Selective Reset in the AND Process

by

Will Winsborough

Computer Sciences Technical Report #684

February 1987

Errata for TR 684

p. 17:

for:

(* Check the sender's Activation ID to be sure the sender is not supposed to be dead. *)

read:

(* Check Pf's Activation ID to be sure Pf is not supposed to be dead. Also, Pf may be marked 'solved,' if the Reset Algorithm made current a member of the Eliminated List for Pf while Pf was processing a redo. In that case Pf's inability to generate a new solution can be ignored. *)

for:

2. If that algorithm reports "AND Process Fails,"

read:

2. Otherwise, run the Backtrack Literal Selection Algorithm with Pf as the input parameter. If that algorithm reports "AND Process Fails,"

p. 18:

for:

(Base Case), remove from the Backtrack Literal Set any members

read:

(Base Case), remove from the Backtrack Literal Set and mark 'solved' any members

p. 19:

for:

This Process has no synchronization requirements with the Failure Message Handler except that adding to and deleting from the Backtrack Literal Set musts be atomic operations.

read:

Execution of the Failure Message Handler must be atomic relative to execution of the Backtracking Algorithm. Deletion from the Backtrack Literal Set must be atomic relative to execution of the Failure Message Handler. Failure messages can be queued upon arrival and handled one at a time by the Failure Message Handler, subject to these synchronization constraints.

for:

(b) Otherwise, generate and record a new Activation ID and send a redo message with it to the process for Pl.

read:

(b) Otherwise, assign null to the Current Solution for Pl, generate and record a new Activation ID and send a redo message with that ID to the process for Pl. Mark Pl 'executing.'

Selective Reset in the AND Process

Will Winsborough

February 2, 1987

Abstract

We present the problem of selective reset in the AND process for AND parallel execution of logic programs. Selective reset is the natural dual to the selective backtrack problem that has received much attention in the literature [3,8,9,10,12]. A method for backward execution is given that performs selective reset and selective backtrack. The algorithms that comprise the method are proven correct for programs that define finite SLD-trees in the sense that, so long as the forward execution algorithms, OR process algorithms and data dependency graphs are correct, the method will find all solutions.

Because it essentially performs a depth first search of the SLD-tree, the AND/OR Process Model cannot be made to find all solutions in the general case (infinite SLD-trees). By constraining the OR process to explore alternative clauses in lexical order and by requiring that the AND process execute interdependent calls in lexical order, the AND/OR Process Model can be made to find the same solutions as Prolog and in the same order. This does not rule out AND parallelism. It just dictates which literal should be given the first chance to bind a shared variable. Further, using a variant of the reset algorithm described here, selective backtrack and reset can be incorporated without harmful deviation from Prolog's enumeration of query solutions. The only difference will be that occasionally the new method will continue to enumerate solutions after Prolog has entered an infinite loop.

We describe informally two alternative versions of the selective backtrack and reset method that reduce the overhead costs but that provide reduced selectivity. Handling of cut and calls with side effects is also addressed.

1 Introduction

Selective Reset is a device to improve execution of logic programs. It has its origins in [1] where a selective reset method is described for sequential interpreters. This paper describes a mechanism for selective reset as well as selective backtrack in the context of the AND process for AND parallel interpreters. The mechanism is called the *Witness Method* for selective backtrack and reset.

Previous methods to improve the performance of the AND process have taken advantage of independence among literals to improve backtrack literal selection. Just as literals that share no unbound variables can be solved (executed) simultaneously [5,4,6], they can often be backtracked independently [5,3,12,10]. The Witness Method for selective backtrack and reset offered here uses the same independence to lessen the amount of work undone by the reset that occurs each time backtracking takes place.

This presentation explains how the Witness Method for selective backtrack and reset works in the context of the AND/OR Process Model [5]. It will be clear, however, that the method is applicable to any AND process, AND parallel model that uses backtracking to handle non-determinism (as opposed to full OR parallelism [9] or committed choice [7]).

1.1 Background

1.1.1 The AND/OR Process Model

We summarize the salient aspects of the AND/OR Process Model. For a complete discussion see [5].

As execution of a user query proceeds in the AND/OR Process Model, a tree of AND and OR processes is created. The root is an AND process and the tree's levels alternate between AND and OR processes. Each AND process tries to solve one query consisting of the conjunction of one or more literals. (We will use the terms call and literal, and solve and execute interchangeably, as seems more natural to the context.) The AND process's query is either a top level, user query or the body of a rule invoked to solve a literal in the grandparent AND process's query. In the latter case, the query is actually the substitution instance of the rule's body under the mgu (most general unifier) of the call with the rule's head.

To execute a call, P_i, the AND process creates a sub-process called an OR process. The OR process unifies P_i with either a fact or a rule-head. In the former case the OR process can report success and a solution to the AND process immediately. In the latter it must create a AND sub-process to solve the rule-body and wait to hear from that descendant.

A *solution* to a literal, P_i, consists of a substitution on the variables of P_i. For example, a solution to smallcity(X) might be {X/urbana}, indicating that the substitution instance of smallcity(X) where X is replaced by urbana can be proven from the program clauses.

In general, if another literal P_j shares a free variable with P_i, we force P_i and P_j to execute sequentially to avoid binding conflicts. Assuming P_i comes before P_j and they share the free variable X, we will call P_i a *generator* of X and P_j a *consumer* of X. We will not start execution of P_j until P_i has completed and the substitution that is the solution to P_i has been applied to P_j.

So, for example, if we have the query,

?- smallcity(X), notTooFlat(X).

we will execute the calls sequentially. If a consumer fails, it is necessary to try it with other solutions to the generator applied. To get those other solutions in the process model, the AND process initiates *backward execution* in response to the failure message from the consumer's OR process. The AND process sends a *redo* message to the OR process for the generator. That OR process must find and report another solution.

In our example, the first solution, {X/urbana}, causes the consumer, notTooFlat(X), to fail. This is because the substitution instance of the consumer in force when execution of the call begins cannot be proven from the program clauses. (Henceforth we shall refer to a substitution instance of a call in force when execution of the call starts as an *activation instance.*) To handle the failure of the activation instance notTooFlat(urbana), the AND process sends a redo message to the OR process for smallcity(X), and a new solution is generated (or the call fails, if no more solutions can be found). We say the first solution has been *eliminated.* If the new solution, perhaps {X/boulder}, induces an activation instance of the consumer that succeeds, then the consumer *accepts* the solution. Otherwise, the generator must be redone again. In the case that no solutions to the generator are acceptable to the consumer, the generator will be redone until no new solutions can be found.[1] The OR process then reports failure. One of two things is done by the AND process to handle the generator's failure. If the generator is also the consumer of a solution generated by a third literal, backward execution is performed recursively. Otherwise, there are no solutions to the query and the AND process reports the query's failure.

Sometimes it is necessary for an AND process to cancel an executing call because the activation instance is no longer viable. For example, if we have the query

?- r(X), s(X), t(X).

r may instantiate X to a ground term, say c. In that case, s and t may be executed concurrently, since no binding conflicts can arise. But if one of them, say s, should fail, X/c would be rejected as a solution to the query. Thus the activation instance t(c) would no longer be of interest. In this sort of circumstance, the AND process sends a *cancel message* to the OR process for t, and t is marked 'waiting.' It, like the failed consumer, s, can be restarted when the generator, r, has been successfully redone.

1.1.2 The Reset Problem

The above description of backward execution does not explain how to handle the case where a literal consumes bindings from several generators. In order

[1]Unless otherwise indicated we assume all calls generate only a finite number of solutions. For a discussion of infinite solution sequences see Section 4.

that all possible query solutions be explored, the consumer must be tried on all combinations of solutions to the generators. There are two problems, each related to ensuring all possible solutions are explored. First, which generator should be redone? Second, when that backtrack literal has been redone, which other generators must be *reset*, that is started over in the enumeration of their solution set?

An example may illuminate. If we have the query

```
?- P(X), Q(Y), R(X,Y).
```

we need to cycle through bindings for X and Y generated by P and Q, trying R with the various combinations. Retaining Conery's nested loops paradigm [5], we will try each Y-binding with the first X-binding, then each with the second, and so on. This is achieved in the Model as follows. When R fails, rejecting the first solution pair, the AND process eliminates the current solution to Q (a Y-binding) and sends (the OR process for) Q a redo message to get a new solution. If none of those combinations are acceptable to R, (the OR process for) Q will eventually respond to the redo with a failure message. When that happens, each Y-binding has been eliminated, and the AND process must try the second X-binding with each Y-binding. So, when Q reports failure, P is redone to find the second X-binding, and Q is reset to make its eliminated solutions, the Y-bindings, available again and to make one of those Y-bindings Q's current solution. (It was Conery [5] who pointed out that the Y-bindings do not have to be recomputed if the AND process just saves them. This is correct because the activation instance of Q remains the same.) Determining which literals to reset when a redo occurs is the reset problem.

