Yet another efficient backward execution algorithm in the AND/OR Process Model

Do-Hyung Kim and Kwang-Moo Choe

Programming Languages Laboratory, Department of Computer Science, Korea Advanced Institute of Science and Technology, 371-1, Kusong-Dong, Yusung-Gu, Taejon 305-701, South Korea

Communicated by K. Ikeda
Received 7 January 1991
Revised 10 August 1991

Abstract


An efficient backward execution algorithm in the AND/OR Process Model for parallel evaluation of logic programs is proposed. The efficiency of the algorithm is achieved by means of information acquired during execution of clauses. The algorithm is considered to be efficient in the sense that it issues fewer number of cancel messages and avoids unnecessary resetting operations. Furthermore, it performs independent redoing and resetting more concurrently than other related works.

Keywords: Applicative (logic) programming, parallel evaluation of logic programs, AND/OR Process Model, AND parallelism, backward execution algorithm, multiple failure, selective resetting

1. Introduction

Among several new programming paradigms ever proposed, logic programming has attracted much attention of many researchers especially owing to massive parallelism embedded in logic programs. This parallelism can be classified into various types [3]: AND, OR, stream, and search parallelism, among which AND and OR parallelism are the most notable ones.

For a given logic program, we can readily construct an AND/OR proof tree which represents the search space for executing the program [8]. That is, we can easily find an analogy between “to execute a logic program” and “to search within its corresponding AND/OR proof tree”. AND (OR) parallelism refers to parallel search from AND (OR) nodes into OR (AND) nodes in AND/OR proof trees. According to the characteristics of AND/OR proof trees, fully parallel search from AND nodes into OR nodes might not be allowed to prevent variable binding conflicts, while OR parallelism could be totally achieved (apart from the problem of efficient implementation, of course).

Thus, many computational models to exploit AND parallelism have been proposed [1–3,6,10–13]. Among them, the AND/OR Process Model [3] is dominant since it reflects computation structures of logic programs (i.e., the AND/OR proof trees) very well, as its name implies. (In fact, we think that many other models including ours are improvements, modifications, or variants of the AND/OR Process Model.)

In the model, a process is created for each node in an AND/OR proof tree. There are two kinds of processes, AND and OR processes, like two kinds of nodes, AND and OR nodes. OR
process and AND process take charge of resolving a literal and a clause, respectively. The operational semantics of the model is conceptually simple. When an OR process is activated, it simultaneously creates child AND processes for all clauses of which head literals can be unified with the literal assigned to this OR process. Then the OR process starts child AND processes and awaits their (success or failure) messages.

The situation becomes a little complicated for an AND process. The AND/OR Process Model, for exploiting AND parallelism, uses three algorithms: the literal ordering, forward execution, and backward execution algorithms. When an AND process is activated, it simultaneously creates child OR processes for "some" body literals, whose all producer literals have been solved or do not exist, in the clause assigned to this AND process. Determining producer-consumer (as a result) relationships among body literals is the role of the literal ordering algorithm and is accomplished by means of data dependency among body literals. Then the AND process also starts child OR processes and awaits their (success or failure) messages. This is the beginning of the forward execution phase. The goal of the forward execution algorithm is binding unbound variables together with terms. If any child OR processes report failure, the backward execution algorithm is triggered and backtracking is performed by the algorithm. When more than one literal reports failure, the situation is called the multiple failure case [13].

Now, let us proceed to the main issue of this paper: the efficient backward execution. One backward execution algorithm may be said to be more efficient than others if it does less amount of unnecessary work or does not wait unnecessarily (e.g., unnecessary serialization of concurrent tasks). In this paper, we present an efficient backward execution algorithm for the AND/OR Process Model. It is efficient in the sense that (1) it does not reset unfruitable literals, (2) it issues fewer number of unnecessary cancel messages, and (3) it performs redoing/resetting more concurrently for the multiple failure case. (In Section 2, redoing, resetting, and canceling will be described in more detail.)

This paper is organized as follows. Section 2 presents problem identification and motivation. Our backward execution algorithm is described in Section 3. Section 4 contains comparison with other related works and some discussion followed by the conclusion.

2. Background and motivation

In this section, more detailed description of backward execution is presented in order to clearly identify the problems that this paper tackles.

