UNIVERSITY OF CALIFORNIA

Irvine

The AND/OR Process Model for Parallel Interpretation of Logic Programs

John S. Conery

Technical Report 204

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science

June 1983

Committee in charge:

Professor Dennis Kibler, *Chair*
Professor Lubomir Bic
Professor James Neighbors

Dedication

For my teachers --

For my parents, Pat and Elaine, both secondary school teachers, who started this whole process;

For Betty Benson, John Sage, and the other outstanding teachers at John Burroughs High School, who inspired me to continue;

For Larry Rosen and Jay Russo, valued friends and colleagues as well as teachers at UC San Diego, who introduced me to the world of scientific research.

This work does not represent the end of my education; it is simply a milestone in the career of a "professional student" who will, thanks to you, never stop learning.

Contents

List of Figures

Acknowledgements

I was very fortunate in my graduate career to have an exceptionally supportive group of faculty and fellow graduate students at UC Irvine. Among the most influential were Lubomir Bic, Kurt Eiselt, Steve Fickas, Doug Fisher, Gene Fisher, Peter Freeman, Les Gasser, Kim Gostelow, Steve Hampson, John King, Rob Kling, George Lueker, Jim Meehan, Wil Plouffe, Maria Porcella, Rami Razouk, Walt Scacchi, Tim Standish, Bob Thomas, and Steve Willson. Conversations with Paul Morris, Jim Neighbors, and Bruce Porter, who patiently listened to numerous detailed and boring descriptions of "AND parallelism," are especially appreciated.

I also benefited by communication, written and otherwise, with researchers outside of UCI. Keith Clark, of Imperial College, London, provided some helpful feedback in the early days of this research. In the Summer and Fall of 1982, talks with Arvind of MIT, Andrezej Ciepielewski and Seif Haridi of the Royal Institute of Technology, Stockholm, and John Glauert, of the University of Manchester, were quite fruitful.

The ICS Department has been very generous with its resources, human and otherwise. The staff, Rose Allen, Pat Harris, Susan Hyatt, Fran Paz, Peggy Rose, Sue Rose, and Phyllis Siegel, all gave me a great deal of help. The computing resources available for research have grown tremendously in the last few years, due mostly to the efforts of the department chair, Julian Feldman. I would never have been able to accomplish so much without the use of these systems. Thank you, Julian.

Finally, I owe a special debt to my advisor, Dennis Kibler. Our relationship has been more along the lines of collaborators with mutual research interests, rather than the traditional student/faculty member relationship. He has been an incredible source of ideas, encouragement, restraint, motivation, and friendship for many years. I hope to see our collaboration continue in the future.

Abstract of the Dissertation

The AND/OR Process Model
for Parallel Interpretation of Logic Programs

by

John S. Conery

Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1983
Professor Dennis Kibler, Chair

Current techniques for interpretation of logic programs involve a sequential search of a global tree of procedure invocations. This dissertation introduces the AND/OR Process Model, a method for interpretation by a system of asynchronous, independent processes that communicate only by messages. The method makes it possible to exploit two distinct forms of parallelism. OR parallelism is obtained from evaluating nondeterministic choices in parallel. AND parallelism arises in the execution of deterministic functions, such as matrix multiplication or divide and conquer algorithms, that are inherently parallel. The two forms of parallelism can be exploited at the same time. This means AND parallelism can be applied to clauses that are composed of several nondeterministic components, and it can recover from incorrect choices in the solution of these components. In addition to defining parallel computations, the model provides a more defined procedural semantics for logic programs; that is, parallel interpreters based on this model are able to generate answers to queries that cause standard interpreters to go into an infinite loop. The interpretation method is intended to form the theoretical framework of a highly parallel non von Neumann computer architecture; the dissertation concludes with a discussion of issues involved in implementing the abstract interpreter on a multiprocessor.

CHAPTER 1

Introduction

A growing area of research in computer architecture is the design of High Level Language Machines. The motivation is to design systems that provide better support for the language constructs used by both application and operating system programmers.

This motive can be explained in terms of a "semantic gap" between the formalisms of the machine language and the programming language [42]. The larger the gap, the harder it is to implement languages reliably and efficiently. For example, a typical construct in a programming language is the array. Implementations of arrays must take into account storage functions which map an array index into a location in memory, range checking (making sure that the value of I in the index expression A[I] is within the bounds of the array A), representation of arrays of complex objects of varying sizes, and so on. If some of these functions are implemented in the architecture, as is the case in a machine that performs a range check with a single machine instruction, then programs written in the high level language can be interpreted more reliably, and possibly even more efficiently.

A new and exciting line of research within the area of language oriented architecture is the design of machines for functional programming languages. These are languages that are based on more abstract formalisms than the von Neumann model of the stored program. John Backus, in the 1978 ACM Turing Award Lecture, gave an eloquent comparison of functional languages and von Neumann languages [3]. What makes functional languages so attractive to computer architects is that there is a great potential for parallel evaluation. These languages are concerned with the definition of functions, and rules for constructing complex functions from primitive functions, but not specifically with the order in which the functions must be evaluated.

