An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs

Yow-Jian Lin
Vipin Kumar
Clement Leung

AI TR86-22 **March 1986**

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.

An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs

Yow-Jian Lin
Vipin Kumar
Clement Leung

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

ABSTRACT

In this paper we present a simple but efficient backtracking scheme which works when AND-parallelism is exploited in a logic program. The scheme is well suited for implementation on a parallel hardware. We show that the backtracking scheme presented by Conery & Kibler in the context of AND/OR process model is incorrect, i.e., in some cases it fails to find a solution when a solution exists. Even if no AND-parallelism is exploited (i.e., all literals are solved sequentially), our scheme is more efficient than the "naive" depth-first backtracking strategy used by Prolog because our scheme makes use of the dependencies between literals in a clause. Chang & Despain have recently presented a backtracking scheme which also makes use of the dependencies between literals. We show that our scheme is more efficient than their scheme in the sense that it does less backtracking.

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.

An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs

Yow-Jian Lin
Vipin Kumar
Clement Leung

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

1. Introduction

AND-parallelism in logic programs refers to executing more than one literal of a clause at the same time. When the literals in a clause don't share any uninstantiated variable, the literals can be solved independently. But if more than one literal share uninstantiated variables, then solving these literals independently could lead to excessive computation. Various techniques have been developed to exploit AND-parallelism when literals in a clause are dependent (i.e., they share uninstantiated variables) [2], [5], [7], [8]. One technique, introduced by Conery & Kibler in the context of AND/OR process model, is to execute only one of the dependent literals to find a ground value for the shared variable, and then execute the remaining literals in parallel if they do not share any other variable [3], [5]. Due to the nondeterministic nature of logic programs, a literal may fail because the variable bindings it has received from other literals are not satisfactory. In this case, we have to re-solve some literals and generate new bindings. For example, in solving a goal statement such as

← p(X), q(Y), r(X,Y).

if r(X,Y) does not satisfy the bindings received from both p(X) and q(Y), then either p(X) or q(Y) will have to generate a new binding. Conery & Kibler have presented a backtracking scheme, called "backward execution algorithm" in [5], for deciding which literals to re-solve in case some literals fail. We found that their backward execution algorithm is incorrect, i.e., in some cases, it fails to find a solution when a solution exists. In this paper we present a simple but efficient backtracking scheme, and prove that it is correct. Even if no AND-parallelism is exploited (i.e., all literals are solved sequentially), our scheme is more efficient than the "naive" depth-first backtracking strategy used by Prolog because our scheme makes use of the dependencies between literals in a clause. Chang & Despain have recently presented a backtracking scheme which also makes use of the dependencies between literals [2]. Compared to their scheme, our scheme is more efficient in the sense that it does less backtracking.

2. Background and Definitions

The discussion in this paper is based upon an abstract model similar to the AND/OR process of Conery & Kibler [3], [5]. We assume that we are given a clause body (with some variables instantiated because of unification on the head). If more than one literal in the clause body share an uninstantiated variable v, then we designate one of these literals as the *generator* of v, and the remaining literals as the *consumers* of v. A

literal can be the generator of some variables, and at the same time, it can be a consumer of some other variables.

The generator-consumer relationship between the literals of a clause can be depicted via a data dependency graph D. D is a directed graph in which a literal L is depicted as an immediate successor of a literal M if for some variable v, M is the generator of v and L is a consumer of v. We use the term "X is a D-predecessor of Y" to mean that "X is a predecessor of Y in D". Similarly, the term "X is a D-successor of Y" means that "X is a successor of Y in D". Note that if the execution of M does not bind v to a ground term, then the successors of M will have to be reordered such that one of them is the generator of variables in the new binding, and the remaining ones are consumer. To simplify the discussion in this paper we assume that the execution of M always binds v to a ground term (so that all consumers of v can be executed in parallel).

We also construct a linear ordering for the literals in the clause body. The reason for this will become clear in section 5. There is no restriction on the relative order of any two literals except that a generator must come before all the literals that consume the variable bindings it generates. One way to construct the ordering is to traverse the data dependency graph D in a breadth-first manner. From now on P_i is used to refer to the i-th literal in the linear ordering.