There are simple solutions to the backtrack literal selection problem and to the reset problem. They are analogous to naive backtracking in sequential Prolog: take as the backtrack literal the call immediately to the left of the failed call, and reset all calls to the right. These rules are correct, but wasteful. Efficient backtrack literal selection has been handled for the AND process by [10,12]. We present a method for performing reset more selectively.

Existing reset methods (of which we are aware) simply make available all eliminated solutions to all calls to the right of the redone call[2]. It may be that many of those solutions were eliminated for reasons having nothing to do with the redone call, and so will certainly be eliminated again (see Section 1.2). This paper presents a reset method that avoids the wasted effort caused by such excessive conservatism. It shows that correct selective reset can be accomplished using an amount of overhead comparable to that for selective backtracking (see Section 5.2).

We do not present the details of forward execution, or of the OR process. Nor do we specify exactly how the generator/consumer relation is determined.

[2] A Technical Report by Vipin Kumar and Yow-Jian Lin, not yet available, apparently contains some suggestions as to how to achieve selective reset.

For discussion of these issues see [5,4,6]. But the generator/consumer relation is central to understanding selective reset. Following the example of Conery and others, we represent that relation by a data dependency graph.

A *data dependency graph* for a query is a directed acyclic graph with one node for each call in the query. We will call a data dependency graph *correct* for the query execution if it contains an arc for each *possible* (generator, consumer)-pair from the generator to the consumer. Here *possible* means possible in any of the execution sequences that may arise while the graph is in force. Thus, in general, if a data dependency graph is defined before execution begins it must be more restrictive of concurrency than if it is defined at runtime for each clause activation, when more is known about which variables must be generated.

The example execution snap shots shown in Figures 2, 4 and 5 of Sections 1.2 and 1.3 adorn data dependency graphs with the arcs labeled by the variable involved in the (generator, consumer)-pair. We represent the configuration of the AND Process that way because the significance for selective backtrack and reset of the process state is only revealed in light of the data dependencies.

We will resume the topic of data dependency graphs briefly in Section 1.3. But first we consider some examples of the reset problem.

1.2 Examples

1.2.1 Why Selective Reset is Useful

Figure 1 gives a program that illustrates why selective reset is desirable. Using several descendant OR processes, some calls can execute concurrently, so there are many sequences of events that could occur in the AND process for the program query. Here is one possible sequence leading to a redo.

1. P_1, P_2 and P_4 succeed: A/a_1, B/b_1, C/c_1
2. P_5 fails
3. P_4 is redone: C/c_2
4. P_3 fails
5. P_2 is redone: B/b_2

In the naive reset method this redo would cause an unnecessary reset to be performed.

Figure 2 shows the configuration of the AND process when P_3 reports failure. It shows the current and eliminated solutions of literals that are solved. The set ($\{P_1\}$) following the eliminated solution is called a witness set. Witness sets are used to record sufficient history information to allow selective reset and backtrack. How this is achieved will become clear in Section 1.4.

When P_3 fails and P_2 is redone, the naive reset method makes C/c_1 an available solution to P_4. But this is unnecessary because that solution was rejected by P_5 independently of the solution to P_2. It is undesirable because, when P_5 eventually rejects C/c_2, if C/c_1 is available it will be made current only

1. $P_1(a_1)$ 2. $P_2(a_1, b_1)$ 3. $P_2(a_1, b_2)$ 4. $P_3(b_2)$ 5. $P_4(a_1, c_1)$ 6. $P_4(a_1, c_2)$ 7. $P_4(a_1, c_3)$ 8. $P_5(c_3)$

?- $P_1(A)$, $P_2(A, B)$, $P_3(B)$, $P_4(A, C)$, $P_5(C)$.

Figure 1: A Simple Program Requiring Only One AND Process

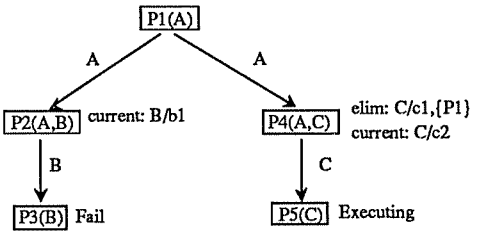


Figure 2: AND Process Configuration Before A Redo Requiring No Reset During Execution of the Program in Figure 1

to be rejected a second time. On the other hand, if C/c_1 is not available, C/c_3 will be tried next, speeding execution toward success.

In the simple case of Figure 1, it is easy to determine from the data dependency graph shown in Figure 2 which literals (the empty set) must be reset when P_2 is redone. Assuming the data dependency graph is fixed statically, as described in [4], some selective reset advantage can usually be compiled into the code for performing reset. In the example of Figure 1, since no literal consumes more than one variable there is no need to cycle through all combinations of solutions to variables consumed by a single literal. Thus, no reset ever need be performed.

1.2.2 Why Discuss a Method that Incurs Runtime Overhead

Chang et al. [4] mention a way to perform selective reset (as well as selective backtrack) according to pre-execution analysis in the context of statically determined data dependency graphs. In Section 5.3 we present the static version of the Witness Method and indicate how the correctness of that method could be derived using the proof, given in Section 3, of the correctness of the dynamic Witness Method. The advantage of statically determined reset is low runtime overhead. The disadvantage is a loss of selectivity for some queries.

Chang's method appears to be identical to the static version of the Witness Method. The presentation in [4] of a static method does not constitute an

1. $P_1(a_1)$ 2. $P_2(a_1, b_1)$ 4. $P_3(a_1, c_1)$
3. $P_2(a_1, b_2)$ 5. $P_3(a_1, c_2)$

6. $P_4(b_2)$ 7. $P_5(b_1, c_1)$ 9. $P_6(c_3)$
8. $P_5(b_1, c_2)$

?- $P_1(A)$, $P_2(A, B)$, $P_3(A, C)$, $P_4(B)$, $P_5(B, C)$, $P_6(C)$.

Figure 3: A Program That Benefits From *Dynamic* Selective Reset

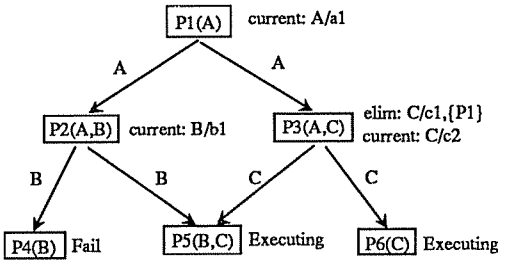


Figure 4: AND Process Configuration Before A Redo Requiring No Reset During Execution of the Program in Figure 3

algorithm nor does it argue why the method is correct. However, this paper, in Section 5.3, only discusses briefly how the algorithm and proof would be done. The motivation for this paper's focus on the algorithm and correctness proof for the *dynamic* version of the Witness Method is twofold.

First, the dynamic version can be applied in a system where the data dependency graph is determined dynamically. While such a system incurs additional runtime overhead over a static graph system, it allows a greater degree of parallelism to be realized, which, for some queries, will outweigh that additional overhead.

Second, the dynamic version, even if used with statically fixed data dependency graphs, can perform reset more selectively than can the static version. Kumar and Lin [10] have already shown that a dynamic backtrack algorithm can be more selective than Chang's static one, even when coupled with statically defined data dependency graphs. We now give an example illustrating why the same is true of the reset problem.

Figure 3 gives a program that illustrates why we can get better selectivity from the dynamic approach. In this case the program query fails. Suppose the following sequence of events takes place in the AND process for the program:

1. P_1, P_2 and P_3 succeed: A/a_1, B/b_1, C/c_1
2. P_6 fails
3. P_3 is redone: C/c_2
4. P_4 fails
5. P_2 is redone: B/b_2

The configuration before the last redo (line 5) is shown in Figure 4. Upon performing that redo, the naive method would reset C/c_1 as a solution to P_3. This is unnecessary because C/c_1 was rejected by P_6 independently of the solution to B. Reseting C/c_1 only causes it to be rejected again. So this query fails more quickly if C/c_1 is not reset.

In this example, a static method for selective reset would have to reset the solutions to P_3 because in general it cannot know before execution whether those solutions will be rejected by P_5 or by P_6. It must take the safe action of providing all solution combinations to P_5.