For example, one way of writing a matrix multiplication program in the FP language is

def MM $= \alpha\ \alpha$ **IP** $\circ$ **Dist**

where **IP** is a function that computes the inner product of two lists of integers, and **Dist** is a function that transforms two matrices into appropriate pairs of rows and columns [3]. The composition of functions **F** and **G**, written **F** $\circ$ **G**, means that **F** should be applied to the object created by **G**. This program specifies that **IP** is to be applied to all of the row/column pairs created by **Dist**, but does not specify the order of these applications, and in fact leaves open the possibility that a multiprocessor could evaluate the inner products in parallel.

The existence of abstract models of computation other than the stored program model enables system designers to apply the familiar top-down methodology to the design of computer architectures. Instead of building machines to support languages that are not much more than abstractions of earlier machines (a rather circular approach), computer architects can start from abstract levels and move step by step towards a concrete implementation of that model. The steps in this top down approach (which has also been called the *language first* philosophy [53]) can be summarized as follows:

-- Select an interesting abstract model of computation.

-- Design a high level programming language based on the selected formalism.

-- Define a method for interpreting programs of the language; if parallelism is a goal of the project, define an interpreter that can carry out a number of steps in parallel.

-- Finally, design a computer architecture tuned to the requirements of the interpreter defined in the previous step; if parallelism is a goal, a requirement of this step is to show how the independent parts of the program can be delivered to the independent processing elements in the architecture.

This dissertation is a contribution to the area of language first computer architecture, starting from the formalism of logic programming, with the long range goal of constructing a multiprocessor for logic programs.

A simplified description of logic programming is that one can use sentences of first order logic as statements of a programming language. Many useful programs have been written in Prolog (an acronym for *pro*gramming in *log*ic), a high level programming language based on the formalism. Among these applications are metacompilers [61], machine learning programs [32], natural language queries for databases [16, 62], and robot problem solving systems [58]. Prolog has also been used as an executable specification language. The expectation is that since Prolog is as abstract as many formal specification languages, it should be possible to write the functional specifications for new programs using the syntax of Prolog; the big advantage to using Prolog is that it is then possible to test the specifications by executing them as programs. Davis' thesis involved generating LISP or Pascal programs from functional specifications written in Prolog [18]. Most recently, Prolog has been identified as the major programming language for the Fifth Generation Computer Project now starting in Japan.

Standard implementation techniques for Prolog are described in terms of a search of a global data structure that represents possible execution paths. One result of this dissertation research is a method of interpretation of logic programs that allows for parallel execution. The model, named the AND/OR Process Model, is flexible enough to obtain the maximum amount of parallelism possible in deterministic programs, and still be applicable to nondeterministic programs. Matrix multiplication programs are examples of programs that exhibit a large amount of inherent parallelism; it will be shown that a logic program for matrix multiplication can work in $O(n)$ time, instead of $O(n^3)$, which is the same speedup expected when matrix multiplication is performed under other parallel models of computation [27]. This potential speedup for deterministic programs is not possible in any other proposal for parallel execution of logic programs. Database queries are examples of nondeterministic processes when there is more than one answer to the query; there will be an example of parallel processing of a database query expressed in logic.

The dissertation is organized as follows: The next chapter is an extensive introduction to the formalism of logic programming and a discussion of the

Prolog programming language. The AND/OR Process Model is introduced in Chapter 3, showing how logic programs can be interpreted by sets of asynchronous and independent processes instead of by one large centralized search algorithm. Details for parallel execution based on the model are then given in Chapters 4 and 5. Chapter 6 is a discussion of issues pertaining to implementing the model on a physical network of processors. Finally, directions for future work and the relation of this work to other research in logic programming and parallel interpretation is the topic of the last chapter.

CHAPTER 2

Logic Programming

The phrase "logic programming" refers to the interpretation of well formed formulas of first order predicate logic as statements of a programming language. The first logic programming system was developed by Colmerauer and his colleagues at Marseille, after it was noticed that techniques used to build a resolution based theorem prover were similar to techniques used in the implementation of programming languages. Since then, the semantics of logic as a programming language have been formalized, and there have been a number of implementations of Prolog, a high level language that extends the formalism of logic programming in ways that make it more useful and efficient for solving practical problems.

It should be stressed at the outset that the parallel control method to be defined in the next chapters is *not* a model for parallel execution of Prolog programs; rather, it is a model for interpreting "pure" logic programs. Many of the extensions of logic programming that make Prolog a practical language are constructs that make sense only in von Neumann (stored program, single processor, single memory space) systems. According to the language first design philosophy outlined in the first chapter, it would be a mistake to force the implementation of these single-processor oriented constructs of Prolog in the parallel model. Alternatively, one should define mechanisms for parallel control of logic, and then implement the practical extensions to the formalism in terms of those mechanisms. As a concrete example, the formalism of logic programming does not provide for conditional expressions. Conditional expressions are defined in DEC-10 Prolog, but the definition relies heavily on the assumption that Prolog is being interpreted by a single processor. Conditional expressions will be defined for the AND/OR Process Model in Chapter 5, using the mechanisms of the parallel control instead of the constructs of Prolog.