To exploit AND-parallelism, each literal is executed by a different process. The process executing a literal can be in one of the three modes: GATHER, EXECUTION and FINISHED. While in the EXECUTION mode, a process can be in one of the two states: SOLVED and UNSOLVED. For brevity, we will often use "a literal L" to mean that "the process corresponding to L". Literals (i.e., the corresponding processes) can communicate with each other via exchanging messages.

Each literal L also maintains a list of literals called *B-list* (denoted as B-list(L)) which is used to find a possible backtrack point when the literal fails. The literals on each B-list are sorted according to the linear ordering, with literals which are earlier in the linear ordering occurring later on the B-list.

The following "forward execution" algorithm determines the state, mode and message pattern of the processes.

3. The Forward Execution Algorithm

For all literals L such that L has no predecessors in the data dependency graph D, start L in the UNSOLVED state of the EXECUTION mode. Start the remaining literals in the GATHER mode. Literals in the UNSOLVED state of the EXECUTION mode can start executing.

When a literal L, running in the UNSOLVED state of the EXECUTION mode, finds a combination of values for the variables it is generating, it then goes from the UNSOLVED state to the SOLVED state, and sends the values of the generated variables to the immediate D-successors. If L has no successors in D, then it goes to the FINISHED mode. In either case, L suspends its execution[1].

[1] Note that L could have continued to execute to find its next solution. This is called OR-parallelism by Conery [3], [4], [5]. To simplify discussion in this paper we assume that no OR-parallelism is being exploited. Our backward execution scheme works even when OR-parallelism is exploited (see Lin's thesis [9] for details).

If a literal L is in GATHER mode and receives values for all the variables it is consuming, then it goes to the UNSOLVED state of the EXECUTION mode and its B-list is initialized to the list of its immediate D-predecessors. At this point we can start executing L.

If a literal has successors in D and all of its immediate D-successors are in the FINISHED mode, then it goes to the FINISHED mode. If all the literals of the clause are in the FINISHED mode, then the execution of the clause terminates. Figure 1 shows the forward execution algorithm in a graphical form.

If a literal P_i fails (i.e., when P_i can find no more solutions while executing in the UNSOLVED state of the EXECUTION mode), then the following "backward execution" algorithm is executed.

4. The Backward Execution Algorithm

If the B-list of P_i is a null list, then the execution of the clause fails. If the B-list of P_i is a nonempty list $[P_j \mid Y]$, then backtracking is done to P_j by performing the following steps.

1. Cancel all the successors of P_j in the data dependency graph D.
2. Reset every generator P_k, where $k > j$, and P_k has not been canceled in Step 1.
3. Merge [Y] to the B-list of P_j. Remove duplicate literals from the new B-list of P_j.
4. Perform redo operation on P_j.
5. If a literal M is in the FINISHED mode and any of its D-successors is no longer in the FINISHED mode (due to steps 1 and 2), then M is moved to the SOLVED state of the EXECUTION mode.

The three operations, *reset*, *cancel* and *redo* are explained in the following.

Reset P_i

1. Set the B-list of P_i to a list containing only its immediate D-predecessors.
2. Cancel all the descendants of P_i in the data dependency graph D (if they have not already been canceled).
3. Start the execution of P_i from the beginning, i.e., as if it has just made the transition from the GATHER mode to the UNSOLVED state of the EXECUTION mode.

Cancel P_i

1. Switch P_i to the GATHER mode.

Redo P_i

1. Change the state of P_i to "UNSOLVED", so that P_i can resume execution to find a new set of values for the variables it generates.

See Appendix B for an illustration of the algorithm. Note that while presenting the forward and backward execution algorithms, we have not specified the actual mechanism for transferring information between various literals (i.e., processes representing the literals). This has to be done carefully to ensure that messages do not get out of

synchronization. The actual mechanism for message transfer is described in detail in Lin's thesis [9]. To simplify the discussion in this paper, we assume that information transfer between the literals can be done instantaneously. We also assume that any two literals do not fail exactly at the same time.