1.3 The Data Dependency Graph

1.3.1 Statically versus Dynamically Constructed Data Dependency Graphs

Correctness of a data dependency graph for a query was defined at the end of Section 1.1: the graph must contain an arc for each possible (generator, consumer)-pair. There can be many correct data dependency graphs. For example, the linear graph (total ordering of calls) is always correct, though it permits no concurrency. As mentioned in Section 1.2, Chang has presented a method of determining correct data dependency graphs at compile time using static program analysis[4]. This avoids runtime overhead, but may miss potential concurrency. On the other hand, Conery [5] defines the data dependency graph incrementally as execution of the calls proceeds. In this way more concurrency can be realized at the cost of runtime graph construction overhead.

The Witness Method is applicable in tandem with both data dependency graph definition mechanisms mentioned above, as well as with intermediate approaches such as in [6]. Of course it is required that the data dependency graph be correct. But it may be defined incrementally, at runtime, as call executions occur and complete.

1.3.2 Redo Can Be Handled Like A Failure

By incorporating the clause head into the data dependency graph for the clause, the same backward execution method can handle failure messages (from below in the computation tree) and redo messages (from above). This was done in [5]. So for the program clause

$$\text{head(X,Y)} \text{ :- } P_1\text{(X,Z)}, P_2\text{(X,W)}, P_3\text{(W,Z,Y)}, P_4\text{(Y)}.$$

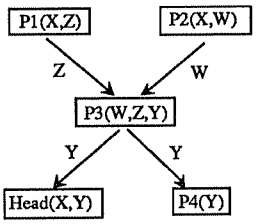


Figure 5: Data Dependency Graph With a Dummy Consumer Node for the Clause Head

and the call, head(1,Y), we may have the data dependency graph shown in Figure 5. (In this example P_1 and P_2 may run concurrently because the only variable they share, X, is already ground.) A redo message can be handled exactly as a failure from the 'dummy' call to head, selectively backtracking directly to P_3, the generator of Y rather than trying to resolve P_4, a call that cannot change the clause solution.

1.3.3 Lexical Order of Calls Should Not Be Violated

Execution order among calls with shared variables affects the solutions found. For a depth-first search strategy such as the AND/OR Process Model, this is true even of pure Horn clause programs. The Prolog primitives not and var aggravate the problem. In order to ensure that the solutions found do not depend on the data dependency graphs used, we must insist that the partial order on the calls defined by the data dependency graphs be consistent with the (total) lexical order. This guarantees that whenever a synchronization constraint must be imposed to avoid binding conflicts, the calls involved will be executed in the same (user specified) order. This requirement is necessary to ensure that the (operational) semantics do not depend on which data dependency graph is used.

1.4 A Refinement of the Selective Reset Problem

Thus far we have behaved as though all solutions to a literal must be reset if any are reset. We now refine that view and define the aim of selective reset to be as follows. When a call is successfully redone, make available only those eliminated *solutions* to later calls that were rejected for a reason that the redo could have removed. To illustrate this idea we return to the data dependency graph displayed in Figure 4. Again, suppose we redo P_2 and want to reset selectively. To carry out our refined understanding of selective reset we make available the eliminated solutions of P_3 that were rejected by P_5, but not those that were rejected by P_6. The idea of resetting individual solutions is similar to the suggestion in [1] that individual clauses be reset separately.

The intuition behind the Witness Method is as follows. Following Conery's

example, we keep eliminated solutions on a list, the Eliminated List, so they don't have to be recomputed. But we associate with each solution some information that allows each solution to be considered for reset independently. That information is a subset, called a *witness set*, of the literals to the left of the call in question.

The reader may object: one set per solution is considerably more information than is required for selective backtracking alone. Section 5.2 shows that one can maintain just one witness set for all solutions (cf. the redo cause set of [12] or the B-list of [10]) and still perform selective reset. However, one must be willing to reset either all or no solutions to a literal.

1.5 Witness Sets

Before considering the formal specification of the algorithms, let us discuss the concept at their base.

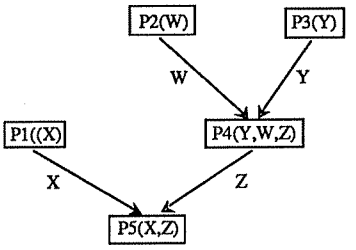
Suppose $P_1, ..., P_i, ..., P_n$ is a query in some AND process, and σ is an eliminated solution to P_i. A *witness set* for σ is a subset, W, of $\{P_j | j < i\}$ such that σ and W have the *Witness Property*: *if the current solutions to the members of the witness set,* W, *are applied, then the associated solution,* σ, *cannot participate in a solution to the query.*

The Witness Property is related to the property used in [2]. The witness set contains essentially similar information to the Redo Cause Set of [12] and the B-list of [10]. Besides being what we need for selective reset, witness sets are sufficient to allow the dual, selective backtrack.

The idea is that if a literal, P_l, to the left of some P_r is redone we can tell whether a solution, σ_r, for P_r must be made available again based on whether P_l is in the witness set associated with σ_r. If P_l is not in σ_r's witness set, there is no point to trying to apply σ_r again: the current solutions to the calls in the witness set are not changed by the redo, so σ_r will be rejected for the same reason as before.

It is always possible to construct a correct witness set for any rejected call-solution so that all members precede the call. This is because the AND/OR Process Model requires the the data dependency graph to be acyclic.

To illustrate the construction and use of witness sets, consider again the data dependency graph displayed in Figure 4. Suppose this time the corresponding query is posed to a data base of program clauses where P_5 fails first. This behavior allows us to conclude that as long as all the ancestors of P_5, $\{P_1, P_2, P_3\}$, are solved with their current solutions, P_5 will fail. The Backtrack Literal Selection Algorithm presented below always chooses to redo the lexically rightmost literal that could remove the cause of failure. So whenever P_5 fails, P_3 will have its current solution, call it σ_3, eliminated. The witness set for σ_3 is then $\{P_1, P_2\}$ because so long as the current solutions are applied to P_1 and P_2, if σ_3 is applied to P_3, P_5 will certainly fail, causing σ_3 to be eliminated. Thus the Witness Property holds. The witness set for σ_3 becomes useful if P_3 fails or if P_2 is redone.



?- $P_1(X)$, $P_2(W)$, $P_3(Y)$, $P_4(Y,W,Z)$, $P_5(X,Z)$

Figure 6: A Query and its Data Dependency Graph

Consider the case where P_3 fails. We know that σ_3 is eliminated whenever the current solutions to P_1 and P_2 are applied. But if a different solution to P_2 were applied, σ_3 might be accepted by P_5. [3] Thus, we conclude to redo P_2, the rightmost literal that is either in a witness set for P_3 or is an ancestor of P_3. In the dual case, where P_2 gets a new solution, the witness set for σ_3 allows us to recognize that σ_3 might now be accepted by P_5 and we therefore reset σ_3, making it again available as a solution to P_3. (Such a situation would arise if, for example, P_4 were to fail, causing P_2 to be redone.)

Let us consider how witness sets might be constructed. When an executing consumer call, P_c, reports failure, a solution, σ, to a previous call will be eliminated. We must record in the witness set for σ all the literals that could affect the failure of P_c because, so long as their current solutions are not changed, whenever σ is applied, P_c will fail, rejecting σ. On the other hand, once a literal that could affect the failure of P_c has its current solution changed, it may be that P_c will succeed, even with σ applied. So we can no longer rule out σ.

So the problem of constructing a witness set for σ is reduced to determining which literals can affect the failure of P_c. Clearly P_c's ancestors are among this collection, since they determine the activation instance of P_c being solved. But, the witness sets for the eliminated solutions to P_c must also be included.

Consider P_4 in Figure 6. P_4 may fail because all its solutions are rejected by P_5. By the above argument, such eliminated solutions would have P_1 in their witness sets, since they have to be tried again if P_1 gets a new current solution. When P_4 fails, rejecting some solution to P_3, σ_3, P_1 must go in the witness set for σ_3. This is because if P_1 got a different solution, P_4 might not fail, so σ_3 would not necessarily be rejected. So when P_4 fails, its rejected solutions' witness sets must be incorporated into the witness set for σ_3. P_2 also goes in the witness set for σ_3 because it is an ancestor of P_4, the literal whose failure caused σ_3 to be rejected.

[3] If a different solution were applied to P_1, P_2 and P_3 would have new activation instances, so their processes would have to be canceled and restarted.

This example illustrates the principle that when a failure of P_c causes σ to be rejected, σ's witness set should include all of the witness sets for eliminated solutions to P_c and all ancestors of P_c.