As mentioned before, the backward execution starts when a literal (henceforth, literal and OR process shall be used interchangeably if there is no danger of misunderstanding) reports failure (for the time being, let us focus on the single failure case [13] only). If the literal has no predecessor literals in data dependency relation, the overall execution of this AND process fails. Otherwise, in order to alleviate the failure, redoing a backtrack literal must occur. Generally, there are several literals which may cure the failure, that is, more than one "candidate backtrack literal" exists. In such cases, which literal should be redone first to rebind its variables? In other words, which variables should change their values first to make another candidate solution tuple for checking its validity? In order not to miss any possible solution tuples, it is obvious for us to select the backtrack literal among candidates with certain fixed rules systematically. To solve this problem, a straightforward and natural method, called nested loop model for tuple generation, was proposed [3] and has been used widely. It imposes a linear ordering on body literals. After constructing the data dependency graph (DDG), which represents producer-consumer (or data dependency, equivalently) relation, we can get a list of linearly ordered literals by a breadth-first traverse of the graph. This list is named linearly ordered literals list (LOLL) and is an important data structure in the AND/OR Process Model.

There are two major problems to be solved in backward execution. The first one has just been described in the previous paragraph, namely the
backtrack literal selection problem. After the backtrack literal is determined according to the nested loop model and the literal is redone to produce different bindings for its variables, it is necessary to re-start execution of literals succeeding the backtrack literal in the LOLL lest we omit any possible value combinations. Thus, we have to decide which literals to re-start (or reset) from the beginning. It is designated as the reset problem. This reminds us of repetitive structures in imperative languages, in which when an outer (index) variable in a nested loop changes its value, all inner variables must be reset. The name is coined after the structure. Literals which consume values forwarded by the redone or reset literals are naturally canceled, since those values have become obsolete now.

Needless to say, the fewer reset literals are, the more efficient backward execution is. There is a possibility of resetting only part of the literals succeeding the backtrack literal, because the data dependency relation is a partial order even though the name of LOLL indicates it is a linear (or simple) order (thus somewhat misleading). This attempt might be called literal-level selective resetting [1,2,10,11]. On the other hand, there are researches which may be named solution-level selective resetting [6,12]. They are trying to make "part" of the past solutions available again for reset literals. Our paper handles literal-level selective resetting only. (Whether it is literal- or solution-level, we believe selective resetting is fairly important for efficient execution of logic programs. It is a dual action with intelligent backtracking in the sense that for the purpose of efficiency both aim at pruning AND/OR proof trees, but at different locations.)

The motivation and goals for our paper are twofold: (1) to show that more efficient backward execution can be devised by the use of run-time information (naturally!), and (2) to sum up several works on literal-level selective resetting and to show that they are essentially the same.

3. Dynamic affecting set

In this section, we first present the rationale of our backward execution algorithm. Then relevant definitions, lemmas, and actual algorithm follow.

3.1. Rationale

Let us consider a query (or goal statement) in a logic program and DDG corresponding to the

![Fig. 1. A query and DDG for it.](image)
query in Fig. 1. Suppose that some solutions of literal no. 5 (a body literal may be identified by its ordinal number afterwards), say \( e_0, e_1, \ldots, e_n \), were rejected by no. 8 since no. 8 could not succeed with any of them. No. 4 failed. Then, no. 2 was chosen as the backtrack literal and redone. The reset literals must be determined now.

\[
\leftarrow p_1(A, B), p_2(A, C), p_3(A, D), p_4(B, C), p_5(B, E), p_6(D), p_7(C, E), p_8(E).
\]

Concerning this (reset) problem, Conery adopted a very simple approach following the nested loop model rigidly: all generator (producer) literals succeeding a redone literal in the LOLL (in this case, no. 3 and no. 5) must be reset [3]. (It, however, is clear from the DDG that resetting no. 3 is useless in this situation.)

Later several researchers pointed out that Conery’s strategy was too conservative and some literals following a redone literal do not have to be reset, i.e., literal-level selective resetting is possible, through certain analyses. The notion of literal-level selective resetting originated from Chang et al.’s backtrack path [1]. Choe et al. [2] and Park et al. [11] also independently developed (literal-level) selective resetting schemes based on the same notion as Chang et al.’s. All these methods noticed that there might be “independence” among some literals in a clause. Thus they do not reset literals which are “independent” of a redone literal.

We think independence among literals can be clearly explained in terms of a very simple notion, called affection. (In this paper, “affection” is used as a synonym of “effect”.) Affection can be classified into two types; “forwarding values” and “causing redoing”. Data dependency (i.e., forwarding/receiving values) represented in the DDG is considered as a special case of affection. The DDG tells us effect of parent literals on child ones, but effect of child literals on parent ones is not expressed in the DDG. Child literals, however, can affect their parent ones too—their failure causes redoing parent literals.