The forward and backward execution algorithms presented so far would only find one solution for a given clause. In practice, we may need more than one solution for a clause. To find the next solution of a given clause, we simply send a fail message from a dummy literal to the rightmost generator of the clause in the linear order (i.e., P_m such that m = max{k | P_k is a generator}). This would restart the execution of the clause. If there are no more solutions, then the execution will terminate with failure. Otherwise, a new solution will be found when all the literals of the clause are in the FINISHED mode. Note that the dummy literal remains invisible at all times except when a new solution to the clause is needed.

5. The Correctness Proof of our Backward Execution Algorithm

By correctness we mean that if there are solutions for a given clause body, the backward execution algorithm should guarantee that every solution can be generated eventually. The order in which these solutions are generated is unimportant.

At the time a literal P_i fails, let X_i = {L| L is a D-predecessor of P_i} ∪ {L| L is a D-predecessor of M and the failure of M has directly or indirectly[2] caused a redo operation on P_i since P_i was most recently changed from the GATHER mode to the EXECUTION mode}. 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 resolved to make P_i succeed again. To ensure that all possible combinations of the values generated by the literals in X_i can be tried, we make use of the linear ordering on the literals defined in section 2. 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 [5]. The following lemma says that {P_m | $P_m \in X_i$ 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: See Appendix A.

Let bi = max{k | $P_k \in X_i$ 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.

Let S_i = {L| L is an immediate D-predecessor of P_i} ∪ {L| L is an immediate D-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 mode to the EXECUTION mode}. Lemma 2 says that bi is equal to max{k | $P_k \in S_i$ and $k < i$}.

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

[2] 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.

Proof: See Appendix A.

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

Proof: See Appendix A.

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}.

6. Intelligent Backtracking without AND parallelism

Even in the absence of parallel hardware, our backtrack algorithm is useful because it is potentially much more efficient than the "naive" backtracking strategy used by Prolog. If only one processor is available, then we can execute P_i such that $i = \min\{j \mid P_j$ is in the UNSOLVED state of the EXECUTION mode$\}$. In this case, the work done by our algorithm (ignoring the communication overhead) is a subset of the work done by the backtracking strategy of Prolog. This happens because, if a literal P_i fails, then our algorithm causes backtracking to P_{bi}, whereas Prolog backtracks to all the literals P_j such that $bi<j<i$ before backtracking to P_{bi}. The discussion in section 5 makes it clear that backtracking to any literal P_j such that $bi<j<i$ is guaranteed to be wasteful.

7. Related Work

7.1. Conery & Kibler's Backward Execution Algorithm

The backward execution algorithm presented by Conery & Kibler [3], [5] is incorrect because, in some cases, it does not find a solution when a solution exists. 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, that literal is removed from the failure context, and the history of the failure is lost. We run Conery's algorithm on the example given in Appendix B to illustrate out point.

Assume that we are solving P_0(A,B,C) with the following set of clauses.

P_0(A,B,C) ← 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_5(b1, c2).
P_5(b2, c1).

The data dependency graph for solving P_0(A,B,C) is in Figure 2. In this example we use the same notation as Conery used in [5]. #N denotes the literal P_N and ↑N denotes the

OR process created to solve P_N.

Suppose that #1, #2, and #3 have succeeded. After the success messages arrived, the status of the AND process is: subgoals #1, #2, and #3 solved; subgoals #4 and #5 pending; failure context empty. The current bindings are a1/A, b1/B, c1/C.

After AND process received fail message from ↑5. The failure context is set to [#5], which is the prefix of the redo list [#5, #3, #2, #1, head]. A redo message is sent to ↑3, and the actions taken for each subgoals to the right of #3 in the linear ordering are:
#4: canceled.
#5: already terminated.
The current bindings are a1/A, b1/B.

The success from ↑3 arrived again. New processes for #4 and #5 are created. Since there is a new process for #5, it is removed from the failure context. The state of the AND process: subgoals #1, #2, and #3 solved; subgoals #4 and #5 pending; failure context empty. The current bindings are a1/A, b1/B, c2/C.

This time AND process receives fail message from ↑4. The failure context is set to [#4], which is the prefix of the redo list [#4, #3, #1, head]. A redo message is sent to ↑3 again, and the remaining subgoals in the linear ordering are handled as follows:
#4: already terminated.
#5: canceled.
The current bindings are a1/A, b1/B.