1.6 The Rest of this Paper

In Section 2 we present a group of algorithms that perform backward execution according to the Witness Method. Section 3 proves that the algorithms do not cause any solutions to be missed, so long as the program has only a finite SLD-tree. In Section 4 we argue that, even in the presence of infinite SLD-trees, by using a slight variant of the algorithms in Section 2, and by placing some restrictions on the OR process, the AND/OR Process Model using the Witness Method for backward execution can be made to perform similarly to and at least as well as Prolog in the presence of infinite SLD-trees. Section 5.1 addresses correct handling of programs with side-effects. In Sections 5.2 and 5.3 we discuss versions of the Witness Method that may be less selective but that require less overhead.

2 Algorithms

We now describe bottom-up a collection of algorithms to handle failure and all aspects of backward execution. Section 2.1 introduces some formal definitions that will be used somewhat in Section 2, but more in Section 3. Section 2.2 introduces the data structures used in the formal algorithms. Sections 2.4 through 2.8 present the algorithms themselves. Roughly, these interact as follows. The Failure Message Handler is invoked when a failure occurs. It selects the backtrack literal using the Backtrack Literal Selection Algorithm and records it with a witness set in a data structure called the Backtrack Literal Set. The members of that set are removed and processed by the Backtracking Algorithm, which saves the rejected solution on the Eliminated List, sends the redo message and performs reset and cancellation as necessary. Section 2.9 then discusses some alternatives in the details of the algorithms.

2.1 Definitions

Let G denote the set of literals to be solved by the AND process. Let n be the cardinality of G. We assume G to be linearly ordered and denote members of G by P_i, for $1 \leq i \leq n$, with the understanding that P_i precedes P_j in the linear order *iff* $i < j$.

Next we define the mutually recursive notions of a *query-solution* and a *call-solution.*

Let V_i be the set of variables occurring in the call P_i. Let σ be the mgu of P_i with a program clause. Then the restriction of σ to V_i is a call-solution for

P_i if the clause is a fact, and $\sigma\tau$[4] restricted to V_i is a call-solution if the clause is a rule and τ is a query-solution for the rule body under σ.

A *query-solution*, Σ_G, to a query, G, relative to the order of the calls in G, is a composition of substitutions,

$$\Sigma_G = \sigma_1\sigma_2 \ldots \sigma_n$$

where σ_i is a call-solution to $P_i\sigma_1\sigma_2 \ldots \sigma_{i-1}$.

We will drop the prefixes, query- and call-, when the meaning is clear from context.

In a given query, a *witness set* for a rejected solution to a call in the query is a subset of the calls to the left. It and the rejected solution must have the Witness Property. A *legacy set* is the same thing as a witness set, except that it is used in the Backtrack Literal Set data structure, rather than on an Eliminated List. (See below.) We draw the distinction because a witness set is paired with a rejected solution whereas a legacy set is paired with a literal whose current solution will be rejected when the Backtracking Algorithm can get to it.

2.2 Data Structures

The Ancestor Set, Descendant Set, and Literal Order Number are data structures that must be established for each literal in the AND process. The Ancestor Set and the Descendant Set reflect the data dependency graph for the query. Any method for constructing the data dependency graph will do, subject to the considerations in Section 1. However, the Ancestor and Descendant Sets for a literal must be constructed by the time the literal is executed. Of course the performance is likely to vary with the quality of the data dependency graphs used and the overhead required to generate them.

- Literal Order Number: positive integer. This is the lexical order of the literals in the query, fixed when the program is written, numbered starting with one. Recall that the partial ordering defined by the data dependency graph is always extensible to this total order.
- Ancestor Set: set of literals. Each literal has an Ancestor Set containing its ancestors in the data dependency graph.
- Descendant Set: set of literals. Each literal has a Descendant Set containing all the literals whose input values are dependent on the given literal's output substitution. If a fixed data dependency graph is used, this set is just the descendants in that graph. Otherwise, the descendents must be known by the time the literal is started executing, though it may not be known how those descendants are ordered among themselves.

[4]We use the definitions of substitution application and composition found in [11], in which the application of σ to E is written $E\sigma$ and which makes $(E\sigma)\tau = E(\sigma\tau)$, for all expressions E.

The following process data structures are maintained by the algorithms presented below. Each literal has an Eliminated List, an Available List, a Current Solution, and a Status. Each executing or solved literal has an Activation ID. The AND process has one Backtrack Literal Set.

- Eliminated List: list of (call-solution, witness set)-pairs. Each literal has an Eliminated List containing solutions to the current substitution instance of the literal that have been rejected by their consumers. When the AND process is created the Eliminated List is empty. It is also emptied every time the values that the literal consumes are changed.

- Available List: list of solutions. Each literal has an Available List containing previously computed solutions to the current substitution instance of the literal. These have been transferred from the Eliminated List by the reset operation. When the AND process is created the Available List is empty. It is also emptied every time the values that the literal consumes are changed.

- Current Solution: call-solution. Established by the Forward Execution Mechanism or by the Reset Algorithm. This is the only solution consumers can work with until it is eliminated and replaced. The Current Solution is null before the call first succeeds, after it is canceled, or when it has failed.

- Status: one of 'waiting,' 'executing,' 'solved,' and 'failed.' Each literal is marked with its current status. 'Waiting' means the call has no solutions yet but can be run as soon as all solutions it consumes are available. We assume the forward execution algorithm handles starting these calls when all dependencies are satisfied and changes there status to 'executing' at that time. We also assume the forward execution algorithm fields success messages, installs the solutions as the current solution, and changes the status to 'solved.' A call marked 'executing,' 'solved,' or 'failed' can have solutions on the Eliminated and Available Lists. A call marked 'solved' must have a current solution but a call marked 'waiting,' 'executing' or 'failed' must not. Since the Reset Algorithm (see Section 2.7) may change the status of a (redone) literal from 'executing' to 'solved,' we assume that upon receiving a success message from an OR process for a call marked 'solved' the forward execution algorithm simply appends the new solution to the Available List.

- Activation ID: a unique identifier that is associated with a process when it is marked 'executing.' This complication is necessary to ensure correctness. When a call is marked 'executing,' and its OR process is sent a start or a redo message, an Activation ID that is unique to this activation is associated with it. The same ID is sent to the OR process for this call and will be included in all messages from that OR process regarding this call activation. As Conery has pointed out, it is necessary to be sure the

message received in the name of this call is for the correct activation of the call.

- Backtrack Literal Set: ordered set of (backtrack literal, legacy set)-pairs. There is one Backtrack Literal Set per AND process. The presence of a pair indicates that the backtrack literal must be redone because its current solution has been rejected. The legacy set will become the witness set for that solution on the Eliminated List. The Backtrack Literal Set is a holding tank for failure information. The Failure Message Handler (Section 2.5 below) adds to the Backtrack Literal Set in response to receipt of a failure message. The Backtracking Algorithm (Section 2.8 below) removes the (backtrack literal, legacy set)-pairs one at a time, using their contents to update the AND process state and to send the necessary redo message.

  The legacy set component of the pair is a kind of witness set: the Witness Property holds for the legacy set and the *current solution* of the literal component. So if the current solution is changed for whatever reason, the entry for that literal (if one exists) must be removed from the Backtrack Literal Set.

  The Backtrack Literal Set is initially empty. Its members are ordered by the Order Number associated with their literal component.

There are three reasons for buffering (backtrack literal, legacy set)-pairs, rather than just the failed literals, to represent the failures remaining to be handled.

First, it is possible that two or more consumers of a generated variable should fail in quick succession, the second failure arriving before the first can be handled. If they both indicate the same generator must be redone, their failure information must be pooled. To see this, suppose each failure were recorded separately. If one failure record were handled independently, causing a new solution to be found for the generator, the other failure record would erroneously appear to reject that new solution. The pair representation allows us to maintain at most one failure record for each backtrack literal. The Failure Message Handler recognizes duplicate backtrack literals and pools the failure information by combining the legacy sets.

Second, redoing one literal may remove the (possible) cause of several failures. By recording the failure with an explicit legacy set it is possible to discover quite easily when this occurs. By the Witness Property, if redoing a literal could remove the cause of failure for some failure recorded in the Backtrack Literal Set, that literal will be in the legacy set of the failure record. The Reset Algorithm (see Section 2.7 below) checks for this and removes failure records from the Backtrack Literal Set as necessary.

Third, it is advantageous to treat first the left most literal whose redo is called for in the Backtrack Literal Set.

2.3 Another Definition

By *available solutions* for a call we will mean solutions on the Available List and solutions not yet reported by the sub-process for the call. Additionally, we include the current solution so long as the call is not in the Backtrack Literal Set.

2.4 Backtrack Literal Selection Algorithm

The Backtrack Literal Selection Algorithm is called by the Failure Message Handler.

Input Parameters:

1. P_f: A literal. P_f has failed.

Output Parameters:

1. (backtrack literal, legacy set)-pair. The correct literal to redo and the witness set for its Current Solution. The Current Solution will be eliminated by the Backtracking Algorithm.