Thus, if we keep information about affection history among literals, we can naturally and efficiently perform backward execution in case of failure. It is intuitively clear that literals which have affected a failed literal are candidates for the backtrack literal, and the rightmost one among them in the LOLL must be selected as the backtrack literal according to the nested loop model. After redoing the backtrack literal, it is also obvious that we must reset generator literals, which have been affected by the backtrack literal, and cancel descendant literals whose parents are redone or reset. Therefore, if two literals do not affect each other, they are independent of each other.

We may also visualize this affection among literals, like data dependency in the DDG, in another directed graph called affection graph (AG). The DDG is a subgraph of the AG. There are two types of arcs in the AG: data and redo arcs. Data and redo arcs represent data dependency and causing redoing, respectively (data arcs come from the DDG). Whenever a failed literal causes redoing its backtrack literal, a new redo arc is created from the failure to the backtrack literal. Hence, if there is a (directed) path from a literal to another literal in the AG, the former affects the latter. (The AG will be described in detail in [5].) For example, in the current failure situation described before with Fig. 1, we can imagine a hypothetical (redo) arc from no. 8 to no. 5 in the DDG, so a path is opened from no. 8 to no. 5, i.e., no. 8 affects no. 5. Then, after redoing no. 2, we must do some operations on literals which have been affected by no. 2 (those are no. 4 and no. 7). In this example, they consume values generated by no. 2, so they are canceled. No. 3 and no. 5 are independent of no. 2, so no operations on them are necessary. On the other hand, if no. 7 had failed instead of no. 8 and caused redoing no. 5, no. 5 would have been an affected literal of no. 2 (note that a path from no. 2 to no. 5, no. 5 \( \rightarrow \) no. 7 \( \rightarrow \) no. 5, would have been open). So no. 5 must be reset by redoing no. 2, since no. 5 is not a child of any redone or reset literals among affected literals of no. 2, i.e., no. 4, no. 5, no. 7, and no. 8. We think these observations naturally reflect intuition behind resetting and canceling.

The dynamic affecting set of a literal, say \( P \), refers to the set of literals which have been affected by \( P \) during execution. The set is dynam-
ically constructed by means of affection history. Further, a dynamic affecting set is partitioned into two disjoint sets: *dynamic canceled* and *reset literals sets*. The aim of this paper is to propose a method to get these sets and utilize them for backward execution.

### 3.2. Algorithm

The AG is a conceptual framework rather than a realistic algorithm. Thus, in the following, we present the definition of dynamic affecting sets. To do so, some preliminaries will be followed. The following definition is necessary to support the nested loop model.

**Definition** (reduced data dependency graph [2]). The "reduced data dependency graph for a literal \( P \)", \( G_p \), is a subgraph of the DDG \( G \) where

\[
\text{VERTEX}(G_p) = \{ Q \mid Q \in \text{VERTEX}(G), \text{ORD}(P) \leq \text{ORD}(Q) \},
\]

\[
\text{EDGE}(G_p) = \{ Q \rightarrow R \mid Q \rightarrow R \in \text{EDGE}(G), \text{ORD}(P) \leq \text{ORD}(Q) < \text{ORD}(R) \}.
\]

\( \text{VERTEX}(G_p) \) and \( \text{EDGE}(G_p) \) denote the set of vertices and edges in \( G_p \), respectively. \( \text{ORD}(P) \) is the ordinal number of \( P \) in the LOLL.

Next, forming new affection relation by redoing is expounded in the following definitions:

**Definition** (causes redoing). Let \( P \) and \( Q \) be literals. Then we say "\( P \) \text{ REDO } Q" iff \( P \neq Q \), \( P \in \text{VERTEX}(G_Q) \), and either of the following two conditions holds:

1. the failure of \( P \) directly caused redoing \( Q \) or
2. there is a literal \( R \) such that \( R \neq P, R \neq Q, R \in \text{VERTEX}(G_Q) \), the failure of \( P \) directly caused redoing \( R \) and \( R \) \text{ REDO } Q.

**Definition** (dynamic affecting set). Let \( P \) and \( Q \) be literals. Then we say \( \text{"} P \text{ D-AFFECT } Q \text{"} \) iff \( P \neq Q, Q \in \text{VERTEX}(G_P) \), and either of the following two conditions holds (\text{DESCENDANTS}(P) stands for \( P \)'s descendants in the DDG):

1. \( Q \in \text{DESCENDANTS}(P) \) or
2. there is a literal \( R \) such that \( R \neq P, R \neq Q, R \in \text{VERTEX}(G_P), R \in \text{DESCENDANTS}(P) \), and \( R \) \text{ REDO } \( Q \).

