tr>
<td>$P_2(B)$</td>
<td>EXECUTING</td>
<td>1:1</td>
<td>$[P_0]$</td>
<td>$P_0$ B Y $P_0$ Y $[P_4, P_5, P^d]$</td>
</tr>
<tr>
<td>$P_5(A,C)$</td>
<td>SOLVED</td>
<td>1:1:0</td>
<td>$[P_1, P_0]$</td>
<td>$P_0$ A X $P_1$ X a2 C $[P_4]$</td>
</tr>
<tr>
<td>$P_4(A,B,C)$</td>
<td>GATHER</td>
<td>1:1:0:0</td>
<td>$[P_1, P_0]$</td>
<td>$P_0$ A X $P_1$ X a2</td>
</tr>
<tr>
<td>$P_3(B)$</td>
<td>GATHER</td>
<td>1:1:0:0:0</td>
<td>$[P_0]$</td>
<td>$P_0$ B Y</td>
</tr>
<tr>
<td>$P^d(A,B)$</td>
<td>GATHER</td>
<td>1:1:0:0:0:0</td>
<td>$[P_1, P_0]$</td>
<td>$P_0$ A X $P_1$ X a2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
P2 succeeds again, and sends \([P_2, b2/Y, 1\cdot1]\) to \(P_4, P_5,\) and \(P^d\). After receiving \([P_2, b2/Y, 1\cdot1]\), \(P_4\) and \(P_5\) can start executing.

\(P_4\) succeeds and sends \([P_4,\SUCCESS, 1\cdot1\cdot0\cdot0]\) to \(P^d\). \(P_5\) also succeeds and sends \([P_5, \SUCCESS, 1\cdot1\cdot0\cdot0\cdot0]\) to \(P^d\).

The table shows the status after \(P^d\) has received a (not outdated) \SUCCESS message from every literal.

<table>
<thead>
<tr>
<th>Literal</th>
<th>State</th>
<th>PGC</th>
<th>B-list</th>
<th>Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_3(A))</td>
<td>SOLVED</td>
<td>1</td>
<td>([P_0])</td>
<td></td>
</tr>
<tr>
<td>(P_3(B))</td>
<td>SOLVED</td>
<td>1\cdot1</td>
<td>([P_0])</td>
<td></td>
</tr>
<tr>
<td>(P_3(A, C))</td>
<td>SOLVED</td>
<td>1\cdot0</td>
<td>([P_1, P_0])</td>
<td></td>
</tr>
<tr>
<td>(P_4(A, B, C))</td>
<td>SOLVED</td>
<td>1\cdot1\cdot0</td>
<td>([P_3, P_2, P_1, P_0])</td>
<td></td>
</tr>
<tr>
<td>(P_5(B))</td>
<td>SOLVED</td>
<td>1\cdot1\cdot0\cdot0</td>
<td>([P_2, P_0])</td>
<td></td>
</tr>
<tr>
<td>(P^d(A, B))</td>
<td>SOLVED</td>
<td>1\cdot1\cdot0\cdot0\cdot0</td>
<td>([P_2, P_1, P_0])</td>
<td></td>
</tr>
</tbody>
</table>


[14] Woo, Nam S. and Kwang-Moo Choe, Selecting the Backtrack Literal in the AND
Figure 1. An example of the forward execution algorithm
Figure 2. The static data-dependency graph used by the scheme of Chang, et. al.

Figure 3. The dynamic data-dependency graph achieved by Degroot’s RAP

Figure 4. The resulting data-dependency graph of Conery and Kibler’s scheme
(a) Before the failure of \(P_5\)

(b) After backtracking has occurred due to the failure of \(P_5\)

Figure 5. An example of the backward execution algorithm
(c) Before the failure of $p_4$

(d) After backtracking has occurred due to the failure of $p_4$

(e) After backtracking has occurred due to the failure of $p_3$

Figure 5. An example of the backward execution algorithm (continued)