Locals:

1. Redo Set: A set of literals that could remedy the failure.
2. Backtrack Literal: The literal selected for backtracking.
3. Legacy Set: Will become witness set for the Backtrack Literal's Current Solution.
4. RightMost: A function that takes a set of literals and returns the one occurring latest in the linear ordering.

Procedure:

1. Redo Set := Ancestor Set$(P_f) \cup \bigcup_{(\sigma,W)\in \text{Eliminated List}(P_f)} W$.
   (* Redoing any one of these literals could remove the cause of failure. *)
2. if Redo Set is empty then exit algorithm with the message "AND Process Fails."
   (* No literals within the query could remove the cause of failure. *)
3. Backtrack Literal := RightMost (Redo Set).
   (* Use the nested loops paradigm for enumeration of possible solutions. The rightmost literal is the inner loop, so we select it to get a new solution. *)
4. Legacy Set := Redo Set - {Backtrack Literal}
5. exit reporting (Backtrack Literal, Legacy Set).

2.5 Failure Message Handler

The Failure Message Handler is invoked upon receipt of a failure message from a descendant OR process.

Input Parameter Set:

1. P_f: A literal. P_f is the literal reporting failure.

Procedure:

1. If the sending process is not marked 'executing' or the Activation ID in the failure message does not agree with the current Activation ID recorded for P_f in the AND process, then exit.

   (* Check the sender's Activation ID to be sure the sender is not supposed to be dead. *)

2. If that algorithm reports "AND Process Fails,"

   (a) then send a message to the parent process reporting failure and cancel all subprocesses.

   (* The Backtrack Literal Selection Algorithm was unable to find a call inside the AND process that could remove the cause of failure. *)

   (b) Otherwise, call the resulting (backtrack literal, legacy set)-pair (P_b, W_{new}). Mark P_f 'failed.'

   (* Record the result in the Backtrack Literal Set for subsequent processing by the Backtrack Algorithm. *)

   If there is already an entry in the Backtrack Literal Set for P_b,

   i. then call its witness set W_{old} and replace the old entry with $(P_b, W_{old} \cup W_{new})$.

   (* Only want one entry per call, so just add new suspects to old legacy set. *)

   ii. otherwise, add the new pair to the Backtrack Literal Set.

2.6 Cancel Algorithm

The Cancel Algorithm is called by the Backtracking Algorithm.

Input Parameters:

1. P_c: a literal. P_c has had its input bindings changed and so must be canceled.

Procedure:

1. Empty both the Eliminated List and the Available List of P_c.

(* A canceled call cannot have any solutions, available or eliminated. *)

2. Assign null to the Current Solution.
3. Remove the Backtrack Literal Set entry for this literal, if one exists.

   (* That entry refers to the rejection of a solution to a problem that we are no longer interested in. *)

4. Send a cancel message to the OR process for P_c.
5. Mark P_c 'waiting.'

2.7 Reset Algorithm

The Reset Algorithm is called by the Backtracking Algorithm.

Input Parameters:

1. i: integer. The Order Number of the first literal to check for resetting.
2. R: a set of literals. R contains the literals that have had their solutions changed during this reset, including the redone call.

Constants:

1. n: an integer. n is the number of literals in the AND process's query.

Procedure:

1. if $i > n$ (Base Case), remove from the Backtrack Literal Set any members whose legacy set components intersect R. Then exit.

   (* An element (P_r, LS) of the Backtrack Literal Set represents the rejection of the current solution to the literal P_r. If a member of the legacy set LS has a new current solution then the Witness Property no longer holds. The solution can no longer be rejected, so the (P_r, LS) must be removed from the Backtrack Literal Set. Note that this does not constitute a new solution to P_r, so no reset need be initiated when the entry is removed. *)

2. Otherwise ($i \leq n$), if P_i's Eliminated List contains any entries such that the witness set component, WS, satisfies $WS \cap R \neq \phi$, then
   (a) move all such solutions to the Available List (discarding witness set components),
   (b) and, if P_i is marked 'failed' or 'executing,' then

      (* P_i now has available solutions so we must make one current. Some eliminated solutions to the right may have to be reset because P_i has a new current solution, so we add P_i to R for the recursive call. *)

i. remove one member of the Available List and make it the Current Solution,

ii. mark P_i 'solved,' and,

iii. recursively call Reset with $i+1$ and $R \cup \{P_i\}$ as the input parameters.

(c) Otherwise (P_i is *not* marked 'failed' or 'executing,' i.e., P_i already has a current solution), call Reset with $i+1$ and R.

3. Otherwise ($WS \cap R = \phi$ for all WS on P_i's Eliminated List), call Reset with $i+1$ and R.

2.8 Backtracking Algorithm

We write this algorithm as though it were a subprocess that continually executes or waits throughout the AND process's life. Alternative implementations are plausible. This process has no synchronization requirements with the Failure Message Handler except that adding to and deleting from the Backtrack Literal Set must be atomic operations.

Locals:

1. P_l: a literal. P_l the least literal in the Backtrack Literal Set.
2. Legacy: a set of literals. Legacy is the witness set associated with P_l.

Procedure:

1. Wait until the Backtrack Literal Set is not empty.
2. Remove (P_l, Legacy), the element whose first component has the least Literal Order Number, from the Backtrack Literal Set.

   (* Choosing the least element is not necessary for correctness, it just improves efficiency. *)
3. Put P_l's current solution on the Eliminated List with Legacy as its witness set.
4. If P_l's Available List is not empty,
   (a) then take the first solution off the list and make it current.
   (b) Otherwise, generate and record a new Activation ID and send a redo message with it to the process for P_l.
5. Run the Cancel Algorithm on each member of P_l's Descendant Set.
6. Run the Reset Algorithm with $\{P_l\}$ as the input parameter.

2.9 Alternatives in the Algorithms

We could have canceled the failed literal's (P_i's) descendants in the Failure Message Handler. But then if the Reset Algorithm were to remove the failure record for that failure, because of some other redo, we would have to start the canceled process's descendents over and repeat the lost work.

In the Backtracking Algorithm it is not necessary to perform reset upon sending a redo to the selected process if, instead, reset is performed when a redo succeeds. To facilitate this, the call status value 'executing' would be refined to 'executing a started call' and 'executing a redone call.' Then the forward execution algorithm could tell whether reset should be performed when the call succeeds. Delaying reset until success is reported would avoid some overhead in the case that the redone process reports failure. But it is not clear whether it is more efficient to eagerly reset and allow the consumers of reset solutions to run ahead so far as they may, or to wait until it is known how far the failure will reach and which processes will be canceled.

We could avoid sending a redo message to the OR process when it has already failed by recording a pseudo entry on the Available List upon receiving the first failure message. Once recorded, the pseudo entry remains the last member of the Available List until the call is canceled. When a new solution to the call is sought by the Backtracking Algorithm, if the only entry on the Available List is the pseudo entry, there is no point in sending another redo message. The call has already reported failure, so backtracking can continue. The Backtracking Algorithm then has to call the Backtrack Literal Selection Algorithm to find a new backtrack literal and legacy set.

3 Correctness of the Algorithms

We assume the Witness Method does not interfere with the soundness of the AND/OR Process Model, since it implements resolution. The problem is to show (partial) completeness.

In this section we show, assuming correctness of the data dependency structures, the forward execution algorithms and the OR process, that the algorithms in Section 2 allow generation of all solutions to the AND process's query, provided the SLD-tree defined by that query is finite. In Section 4 we will argue informally that under certain restrictions to the Model, the AND/OR Process Model using the Witness Method for backward execution finds a superset of those solutions found by the standard depth first interpretor, even in the presence of infinite SLD-trees.

Define a *correct solution sequence* for a query, G, to be an enumeration of the multiset of substitutions corresponding to success leaves in the SLD-tree for G. We will say that a call P_i *generates* a solution sequence if the sequence of solutions generated in response to repeated redo messages is that sequence and, in the case of finite solution sequences, the next redo message evokes failure.

Lemma 1 Let G be the set of calls in some query, $P_i \in G$. If P_i fails and R is a subset of G having the property that "if the current solution to each element of R is applied, P_i will fail," then some element of R must be redone before a solution to G can be found.

Proof Obvious. ∎

Lemma 2 Each (Solution, Witness Set)-pair put on a Eliminated List by the Backtracking Algorithm has the *Witness Property*, *viz.*, "if the current solutions to the calls in the witness set are applied, then the corresponding solution cannot participate in a query-solution."