Then, \( \text{D-AFFECTING}(P) = \{ Q \mid P \text{ D-AFFECT } Q \} \).

Now we can define independence among literals in terms of their affecting sets.

**Definition** (dynamic independence). Let \( P \) and \( Q \) be two different literals. Then we say "\( P \) and \( Q \) are dynamically independent of each other" iff \( P \notin \text{D-AFFECTING}(Q) \) and \( Q \notin \text{D-AFFECTING}(P) \).

If a literal is redone or reset, literals in its dynamic affecting set are influenced by the change. If they consume some values generated by the redone or reset literal, which may be changed, they must be canceled naturally. Otherwise, they are reset.

**Definition** (dynamic canceled/reset literals set).

1. \( \text{D-CANCEL}(P) = \{ Q \mid P \in \text{CHILDREN}(R), \text{ORD}(P) \leq \text{ORD}(Q) \leq \text{ORD}(R) \} \).
2. \( \text{D-RESET}(P) = \text{D-AFFECTING}(P) - \text{D-CANCEL}(P) \).

\( \text{CHILDREN}(P) \) denotes \( P \)'s children in the DDG.

The following two propositions summarize obvious observations on dynamic affecting sets. (The proofs to the propositions in this paper are omitted. They are, however, fairly simple and obvious.)

**Lemma** (dynamic transitivity). If \( P \text{ D-AFFECT } Q \) and \( R \in \text{D-CANCEL}(Q) \), then \( R \in \text{D-CANCEL}(P) \).

**Theorem** (dynamic cancellation). If \( P \text{ D-AFFECT } Q \) and \( R \in \text{D-CANCEL}(Q) \), then \( R \in \text{D-CANCEL}(P) \).

We are now in a position to describe our backward execution algorithm.

**Algorithm. Backward execution.**

1. /*Selection of the backtrack literal*/
   1-1) Construct the set of candidate backtrack literals, \( L_B \), for a failed literal \( L_f \).
The set of candidate backtrack literals can be obtained by any algorithm in [1, 6, 7, 9, 10, 12, 13]. The set appears in different names in the literature, such as B-LIST, REDO SET, etc. /*The set of candidate backtrack literals can be obtained by any algorithm in [1.6, 7, 9, 10, 12, 13]. The set appears in different names in the literature, such as B-LIST, REDO SET, etc.*/

1. Select the backtrack (in fact, rightmost) literal, \( \ell \), among the elements in \( L_B \).
2. *Updating affecting sets, redoing, resetting, and canceling*
   1. Update the affecting sets of the elements in \( L_B \) - \{ \ell \}.
      /*This operation seems obvious from the definitions in this section. Examples of updating a kind of affecting sets can be found in [4, 6, 7].*/
   2. Send a redo message to the OR process for \( L_B \).
   3. For each literal in \( D_{\text{RESET}}(L_B) \), perform reset operation, that is, let the literal regenerate its solutions from the beginning again.
   4. After resetting each literal, say \( \ell_r \), in \( D_{\text{RESET}}(L_B) \), delete \( \ell_r \) from \( D_{\text{AFFFECTING}}(L_B) \) (if any) from the BACKTRACK_LIST.
   5. /*Analogically speaking, the dynamically constructed redo arcs on a path from \( L_B \) to \( \ell_r \) in the AG are discarded by resetting.*/
   6. For each literal in \( D_{\text{CANCEL}}(L_B) \), send a cancel message to the corresponding OR process.

3.3. Handling multiple failure

Due to concurrent execution of many OR processes, multiple failure cases are quite likely to occur frequently.

Woo and Choe originally proposed an algorithm handling multiple failure [13]. As succeeding researches find the fact that there is independence among some literals, algorithms managing multiple failure in parallel have appeared [2, 11]. Their rationale is simple and straightforward: when two different backtrack literals for two different failed literals are independent, they can be redone concurrently.

Our algorithm to handle multiple failure is a dynamic version of that in [2].

Algorithm. Dynamic multiple backtracking

1. For each failed literal \( L_f, (i = 1, 2, \ldots, n) \),
   1-1. Invoke an algorithm to find the backtrack literal \( L_{b_i} \) for \( L_f \).
   1-2. If the previous step returns "AND process fails" as a result, then exit this algorithm with "AND process fails." Otherwise, continue,
   2. Rearrange \( L_{b_i} \) in ascending order, according to \( \text{ord}(L_{b_i}) \), to make the BACKTRACK_LIST.
   3. Repeat the following sub-steps until the BACKTRACK_LIST becomes empty:
      3-1. Select the leftmost literal, say \( L_{b_i} \).
      3-2. Delete \( L_{b_i} \) and the literals in \( D_{\text{AFFECTING}}(L_{b_i}) \) (if any) from the BACKTRACK_LIST.
      3-3. Perform the backward execution algorithm for \( L_f \).