#3 fails this time. It sends back a fail message. After receiving fail message from ↑3, AND process sets the failure context to [#4, #3], which is the prefix of the redo list [#4, #3, #1, head]. This time a redo message is sent to ↑1, and the actions taken for each remaining subgoals are:
#2: canceled.
#3: already terminated.
#4: already terminated.
#5: already terminated.
The current binding is none.

Since there is only one solution for P_1, AND process will receive fail message from ↑1. It then sets the failure context to [#4, #3, #1], which is the prefix of the redo list [#4, #3, #1, head]. The suffix is [head], and the AND process fails. However a solution a1/A, b2/B, and c1/C does exist for P_0. The reason Conery's algorithm cannot find this is because part of the history of failure for #3 (i.e., the failure of #5) was removed from the failure context and lost.

7.2. Semi-intelligent Backtracking of Chang & Despain

Chang and Despain have recently presented a semi-intelligent backtracking scheme [2]. Their scheme is similar to our scheme except that backtrack point at each literal is statically determined. In their scheme, each literal is involved in either type-I backtracking or type-II backtracking. Type-I backtracking occurs when a literal has a failure level of 0 (failure level is defined in the proof of Lemma 3 in Appendix A). In this case, the two schemes backtrack exactly to the same literal.

When a literal is involved in type-II backtracking, then it backtracks to P_j such that $j = \max\{m \mid P_m$ is a D-predecessor of any literal P_k which may cause the failure of P_i and $m < i < k\}$. Since the analysis is done for the worst case, the set they use to choose P_j is a superset of X_i defined in the section 5. Therefore $bi \leq j < i$. If $bi < j$, then

P_i would keep failing for all the new values generated by P_j for exactly the same reasons for which it failed before. And hence backtracking to P_j is guaranteed to be wasteful.

7.3. The Intelligent Backtracking Scheme of Bruynooghe, Pereira & Porto

Bruynooghe, Pereira, Porto, and Cox proposed an Intelligent Backtracking scheme which involves more complicated run-time data structures [1], [10], [6] than used by our scheme. In order to achieve intelligent backtracking in their scheme, it is necessary to keep track of a set of candidate goals for backtracking upon failure of a goal G. In their scheme, this set consists of the ancestor (parent is sufficient) of G, the modifying goals for G, and the legacy set of G.

Since their method is based on "deduction tree", backtracking to parent simply means re-solving the failed goal with different alternatives, which is taken care of automatically in our algorithm when we try to solve a literal. The legacy set is also passed around in our algorithm when backtrack operation is performed. The only difference between our algorithm and theirs is the way of deciding the set of modifying goals. In our algorithm, when a goal fails, each of its immediate D-predecessors in the data dependency graph is a modifying goal. But in their scheme, deciding a modifying goal involves a lot of run-time analysis on things such as where the unification conflicts happened, which literal is strongly deterministic, etc. It also needs a lot of run-time bookkeeping for information such as which literal binds which variable. Although the run-time analysis and bookkeeping may help their scheme to reduce the number of modifying goals, the overhead in terms of both time and space can be excessive.

8. Concluding Remarks

We have presented a backtracking scheme which works when AND-parallelism is exploited in a logic program. The scheme is well suited for implementation on a parallel hardware. Since, the scheme makes use of the dependencies between the literals in a clause, it is potentially useful (as a replacement for the naive backtracking strategy of Prolog) even if no parallel hardware is available to exploit AND-parallelism. In the sequential case, our scheme avoids unnecessary backtracking at the expense of the overhead associated with the transfer of messages between literals. This overhead is much smaller than the overhead associated with the intelligent backtracking scheme of Porto, Pereira and Bruynooghe.

Appendix A

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 an ancestor of P_i. Let P_m be an ancestor of P_j, where the failure of P_j 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.

Lemma 2. $bi = \max\{k \mid P_k \in S_i \text{ and } k < i\}$.