Proof Use induction on the *failure level* of the witness set. Failure level was defined in [10] for literals. We define and use the same concept as it applies to witness sets:

1. If a witness set was generated by the failure of a literal that produced no solutions, the *failure level* of the witness set is Zero.
2. Otherwise, let the failure of the literal P_f cause the rejection of the solution σ_r for the literal P_r. Let Ω be the set of witness sets for solutions on P_f's Eliminated List when P_f failed. Then the *failure level* of the witness set for σ_r is $n+1$ where

$$n = max\{FailureLevel(W) | W \in \Omega\}.$$

First we treat the single rejection case where the Backtrack Literal Selection Algorithm never selects a Backtrack Literal that already occurs in a (backtrack literal, legacy set)-pair in the Backtrack Literal Set. Then we show the multiple rejection case, where the Failure Message Handler modifies an existing pair by adding to its legacy set, is also correct.

Basis Suppose a literal, P_f, fails without generating any solutions. This implies that the current substitution instance of P_f cannot be solved. Any correct data dependency graph has the property that all literals that affected the substitution instance of P_f are ancestors of P_f, so we may conclude that, "so long as the current solutions are applied to P_f's ancestors, P_f will fail."

In this case (failure level = 0) the Backtrack Literal Selection Algorithm takes the rightmost ancestor, call it P_r, to be the Backtrack Literal and the other ancestors, call that set W, to be the Legacy. So long as the current solutions to the calls in W are applied, if the current solution to P_r, call it σ_r, is applied then P_f will fail. Therefore, W and σ_r have the Witness Property because, so long as the current solutions to W are applied, σ_r cannot participate in a query-solution.

Step Assume Lemma 2 for all witness sets of level $\leq$n. Let P_f be a literal that, as it fails, has witness sets with failure level n but no higher. Then by the induction assumption we have that, for each solution on P_f's Eliminated List the Witness Property holds: "if all the members of the witness set are solved with their current solutions, then the corresponding solution on the Eliminated List cannot participate in a query solution."

The Backtrack Literal Selection Algorithm takes the union of all the witness sets on P_f's Eliminated List and the Ancestor Set to be the Redo Set. Call that Redo Set R. Since R is a superset of (correct) witness sets for all solutions to P_f, we have that, "if the current solutions are applied to all members of R, then no solutions to P_f can participate in a query-solution." Suppose the Backtrack Literal Selection Algorithm selects $P_r \in R$ to be the backtrack literal. Call the current solution to P_r σ_r. Together, σ_r and the set $W = R - \{P_r\}$ have the property that "so long as the current solutions are applied to W and σ_r is applied to P_r, the no solutions to P_f can participate in a query-solution." This implies that "so long as the current solutions are applied to W, σ_r cannot participate in a query-solution." Thus, σ_r and W have the Witness Property we set out to prove.

Multiple Rejection We now show that the Failure Message Handler maintains the Witness Property for elements of the Backtrack Literal Set in the presence of multiple rejections. Specifically, if (P_r, *legacy*) is a pair in the Backtrack Literal Set with the Witness Property holding for *legacy* and the current solution to P_r, and some P_f fails, causing (P_r, *legacy'*) to be returned by the Backtrack Literal Selection Algorithm, then the Witness Property holds for (P_r, *legacy* $\cup$ *legacy'*). But this is trivial, since the antecedent, "the current solution to the literals in *legacy* $\cup$ *legacy'* are applied," is stronger than the antecedent of the Witness Property for *legacy* and P_r, which we have, and the consequent is the same.[5] ∎

Lemma 3 The Reset Algorithm correctly restores to the Available Lists all the call-solutions on the Eliminated Lists for all literals P_k, $k \geq i$, that could possibly participate in a query-solution involving the current solutions to P_j, $j < i$, provided the input parameter R contains all literals P_j, $j < i$, that have new solutions.

Additionally, the Reset Algorithm makes available (by removing the corresponding literals from the Backtrack Literal Set) those current solutions

[5]For the purpose of selective reset, better behavior might be achieved by leaving the old, stronger legacy set in place. This would be correct for selective reset, but the legacy set is also used for selecting the backtrack literal. For that purpose we cannot neglect the members of *legacy'* − *legacy* lest some query solutions be missed. This suggests a more aggressive (and expensive) version of the Witness Method that uses separate data structures for these two purposes.

that could participate in a query-solution involving their predecessors' current solutions (because redoing or resetting changed one of those current solutions.)

When a redo and a reset are done, they may remove the cause of some failure that has occurred but has not been handled yet by the Backtrack Algorithm. Lemma 3 says the Reset Algorithm removes the artifact of such failures from the Backtrack Literal Set.

Proof The proof is by induction on $k = n - i + 1$, the number of literals remaining to be explored by the Reset Algorithm.

Basis $k = 0$. Here we show the second part of the lemma, that the Reset Algorithm correctly removes entries from the Backtrack Literal Set. Any literal's current solutions could participate in a query-solution involving its predecessors' current solutions when the Backtrack Literal Set is empty. When a literal is added to the Backtrack Literal Set its current solution cannot participate in a query-solution involving the current solutions to its legacy set. But if the current solution to one of those legacy members is changed, as when it is backtracked or reset, then the current solution to the literal could possibly participate in a query-solution. It is in exactly these circumstances that the Reset Algorithm removes an entry from the Backtrack Literal Set.

Step Assume for $k - 1$, show for k. There are three cases on Pi:

Case 1 $WS \cap R \neq \phi$ for some WS on P_i's Eliminated List and P_i's Current Solution is null. By Lemma 2 it is sufficient to only put on the Available List the solutions whose witness sets intersect R. The Reset Algorithm does reset those solutions in line 2.a. The others can remain eliminated since none of their witnesses have been redone. Because the Current Solution to P_i is null before the reset, it is changed afterwards. Therefore, it must be added to R for the recursive call so that the induction assumption may apply. The Reset Algorithm does that, so this case is correct.

Case 2 $WS \cap R \neq \phi$, but P_i's Current Solution is not null. This only differs from case 1 in that the Current Solution to P_i is not changed. Thus, P_i does not have to be added to R for the induction assumption to be applied to the recursive call.

Case 3 $WS \cap R = \phi$ for all WS on P_i's Eliminated List. In this case Lemmas 1 and 2 tells us no solutions need be reset. So the algorithm is correct in its handling of P_i's solutions. Since P_i's Current Solution is not changed it does not have to be added to R for the induction assumption to apply to the recursive call. ∎

For the subsequent results we use implicitly the following assumptions:

1. The program defines a finite SLD-tree.
2. Correctness of Forward Execution Algorithm.
3. Correctness of Data Dependency Graphs.
4. Correctness of the OR process (or whatever mechanism is used to step through alternative rules).

We use the first assumption because, by soundness, it implies that calls terminate and generate finite solution sequences. Although these weaker properties are all that our proofs require, it seems unattractive to characterize the instances for which the correctness assertions hold in terms of the behavior of our algorithm on those instances. The latter three assumptions reflect the limited scope of this paper.

Lemma 4 Let G be the set of calls in some query. Define $S_i = \{P_j \in G | j < i\}$ and $U_i = \{P_j \in G | j \geq i\}$. Let Σ_i be a solution to S_i. Suppose all $P_i \in G$ generate correct solution sequences. (By assumption these must be finite.) And suppose all call-solutions that participate in solutions to G are available solutions [6]. Then, in response to a sequence of redo messages for the query, an AND process using the algorithms presented in Section 2 will, first, generate a correct solution sequence for $U_i\Sigma_i$, and, then, a failure will be reported (by a call in U_i) that generates a Redo Set having no elements in U_i.

Proof Use induction on $k = |U_i|$. We assume the data dependency graph for U_i is the corresponding subgraph of the data dependency graph for G.

Basis When $k = 1$ the solutions to $U_n\Sigma_i$ are exactly the solutions to $P_i\Sigma_i$. By the assumption that $P_i \in G$ generates correct solution sequences, all solutions to $U_n\Sigma_i$ are found.

Step By definition, P_i is the first element in U_i. By assumption, the call to $P_i\Sigma_i$ generates a correct solution sequence. Let σ_i be the first solution to $P_i\Sigma_i$ generated. The induction assumption may be applied to $(U_i - \{P_i\})\Sigma_i\sigma_i = U_{i+1}\Sigma_i\sigma_i$. Thus we have that all query-solutions for U_i that involve the first call-solution to P_i are generated and a failure occurs with a redo set containing no members of U_{i+1}. If the redo set does not contain P_i then a failure has occurred which, by Lemmas 1 and 2, cannot be removed by replacing P_i's solution. Therefore, in that case, there are no more solutions to $U_i\Sigma_i$. The algorithm has found all the solutions to $U_i\Sigma_i$, and

[6] Recall that a solution to a call is available if either it is on the Available List for the call, it has not yet been reported by the descendant process for the call or it is the current solution to the call and there is no entry for the call in the Backtrack Literal Set.

failed with a redo set containing no members of U_i. If, on the other hand, the redo set does contain P_i, then P_i is redone, since the Backtrack Literal Selection Algorithm uniformly selects the last element of the redo set to be the Backtrack Literal. A new solution to P_i is found, according to the correct call-solution sequence assumption. By Lemma 3 we have that the Reset Algorithm will make available all solutions to members of U_{i+1} that could participate in a query-solution. So we may again apply the induction assumption as we did for the first solution to P_i and conclude that all query-solutions involving the second solution to P_i are found.