4. Comparison and discussion

In this section, we compare other literal-level selective resetting schemes with ours.

4.1. Choe et al.

The notion of affection was first suggested by Choe et al. [2]. Their "affecting sets" are essentially the same as our dynamic affecting sets in concept. Their analysis, however, is static, and thus conservative in the sense that the method may not find certain independence among literals. Consequently their scheme may do unnecessary resetting and canceling. Also their method may be less parallel in backtracking.

For the purpose of reference and comparison (see Sections 4.4 and 4.5), their key definitions and propositions (in slightly modified notation) are included here.

Definition (static affecting set [2]). Let \( P \) and \( Q \) be literals. Then we say \( P \ s_{\text{AFFECT}} Q \) iff \( P \neq Q, Q \in \text{vertex}(G_P) \), and the two vertices \( P \) and \( Q \) are connected in \( G_P \). The two vertices are said to be connected iff there is a path between
two vertices when we ignore the direction of the edges. Then, \( s\text{-affecting}(P) = \{Q \mid P \ s\text{-affect} \ Q \}. \)

**Definition** (static independence). Let \( P \) and \( Q \) be two different literals. Then we say “\( P \) and \( Q \) are statically independent of each other” iff 
\[ P \notin s\text{-affecting}(Q) \text{ and } Q \notin s\text{-affecting}(P), \]
or equivalently 
\[ s\text{-affecting}(P) \cap s\text{-affecting}(Q) = \emptyset. \]

**Definition** (static canceled/reset literals set [2]).

1. \( s\text{-cancel}(P) = \{Q \mid \exists \text{ CHILDREN}(R), \ R \in \{P\} \cup s\text{-affecting}(P)\}. \)
2. \( s\text{-reset}(P) = s\text{-affecting}(P) - s\text{-cancel}(P). \)

**Lemma** (static transitivity [2]). If \( P \ s\text{-affect} \ Q \) and \( Q \ s\text{-affect} \ R \), then \( P \ s\text{-affect} \ R. \)

**Theorem** (static inclusion [2]). If \( P \ s\text{-affect} \ R, \ Q \ s\text{-affect} \ R, \) and \( \text{ORD}(P) < \text{ORD}(Q), \) then \( P \ s\text{-affect} \ Q \) and hence \( \text{Affect}(Q) \subseteq s\text{-affecting}(P). \)

**Theorem** (static cancellation [2]). If \( P \ s\text{-affect} \ Q \) and \( R \in s\text{-cancel}(Q), \) then \( R \in s\text{-cancel}(P). \)

### 4.2. Park et al.

Park et al. proposed another literal-level selective resetting method [11], which uses static analysis and exhibits the same behavior as Choe et al.’s. Although their scheme is a little different from Choe et al.’s in notation, the philosophy upon which those two methods are based is identical, as mentioned in Section 3.1.

Also, for the purpose of reference and comparison (see Sections 4.4 and 4.5), the key definitions of their method are contained here.

**Definition** (Park et al. [11]). \( \text{CON} \) represents the set of all consumer literals, i.e., all leaf nodes in the DDG.

1. The “parent set of a literal \( P \)” is given as \( \text{parent}(P) = \{Q \mid \text{there is a directed edge from } Q \text{ to } P\}. \)
2. The “directly reachable literal set of a literal \( P \)” is given as 
\[ \text{drls}(P) = \{Q \mid \text{there is a directed path from } P \text{ to } Q\}. \]
3. The “indirectly reachable literal set of a literal \( P \)” is given as 
\[ \text{idrls}(P) = \text{id}(P) - \text{drls}(P), \]
where
\[ \text{id}(P) = \{Q \mid \text{there is a path between } P \text{ and } Q \text{ ignoring the direction of edges, such that each literal on the path (including } Q \text{) is later than } P \text{ in the linear ordering)}\].
4. The “related reset literal set of a literal \( P \)” is given as 
\[ \text{rrls}(P) = \{Q \mid Q \in \text{idrls}(P) \text{ such that } \text{parent}(Q) \cap \text{idrls}(P) = \emptyset\}. \]
5. The “cancel literal set of a literal \( P \)” is given as 
\[ \text{cls}(P) = \bigcup_{Q \in \text{rrls}(P) \cup \{P\}} \text{drls}(Q). \]
6. The “cancel literal set of a literal \( P \)” is given as 
\[ \text{ecls}(P) = \text{id}(P) - \text{drls}(P), \]
7. Given two generator literals, \( P \) and \( Q \), \( P \) and \( Q \) are said to be “independent” iff 
\[ (\text{rrls}(P) \cup \text{ecls}(P)) \cap (\text{rrls}(Q) \cup \text{cls}(Q)) = \emptyset. \]