Proof: First we prove that $P_{bi} \in S_i$. By definition, the set S_i is the union of the immediate D-predecessors of P_i and those literals which are immediate D-predecessors of some P_j such P_j has directly or indirectly caused the failure of P_i. Suppose P_{bi} is not an immediate D-predecessor of any such P_j, and P_{bi} is not an immediate D-predecessor of P_i. Then since $P_{bi} \in X_i$, P_{bi} must be an ancestor of some P_m which is either an immediate D-predecessor of a certain P_j, or an immediate D-predecessor 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, $bi < m$ because P_{bi} is an ancestor of P_m. Thus, there is a $P_m \in X_i$ such that $bi < m < i$, which is contrary to the definition of bi.

Now we prove that $bi = \max\{k \mid P_k \in S_i \text{ and } k < i\}$. From the definition of X_i and S_i, it follows that $\{P_k \mid P_k \in S_i \text{ and } k < i\} \subseteq \{P_k \mid P_k \in X_i \text{ and } k < i\}$. Hence $bi = \max\{k \mid P_k \in X_i \text{ and } k < i\} \geq \max\{k \mid P_k \in S_i \text{ and } k < i\}$. But $P_{bi} \in S_i$ and $bi < i$; therefore $bi \leq \max\{k \mid P_k \in S_i \text{ and } k < i\}$. It follows that $bi = \max\{k \mid P_k \in S_i \text{ and } k < i\}$.

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

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 mode to the EXECUTION mode, then the failure level of L=0.
2. Otherwise, the failure level of L = n+1, where n = max{f(Q)| failure of Q has directly invoked a redo operation on L since L started executing most recently, and f(Q) was the failure level of Q at the time Q failed}.

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

Since no redo operation has been done on P_i, $\{P_j \mid P_j \in \text{B-list}(P_i)\} = S_i = \{P_j \mid P_j$ is an immediate D-predecessor 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 EXECUTION mode, assume that the failure of $P_{i1},...,P_{ik},...,P_{in}$ have directly invoked redo operations on P_i. Note that each P_{ik} $(1 \leq k \leq n)$ is either an immediate D-successor 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=$i1$ to in, and the set of immediate D-predecessors of P_i. Therefore, any literal P_j in $\{P_j \mid P_j \in S_i$ and $j<i\}$ should be either an immediate D-predecessor of P_i or in S_{ik} for some $1 \leq k \leq n$. If P_j is an immediate D-predecessor of P_i, then P_j is surely in B-list(P_i). If P_j is not an immediate D-predecessor P_i, then (because $j<i<ik$) P_j should appear in B-list(P_{ik}) by the induction hypothesis. Hence, it would have been merged into B-list(P_i) by our backward execution algorithm while handling the failure of P_{ik}. Therefore B-list(P_i) contains every literal in $\{P_j \mid P_j \in S_i$ and $j<i\}$.

If P_m is in B-list(P_i), then either P_m is an immediate D-predecessor 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 an immediate D-predecessor 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 $\{P_j \mid P_j \in S_i$ and $j<i\}$ contains every literal in B-list(P_i).

Appendix B

The following is an example of how our backward execution algorithm works. Assume that we are solving P_0(A,B,C) with the following set of clauses.

P_0(A,B,C) ← 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_5(b1, c2).
P_5(b2, c1).

The data dependency graph for solving P_0(A,B,C) is in Figure 2.

Unfinished-successors of P_i represents the set of its immediate D-successors which are not in the FINISHED mode. It becomes non-empty only when P_i sends out generated values. #N denotes subgoal P_N and ↑N denotes the process created to solve P_N.

<1>#1, #2, #3, and #4 have succeeded. Process for #5 is still executing.
Current bindings are a1/A, b1/B, c1/C.

Literal	Mode	Status	Unfinished-successors	B-list
#1	execution	solved	(#2, #3)	()
#2	execution	solved	(#5)	(#1)
#3	execution	solved	(#5)	(#1)
#4	finished	-	()	(#3)
#5	execution	unsolved	()	(#3, #2)

<2>#5 fails. ↑3 is sent a redo message. ↑4 and ↑5 are canceled.
Current bindings are a1/A, b1/B.

Literal	Mode	Status	Unfinished-successors	B-list
#1	execution	solved	(#2, #3)	()
#2	execution	solved	(#5)	(#1)
#3	execution	unsolved	()	(#2, #1)
#4	gather	-	()	()
#5	gather	-	()	()

<3>#3 succeeds again. Both P_4 and P_5 start executing.