This line of reasoning is repeated until the redo set provided by the induction assumption does not contain P_i or another solution to $P_i\Sigma_i$ is requested and the call fails. The former case is shown correct above. In the latter we certainly have generated all query-solutions, and the failure of P_i will generate a redo set containing no members of U_i. By assumption $P_i\Sigma_i$ only generates finitely many solutions, so we have that, in a finite number of steps, a failure occurs that generates a redo set having no elements in U_i. ▮

Corollary Assuming the calls in G generate correct solution sequences, all solutions to G are found by an AND process using the Witness Method described in Section 2. When all solutions have been found, a redo message will prompt a failure response.

Proof When an AND process is created, all solutions are available solutions. By taking $i = 0$ we have S_i empty, $U_i = G$, and Σ_i the identity substitution. Therefore, Lemma 4 can be applied directly. ▮

Theorem Under the assumptions mentioned on page 24, an AND process executing a query G using the Witness Method responds to a sequence of redo messages by enumerating the multiset of solutions corresponding to the success leaves of the SLD-tree for G.

Proof This just says the hypothesis, "assuming all calls in G generates correct solution sequences," can be dropped from the Corollary. The proof is a simple induction on the maximum depth the AND process tree reaches during computation. We shall measure the depth of the AND process tree by the number of AND processes on the longest branch. The maximum depth reached during computation exists and is finite because of the assumption that the SLD-tree for G is finite.

Basis An AND process tree of depth 1 contains one AND process, all of whose calls are solved by facts. By assumption the OR process correctly steps through applicable clauses. So the calls generate correct solution sequences and the Corollary applies.

Step By the induction hypothesis, the AND process's OR sub-processes are provided with correct solution sequences for all applicable clause bodies. We assume those OR processes correctly report those solutions, so the Corollary applies. ▮

4 Behavior in the Presence of Infinite SLD-trees

The AND/OR Process Model does a depth first search of the SLD-tree, as does Prolog. For this reason it is not a complete interpretor: some solutions may not be found, though they are logical consequences of the data base (and hence, by completeness of SLD-resolution, correspond to a success leaf somewhere in the SLD-tree). When more than one rule can be resolved with an activated call the various alternatives are represented in the SLD-tree by different subtrees (see [11]). The AND/OR Process Model, like Prolog, explores each subtree exhaustively before proceeding to the next. In general, these subtrees may not be finite, so the alternatives may never be explored. This problem is inherent in the search strategy, and no amount of intelligent backtracking can aviod it.

But Prolog programmers have developed techniques for controlling the structure of the SLD-tree and the order in which the subtrees are searched. While these techniques remove Prolog from the ideal plane of declarative languages, there is a considerable number of Prolog enthusiasts, nonetheless. A complete yet efficient interpretor is still a problem for research. So we cater to the Prolog programmer by offering a variant of the Witness Method that generates more or less the same solutions in the same order as Prolog. The difference between the Witness Method Variant and Prolog is that the Variant may occasionally continue to find more solutions past the point where Prolog has been caught in a non-terminating computation that generates no solution. Unless and until that comes to pass, the Variant will generate the same solutions in the same order as Prolog.

To allow solutions to be generated in Prolog order we must impose two restrictions on the forward execution portion of the AND/OR Process Model. The first restriction, compatibility with lexical call order, was discussed in Section 1.3.3, above. The second is to require the OR process to report solutions according to the lexical order of the clauses that generated them. It must report first all the solutions reported by the AND process for the first matching clause, and so on. It also must preserve the order in which each AND process reports its solutions. This drastically limits the opportunity for OR parallelism, though some may still be possible in the form of eager generators.

Now we describe the change required in the backward execution algorithms given in Section 2, above.

4.1 A Witness Method Variant

As it was presented in Section 2.7, the Reset Algorithm leaves in place the current solution of a literal even when it makes available a solution that was reported earlier. This has the advantage of not interrupting consumers of the current solution, but it can cause the sequence of solutions to depend on the Backtrack and Reset Method, the strength [7] of the witness sets, and the data dependency graph. If the Reset Algorithm is changed to merge the reset members of the Eliminated List into the Available List according to the order in which they were reported by the OR process, and to preempt the current solution as necessary to be sure the current solution is the first available solution, then we claim the solutions would be found and reported in Prolog order.

Induction on the height of the AND/OR process tree and the fact that the matching clauses are explored in Prolog order allow us to conclude that, so long as the calls terminate, solutions to calls are generated in Prolog order. The main point in the argument is that both Prolog and the Variant Method try call-solution combinations according to the nested loops model [5]. By the induction assumption and the sorted merging of solutions by the reset operation, we have that, after each backtrack, each call's list of available solutions agrees with the order that Prolog would discover them in, except that some call-solutions that cannot lead to successful query-solutions are omitted. Since the combinations of viable call-solutions are explored in the same (nested loops) order as they would be in Prolog, the successful combinations must be found in that same order. Hence, the query-solutions must be reported in the same, Prolog order.

So, up until the time it executes a call that fails to terminate, the Variant will report the same solutions as Prolog.

Infinite paths in the SLD-tree cause termination problems for depth first search in two ways: either some call fails to terminate or some call that cannot remove the source of failure is backtracked to but succeeds every time it is given a redo message (it has an infinite solution sequence), preventing backward execution from reaching a point that could remove the source of failure. The Witness Method Variant may avoid execution of some non-terminating call activation instances by deducing that the combination of predecessor solutions cannot participate in a query solution for some other reason. It also may skip redoing a call because it cannot remove the cause of failure. In this way, the Witness Method Variant may occasionally continue to find solutions after Prolog is in an infinite loop. On the other hand, if the Variant executes a call that never terminates, the only way Prolog can avoid execution of the same call is to already be caught in another infinite call.

We hope this informal argument is convincing.

There are two reasons we did not present this Variant in Section 2. First, we want it to be clear that the Witness Method is applicable to the original

[7] In general a variety of witness sets have the witness property for a given rejected solution. The smaller witness sets are stronger, since they result in fewer resets being performed.

AND/OR Process Model, as proposed in [5]. That model does not attempt to mimic Prolog's enumeration of solutions, so the overhead of merging Available Lists in sorted order and cancelling consumers of preempted current solutions would be wasted effort. Second, the algorithms are hard enough to follow as they are currently presented. It seems pedagogically preferable to discuss controlling enumeration order separately.

5 Variations and Further Remarks

5.1 Subtrees With Side Effects

Skipping a call that cannot remove the source of failure is fine when the call has no side effects. But calls that can cause side effects, like print or assert, must not be skipped and they must always recompute their solutions, not just copy them from the Eliminated List. Otherwise, the behavior of the program will be affected by selective backtracking and reset.

To ensure correct behavior, calls with possible side effects must be treated as synchronization bottlenecks for the whole query in which they appear. All previous calls must succeed before executing the synchronized call and all successive calls must wait to start until the synchronized call successfully completes. Consider, for example,

?- generate(X), test(X), print(X).

We must ensure test(X) accepts a solution to generate(X) before printing it. For correct execution of the query

?- assert(P(X)), P(Y).

it is necessary to complete the assert before going onto P(Y). A simple, sufficient solution is to make all its predecessors ancestors of the call with side effects and to make it an ancestor of all its successors. This ensures both synchronization constraints and it forces the call to be canceled and reexecuted, instead of merely reset, when backtracking through the call with side effects.

Unfortunately, we must treat specially not only the primitive calls that cause side effects, but every call that could indirectly cause such a primitive to be activated. This necessitates a reachability analysis be performed before creating the data dependency graph. It may be necessary to repeat that analysis each time new clauses are added to the data base. This may be plausible for additions that take place between queries, though it might require static data dependency graphs to be revised. But what about assert and retract calls that change the reachability relation during query execution? We suggest requiring the predicate symbols in the assert calls to be constants. Then the reachability analysis can be done under the worst case assumption that all clause (templates) that could be asserted will be. (Note that, thus far, these problems are not peculiar to

the Witness Method, but reflect a basic conflict between side effects, on the one hand, and concurrency and selective backtrack and reset.)