### 4.3. Ng and Leung

Ng and Leung suggested yet another literal-level selective resetting scheme [10]. Even though their method utilizes run-time information, it still behaves less efficiently than ours (details can be found in [4]).

To make things worse, their scheme is “not complete”, that is, it may miss some valid solutions in some situations by omitting necessary resetting [4]. Since the method is incorrect, no comparison of it with ours will be attempted in the following sections.
4.4. Discussion

Both methods in Sections 4.1 and 4.2 statically analyze a clause and fix reset/canceled literals sets for each body literal. Thus those methods are more conservative than ours which utilizes run-time information (i.e., affection history). Of course, we must pay some run-time overhead for improved behavior.

For example, let us consider Fig. 1 again. In the first failure situation mentioned in Section 3.1, both methods reset no. 5 among the generator literals following the redone literal no. 2, even if resetting no. 5 is fruitless. All no. 5's old solutions were rejected by no. 8, and no. 8 does not consume any variables other than E, the generated variable of no. 5. If no. 5's solutions had been rejected by no. 7 instead of no. 8, no. 5 must be reset not to miss any possible combinations of values of C and E. Both schemes, however, cannot help resetting no. 5, since they cannot tell at compile time that no. 7 will not fail at run time.

It seems fairly clear that owing to run-time information, our scheme does fewer resetting and canceling than those in [2,11]. Additionally, more concurrent backward execution is possible.

In summary, the relation between many defined in [2,11] and this paper is listed below.

Theorem (relationships among sets).

1. \[ RRLS(P) \cap \text{CON} = \emptyset \{11\}; \]
   \[ S_{\text{RESET}}(P) \cap \text{CON} = \emptyset; \quad D_{\text{RESET}}(P) \cap \text{CON} = \emptyset. \]

2. \[ RRLS(P) \cap \text{CLS}(P) = \emptyset \{11\}; \]
   \[ S_{\text{RESET}}(P) \cap S_{\text{CANCEL}}(P) = \emptyset; \quad D_{\text{RESET}}(P) \cap D_{\text{CANCEL}}(P) = \emptyset. \]

(3) \[ \text{DRLS}(P) \cup \text{IDRLS}(P) = \text{RRLS}(P) \cup \text{CLS}(P) \{11\}; \]
   \[ D_{\text{RESET}}(P) \cup D_{\text{CANCEL}}(P) \subseteq \text{DRLS}(P) \cup \text{IDRLS}(P). \]

(4) \[ S_{\text{RESET}}(P) = \text{RRLS}(P); \quad D_{\text{RESET}}(P) \subseteq \text{RRLS}(P). \]

(5) \[ S_{\text{CANCEL}}(P) = \text{CLS}(P); \quad D_{\text{CANCEL}}(P) \subseteq \text{CLS}(P). \]

On the efficiency of the proposed scheme, we thought that our algorithm can be shown to be more efficient than other related works through implementation. As well known, in the AND/OR Process Model, a child OR process (and its child AND processes, and their child OR processes, and so on) exists, which corresponds to each vertex in the DDG. The DDG and affecting sets are data structures managed by an AND process, so updating them for selective resetting and fewer number of cancel messages seems to be less expensive than terminating and re-spawning numerous processes by unnecessary resetting and canceling.

We can think of several forms of implementation. After the backtrack literal is determined, it can be added to the affecting set of each remaining candidate backtrack literal, as in this paper or [6]. Alternatively, we can construct the affecting sets when they are actually needed later, after collecting information for such task during execution of an AND process, like [4,7,10].

4.5. Comparison by example

In this section, we demonstrate the efficiency of our algorithm with a concrete example.