Current bindings are a1/A, b1/B, c2/C.

Literal	Mode	Status	Unfinished-successors	B-list
#1	execution	solved	(#2, #3)	()
#2	execution	solved	(#5)	(#1)
#3	execution	solved	(#4, #5)	(#2, #1)
#4	execution	unsolved	()	(#3)
#5	execution	unsolved	()	(#3, #2)

<4>#4 fails. ↑3 is sent another redo. ↑ 4 and ↑5 are canceled.

Current bindings are a1/A, b1/B.

Literal	Mode	Status	Unfinished-successors	B-list
#1	execution	solved	(#2, #3)	()
#2	execution	solved	(#5)	(#1)
#3	execution	unsolved	()	(#2, #1)
#4	gather	-	()	()
#5	gather	-	()	()

<5>#3 can not generate any new binding and therefore it fails. ↑2 is sent a redo message. ↑3 is reset. ↑4 and ↑5 are already canceled.

Conery's backward execution algorithm will incorrectly decide to send ↑1 a redo message in this case, as we have shown in section 7.1.

Current bindings are a1/A.

Literal	Mode	Status	Unfinished-successors	B-list
#1	execution	solved	(#2, #3)	()
#2	execution	unsolved	()	(#1)
#3	execution	unsolved	()	(#1)
#4	gather	-	()	()
#5	gather	-	()	()

<6>#2 succeeds and generates a new value for variable B. #3 also succeeds and generates a value for variable C which is the first value it generated before. Both ↑4 and ↑5 can start executing.

Current bindings are a1/A, b2/B, c1/C.

Literal	Mode	Status	Unfinished-successors	B-list
#1	execution	solved	(#2, #3)	()
#2	execution	solved	(#5)	(#1)
#3	execution	solved	(#4, #5)	(#1)
#4	execution	unsolved	()	(#3)
#5	execution	unsolved	()	(#3, #2)

<7>This time both #4 and #5 succeed.

The final bindings are a1/A, b2/B, c1/C.

Literal	Mode	Status	Unfinished-successors	B-list
#1	finished	-	()	()
#2	finished	-	()	(#1)
#3	finished	-	()	(#1)
#4	finished	-	()	(#3)
#5	finished	-	()	(#3, #2)

References

[1] M. Bruynooghe and L. M. Pereira, Deduction Revision by Intelligent Backtracking, pp. 194-215 in *Implementations of Prolog,* ed. J. A. Campbell, Ellis Horwood Limited, 1984.

[2] J.-H. Chang and A.M. Despain, Semi-Intelligent Backtracking of Prolog Based on a Static Data Dependency Analysis, *Proceedings of IEEE Symposium on Logic Programming,* pp. 10-21, August, 1985.

[3] J. S. Conery, The AND/OR Process Model for Parallel Interpretation of Logic Programs, *Ph.D. Thesis, (Technical Report 204),* Irvine, California, University of California, June, 1983.

[4] J. S. Conery and D. F. Kibler, Parallel Interpretation of Logic Programs, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture,* pp. 163-170, ACM, October, 1981.

[5] J.S. Conery and D.F. Kibler, AND Parallelism and Nondeterminism in Logic Programs, *New Generation Computing 3(1985),* pp. 43-70, OHMSHA,LTD. and Springer-Verlag, 1985.

[6] P. T. Cox, Finding Backtrack Points for Intelligent Backtracking, pp. 216-233 in *Implementations of Prolog,* ed. J. A. Campbell, Ellis Horwood Limited, 1984.

[7] L.V. Kale and D.S. Warren, A Class of Architectures for a Prolog Machine, *Proceeding of the Second International Logic Programming Conference,* Uppsala, Sweden, pp. 171-182, July, 1984.

[8] S. Kasif and J. Minker, The Intelligent Channel: A Scheme for Result Sharing in Logic Programs, *Proceedings of the 9th IJCAI,* Los Angeles, pp. 29-31, August, 1985.

[9] Y.J. Lin, A Parallel Implementation of Logic Programs, *Ph.D. dissertation,* Austin, Texas, The University of Texas at Austin, in preparation.