Also unfortunately, asserts can make a previous failure unreproducible. It may no longer be that "if all the current solutions to calls in the witness set are applied, then the call fails," because the assert has changed the program. This destroys the correctness of the witness sets. So long as all predicates being asserted are compile-time constants, we may avoid this problem. Calls to predicates that can reach an assertable call must be treated differently: all predecessors that can reach calls to assert must be made their ancestors in the data dependency graph. This will make all such predecessors members of the Redo Set when the assertable call fails. Any solutions eliminated because of that failure will be made available again as soon as any predecessor that can reach a call to assert succeeds.

Cut can only be incorporated if the model imposes Prolog's clause-try order with clauses that contain cuts. If the OR process is made to try each clause with cuts before its lexical successors, the cut can be implemented. Since it has side effects, cut must be treated as described above. The effect of executing a cut requires a new kind of message to be sent by the OR process to its parent AND process. The Cut message tells the parent to remove the alternative matching clauses from further consideration.

5.2 Less Strong Witness Sets Allow Faster Reset

If the overhead of checking each solution's witness set independently is not worthwhile, we can just store the union of the witness sets for all solutions to a literal and reset either all or no solutions, according to that one set. When a literal fails, the union is identical to the redo set constructed by the Backtrack Literal Selection Algorithm and so can be used to select the backtrack literal. Since the union contains all witness sets, it has the Witness Property for all solutions to the literal. So correctness follows from the results in section 3. Of course, some solutions will be reset this way that would not be were the witness sets kept separately for each solution.

The cost of this version of the Witness Method for selective backtracking and reset minimally exceeds that for selective backtracking alone [10,12]. No more space is required except an amount of local storage for the recursive Reset algorithm's local variables linear in the number of calls in the query. Realistically, this is a small constant since clause bodies and top level queries are bounded in length in practice, if not in theory. The time requirement beyond that for selective backtracking alone is, for each redo and each literal to the right of the redone literal, the time for one set intersection (witness set $\cap$ literals with new solutions), one set test for emptiness (of intersection to decide whether to reset the call's solutions), and one set union (if the current solution is changed). Note that the selective-backtrack/naive-reset methods in [10,12] visit most literals to the right of the redone call just to make all solutions available again.

Since the sets range over a small universe, the literals in the query, a bit-vector implementation could be utilized to make each of those set operations efficient.

5.3 Static Witness Sets

In a system such as that of [3,4], where the data dependency graph is statically fixed, we can minimize the runtime overhead by also fixing the witness sets statically. Static versions of witness set are constructed to contain all literals that the Backtrack Algorithm could ever possibly put there. This can be accomplished by looking at the ancestor sets of all literals in the dependency graph. Let R be the relation containing a pair of literals, (P_i, P_j), if P_i and P_j appear in the ancestor set for some third P_k and $i < j$. Note that (P_i, P_j) is in R just in case P_i could be in one of P_j's witness sets. Let R^* be the transitive closure of R. (P_i, P_j) is in R^* just in case redoing P_i could possibly call for P_j to be reset.

If this method is used then, rather than searching witness sets, we can determine at compile time which literals should be reset if a given P_i is redone. It is all P_j such that $(P_i, P_j) \in R^*$

This is not an algorithm, but it does indicate how one would be constructed. A simplification of the proof of Lemma 2 could be used to show the analogous result: let $W_j = \{P_i | (P_i, P_j) \in R^*\}$; so long as the current solutions to members of W_j are applied, the eliminated solutions for P_j cannot participate in a query-solution. That is the core of the correctness proof. The balance depends on the actual reset algorithm.

5.4 Further Research

We plan to explore the trade-off between increased selectivity and overhead. Besides the less exact methods suggested in sections 5.2 and 5.3 there are more exact ways of constructing the witness sets.

We plan to assess empirically the usefulness of these various methods for different forms of query. It is conjectured that through static analysis the compiler may be able to select different approaches for different rules in the program and hence only pay the high overhead of the more elaborate methods in the cases where they are likely to be worthwhile. This will be facilitated by the natural boundary that the AND Process Model draws around each rule-body's execution.

5.5 An Existing Model That Does Selective Reset

In [8] a semi-intelligent (i.e., somewhat selective) backtracking scheme is presented for Restricted AND parallelism [6]. That method achieves a somewhat selective reset in a very natural way. In Restricted AND parallelism, the form of possible data dependency graphs is limited. Roughly, the last paths to part must be the first to rejoin. Semi-intelligent backtracking is achieved for Type I

backtracking[8] by selecting as the backtrack literal the last call that must have completed before the failed call was started. To handle Type II backtracking, the AND process records how far forward execution has reached in each branch of the dependency graph. The right most ancestor of that point that is to the left ot the failed literal is selected as the backtrack literal. Parallel branches that are independent of the redone call are not reset.

It is not necessary to retain failure history information as in this work (the witness sets) and as in [10,12]. Selective behavior can be achieved by a simple recursive backtrack algorithm that requires no information to be saved between invocations. It comes as no surprise that the data dependency graphs given in the counter examples [10,12] to Conery's [5] original backtracking method cannot occur in Restricted AND parallelism.

There is no doubt that Restricted AND parallelism is an important model because, as M. V. Hermenegildo has shown, it allows highly efficient implementation. But, before dismissing the work of [10,12] and the work presented here, the reader is encouraged to consider the limitations of the semi-intelligent backtracking presented in [8] when attempting to solve a problem with a highly connected data dependency graph, such as the map coloring problem, in Restricted AND parallelism. The ability of Restricted AND parallelism to represent the real data dependencies depends on their simplicity. As the complexity of the graph increases the selectivity of the backtrack and reset diminishes.

6 Conclusion

We have presented algorithms for backward execution that perform selective backtrack and reset. We call that scheme the Witness Method for Selective Backtrack and Reset. The algorithms have been proven correct for programs with finite SLD-trees. We have argued that the AND/OR Process Model with the Witness Method can be restricted so as to find roughly the same solutions as does Prolog and in the same order.

We have indicated alternative variants of the method that generate solutions in Prolog's order and that require less overhead. Also, we have indicated how to handle calls that could have side effects so that selective backtrack and reset do not change the semantics of the program.

7 Acknowledgements

We want to thank Charles N. Fischer for substantial assistance with this presentation. If the paper is at all readable, it is largely due to his criticism and

[8]Type I backtracking [3] occurs when a call fails without generating any solutions. Type II backtracking is when some solutions were eliminated and so the failure is in response to a redo.

suggestions. We also thank Ken Kunen, Charles Stewart and Felix S.-T. Wu for many helpful discussions of the ideas offered here.

References

[1] Maurice Bruynooghe, "Analysis of Dependencies to Improve the Behaviour of Logic Programs," *Conference on Automated Deduction*, 1980, pp. 293–305.

[2] M. Bruynooghe, L. M. Pereira, "Deduction Revision by Intelligent Backtracking," *Implementations of Prolog*, Ellis Horwood, 1984, J. A. Campbell, ed., pp. 194–215.

[3] Jung-Herng Chang, Alvin M. Despain, "Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis," *IEEE Symposium on Logic Programming*, 1985, pp. 10–21.

[4] Jung-Herng Chang, Alvin M. Despain, Doug DeGroot, "AND-Parallelism of Logic Programs Based on a Static Data Dependency Analysis," *IEEE Spring CompCon*, 1985, pp. 218–225

[5] John S. Conery, *The AND/OR Process Model for Parallel Interpretation of Logic Programs*, PhD Thesis, University of California Irvine, 1983.

[6] Doug DeGroot, "Restricted And-Parallelism," *ICOT International Conference on Fifth Generation Computer Systems*, 1984, pp. 471–478.

[7] Steven Gregory, *Design, Application and Implementation of a Parallel Logic Programming Language*, PhD Thesis, Imperial College of Science and Technology, 1985.

[8] M. V. Hermenegildo, R. I. Nasr, "Efficient Management of Backtracking in AND-Parallelism," *Third International Conference on Logic Programming*, 1986, Springer-Verlag, "Lecture Notes in Computer Science," Vol. 225.

[9] Laxmikant Vasudeo Kale, *Parallel Architectures for Problem Solving*, PhD Thesis, State University of New York at Stony Brook, 1985.

[10] Yow-Jian Lin, Vipin Kumar, Clement Leung, "An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs," *Third International Conference on Logic Programming*, 1986, Springer-Verlag, "Lecture Notes in Computer Science," Vol. 225.

[11] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1986, pp. 18-19.

[12] Nam Sung Woo, Kwang-Moo Choe, "Selecting the Backtrack Literal in the AND/OR Process Model," *IEEE Symposium on Logic Programming*, 1986, pp. 200–210.