Let us consider a logic program in Fig. 2. The DDG for its query is depicted in Fig. 3.


\[ p1(a0, b0). \quad p2(a0, c0). \quad p2(a0, c1). \quad p3(a0, d0). \quad p4(b0, e0). \]
\[ p4(b0, e1). \quad p5(b0, c1). \quad p6(c0, d0). \quad p6(c1, d0). \quad p7(d0). \]
\[ p8(c0, e0). \quad p8(c0, e1). \quad p8(c1, e0). \quad p8(c1, e1). \quad p9(d0, e0). \]

Fig. 2. An example program showing the efficiency of our scheme.
Table 1
A snapshot of Choe et al.'s relevant sets at event (4)

<table>
<thead>
<tr>
<th>Literal</th>
<th>Choe et al.'s S _AFFECTING</th>
<th>Choe et al.'s S _CANCEL</th>
<th>Choe et al.'s S _RESET</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>{p2, p3, ..., p9}</td>
<td>{p2, p3, ..., p9}</td>
<td>#</td>
</tr>
<tr>
<td>p2</td>
<td>{p3, p4, ..., p9}</td>
<td>{p5, p6, ..., p9}</td>
<td>{p3, p4}</td>
</tr>
<tr>
<td>p3</td>
<td>{p4, p6, p7, p8, p9}</td>
<td>{p6, p7, p8, p9}</td>
<td>{p4}</td>
</tr>
<tr>
<td>p4</td>
<td>{p8, p9}</td>
<td>{p8, p9}</td>
<td>#</td>
</tr>
<tr>
<td>p5</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>p6</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>p7</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>p8</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>p9</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

Then, the following sequence of events occurs in the AND process for the query, under the AND/OR Process Model.

(1) p1 succeeds with \{A/a0, B/b0\}. (Henceforth, we will abbreviate a literal to its predicate name only, when there is no danger of confusion.)

(2) p2, p3, and p4 succeed with \{A/a0, C/c0\}, \{A/a0, D/d0\}, and \{B/b0, E/e0\}, respectively.

(3) p6, p7, and p8 succeed with \{C/c0, D/d0\}, \{D/d0\}, and \{C/c0, E/e0\}, respectively.

(4) p5 and p9 fail with \{B/b0, C/c0\} and \{D/d0, E/e0\}, respectively.

Now failure took place. Furthermore, the failure situation is a multiple failure case (two literals no. 5 and no. 9 reported failure). To alleviate this failure situation, a backward execution algorithm must be run.

Let us assume that we adopt Choe et al.'s or Park et al.'s backward execution algorithms. Then, at this stage, the contents of the data structures used in their backward execution algorithms are shown in Tables 1 and 2, respectively.

Further, by some backtrack literal selection algorithm, let us assume that no. 2 and no. 4 are determined as the backtrack literals for no. 5 and no. 9, respectively. These two backtrack literals, however, cannot be redone in parallel, since they are not independent according to their definitions (note that \(S\_AFFECTING(p2) \cap S\_AFFECTING(p4) = \emptyset\) and \((\text{RRLS}(p2) \cup \text{CLS}(p2)) \cap (\text{RRLS}(p4) \cup \text{CLS}(p4)) = \emptyset\)). Thus, only the leftmost backtrack literal, i.e., p2, is redone. After that, literals in \(S\_RESET(p2)\) (or \(\text{RRLS}(p2)\)) are reset and those in \(S\_CANCEL(p2)\) (or \(\text{CLS}(p2)\)) are canceled.

Therefore, the following sequence of events (from (5) to (8)) occurs from now on. We would like to point out that Choe et al.'s and Park et al.'s algorithms exhibit exactly the same behavior as mentioned before. Note that the columns, \(S\_CANCEL\) and \(S\_RESET\), in Table 1 are the same.

Fig. 3. The DDG for the query in Fig. 2.
as the columns, CLS and RRLS, in Table 2, respectively.

(5) \(p_2\) is selected as the backtrack literal for \(p_5\).

(6) \(p_2\) is redone and succeed with \((A/a_0, C/c_1)\).

(7) \(p_3\) and \(p_4\) are reset.

(8) \(p_5, p_6, p_7, p_8,\) and \(p_9\) are canceled.

On the other hand, the contents of the data structures used in our backward execution algorithm, when \(p_5\) and \(p_9\) failed, are shown in Table 3.

Following our backward execution algorithm, the two backtrack literals \(p_2\) and \(p_4\) are independent of each other, because \(p_2 \notin \text{D\_AFFECTING}(p_4)\) and \(p_4 \notin \text{D\_AFFECTING}(p_2)\). So they can be backtracked concurrently. Therefore, the following sequence of events (from (5') to (6')) takes place.

(5') \(p_2\) and \(p_4\) are selected as the backtrack literals for \(p_5\) and \(p_9\), respectively.

(6') \(p_2\) and \(p_4\) are redone and succeed with \((A/a_0, C/c_1)\) and \((B/b_0, E/e_1)\), respectively.