[10] L. M. Pereira and A. Porto, Selective Backtracking, pp. 107-114 in *Logic Programming,* ed. S.-A. Tarnlund, Academic Press, 1982.

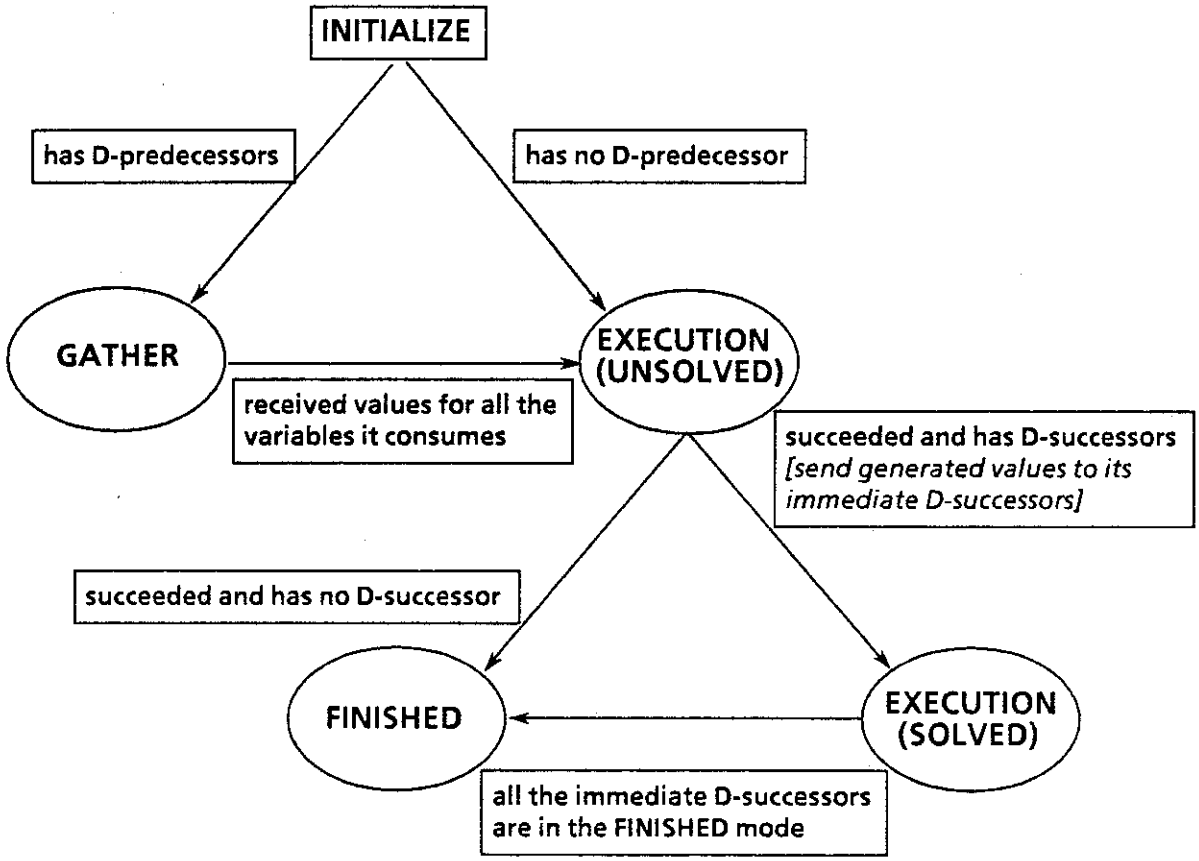

Figure 1: Graphical form of the Forward Execution Algorithm

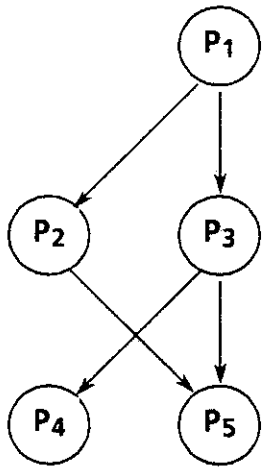

Figure 2: Data dependency graph of the example