After backtracking to \(p_2\) and \(p_4\), we must update the affecting sets of relevant literals, since new redo relations are set up between \(p_5\) and \(p_2\), and between \(p_9\) and \(p_4\). In this example, such relevant literals are \(p_1\) and \(p_3\). \text{D\_AFFECTING}(p_1), however, does not change since a newly affected literal \(p_2\) is already included in the set. For \text{D\_AFFECTING}(p_3), \(p_4\) and \(p_8\) (one of \(p_4\)’s children, which was not in \text{D\_AFFECTING}(p_3)) are added to the set. The result is shown in Table 4.

After updating affecting sets, we must reset

Table 2
A snapshot of our relevant sets at event (4)

<table>
<thead>
<tr>
<th>Literal</th>
<th>D_AFFECTING</th>
<th>D_CANCEL</th>
<th>D_RESET</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p_1)</td>
<td>(p_2, p_3, \ldots, p_9)</td>
<td>(p_2, p_3, \ldots, p_9)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_2)</td>
<td>(p_5, p_6, p_8)</td>
<td>(p_5, p_6, p_8)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_3)</td>
<td>(p_6, p_7, p_9)</td>
<td>(p_6, p_7, p_9)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_4)</td>
<td>(p_8, p_9)</td>
<td>(p_8, p_9)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_5)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_6)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_7)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_8)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_9)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

Table 3
A snapshot of Park et al.'s relevant sets at event (4)

<table>
<thead>
<tr>
<th>Literal</th>
<th>Park et al.'s DRLS</th>
<th>Park et al.'s IDRLS</th>
<th>Park et al.'s CLS</th>
<th>Park et al.'s RRLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p_1)</td>
<td>(p_2, p_3, \ldots, p_9)</td>
<td>(\emptyset)</td>
<td>(p_2, p_3, \ldots, p_9)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_2)</td>
<td>(p_5, p_6, p_8)</td>
<td>(p_3, p_4, p_7, p_9)</td>
<td>(p_5, p_6, \ldots, p_9)</td>
<td>(p_3, p_4)</td>
</tr>
<tr>
<td>(p_3)</td>
<td>(p_6, p_7, p_9)</td>
<td>(p_4, p_8)</td>
<td>(p_6, p_7, p_8, p_9)</td>
<td>(p_4)</td>
</tr>
<tr>
<td>(p_4)</td>
<td>(p_8, p_9)</td>
<td>(\emptyset)</td>
<td>(p_8, p_9)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_5)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_6)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_7)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_8)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_9)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

Table 4
A snapshot of our relevant sets at event (6)

<table>
<thead>
<tr>
<th>Literal</th>
<th>Ours D_AFFECTING</th>
<th>Ours D_CANCEL</th>
<th>Ours D_RESET</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p_1)</td>
<td>(p_2, p_3, \ldots, p_9)</td>
<td>(p_2, p_3, \ldots, p_9)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_2)</td>
<td>(p_5, p_6, p_8)</td>
<td>(p_5, p_6, p_8)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_3)</td>
<td>(p_4, p_6, p_7, p_8, p_9)</td>
<td>(p_6, p_7, p_8, p_9)</td>
<td>(p_4)</td>
</tr>
<tr>
<td>(p_4)</td>
<td>(p_8, p_9)</td>
<td>(p_8, p_9)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_5)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_6)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_7)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_8)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(p_9)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
and cancel some literals. The following events (from (7') to (8')) take place. Observe that no literals are reset and \( p_7 \) is not canceled in our backward execution, while \( p_3 \) and \( p_4 \) were reset and \( p_7 \) was also canceled in Choe et al.'s or Park et al.'s backward execution.

(7') No literals are reset.

(8') \( p_5, p_6, p_8, \) and \( p_9 \) are canceled (\( p_7 \) is not canceled).

5. Conclusion

We proposed another backward execution algorithm for the AND/OR Process Model. It issues fewer number of cancel messages and avoids useless resetting operations by means of run-time information. Also the scheme performs independent redoing and resetting more concurrently than other related works. The rationale of the proposed method is explained through a simple notion called affection. We think that the concept correctly grasps problems of backward execution and lucidly depicts intuition behind algorithms regarding backward execution. Comparison with related works using rather formalized notation is performed too.

Acknowledgment

The authors would like to thank the anonymous referees for valuable comments on an early draft. Also, the coordinating efforts of the communicating editor, Professor K. Ikeda, should be deeply appreciated. The first author further wishes to thank Miss Jiyun Lee for proofreading the first draft.

References