This chapter is an extensive discussion of logic programming. It starts with the definition of the syntax, formal semantics, and standard (single processor) control. Then there is a detailed description of the Prolog language, with its extensions to the formalism, followed by a discussion of alternative (but still single processor) control strategies found in various Prolog systems. Finally, there is a section on potential sources of parallelism in logic programs.

The rather lengthy discussion of Prolog and logic programming is included here for three reasons. First, it sets the context for the definition of parallel control, by defining what are minimum requirements for a logic programming interpreter, as opposed to what are practical extensions. Second, some of the principles illustrated by the alternate control will be used in the definition of parallel control as well. Third, an interpreter based on the AND/OR Process model has been implemented, and it will be described in detail in Chapters 4 and 5. The interpreter is written in DEC-10 Prolog, and sections of it are listed in the Appendices for readers interested in the fine details of the AND/OR Process Model. Hopefully this introduction to Prolog will help readers unfamiliar with the language to understand the interpreter.

2.1. Syntax

A logic program is a set of formulas of first order predicate calculus. Most interpreters (including the parallel interpreter defined in this dissertation) accept only formulas written as *Horn clauses*, which are a subset of the formulas that can be written using the full syntax of predicate calculus. Although at first this appears restrictive, in fact any sentence of first order logic can be transformed into a set of Horn clauses [43]. A discussion of the merits and difficulties of writing expressions using only Horn clauses can be found in Kowalski's book [34].

A clause is defined to be a set of positive or negative ***literals***, each of the form

$$p(a_1, \cdots a_n)$$

or

$$\neg p(a_1, \cdots a_n)$$

The symbol p is a *predicate symbol*, and the a_i are the *arguments* of the literal. A clause in general can have any number of positive or negative literals, but a Horn clause contains at most one positive literal and zero or more negative literals.

The arguments of a literal are *terms*. A term is either a ***variable***, or it is an object composed of a *function symbol* and arguments. Nonvariable terms are also written in the prefix form, as in

$$f(x_1, \cdots x_m)$$

where f is the function symbol. The arguments x_i of a term are themselves terms. A term or literal with n arguments is said to be ***n-ary***, or of ***arity*** n. A 0-ary term is an *atom*, and is written without parentheses: $x()$ is simply x.

In order to disambiguate the names of variables and nonvariable terms, the names of variables will start with upper case letters. In the execution of the program, variables may be *instantiated* to other terms. When variable X is instantiated to term t, X is said to be *bound*, or to have a *value* of t.

Terms are the basic data structures of logic programs; all objects of the problem domain must be represented in the program as terms, just as objects must be represented as lists in LISP. Terms have all of the versatility of lists as far as representing objects. In general, a LISP list such as (FOO A B) can be written as the term

$$foo(a,b)$$

with the first item in the list used as the function symbol.

Syntactic sugaring is provided by allowing nonalphabetic symbols as function symbols, and letting these symbols be written as infix operators, so $+(X,Y)$ and $X+Y$ are legal terms.

Clauses in logic programs are most often written in the form of implications, with the single positive literal on the left and the remaining negative literals

forming a conjunction on the right.[1] Depending on whether there is a positive literal or not, and on whether there are any negative literals, there are four possible kinds of clauses in a logic program:

Implication

$p \leftarrow q \ \& \ r.$
One positive literal, one or more negative literals.

Unit Clause

$p \leftarrow .$
One positive literal, zero negative literals.

Goal Statement

$\leftarrow q \ \& \ r.$
Zero positive literals, one or more negative literals.

Null Clause

$\square$.
Zero positive literals, zero negative literals;
represents a contradiction.

The single (positive) literal to the left of the arrow is the ***head*** of the clause, and the (negative) literals to the right comprise the *body*.

Finally, here is the definition of some terminology often used to explain the execution of a logic program:

-- An n-ary *procedure* for p is defined to be the set of all clauses which have an n-ary literal with predicate symbol p as the head literal.

-- Quite often the execution of a logic program is explained in terms of problem solving. Clauses are referred to as *goals*, and literals in the body are referred to as *subgoals*.

-- A literal in the body of a clause is said to be a *call* to a procedure; if the call succeeds, the literal is *solved*.

-- A *ground term* is a term in which none of the arguments is a variable or contains variables.

---

[1] A clause is actually a disjunction of literals, *e.g.* the set $\{ \neg p(a,b), q(X), \neg r(X,f(a)) \}$ is actually $\neg p(a,b) + q(X) + \neg r(X,f(a))$. Since $P + (\neg Q + \neg R) \equiv P \leftarrow Q \ \& \ R$ the above clause can also be written as $q(X) \leftarrow p(a,b) \ \& \ r(X,f(a))$.

A simple logic program, which most examples in this chapter will reference, is given in Figure 1. This program has seven procedures, six of which are simply sets of unit clauses. The seventh, *paper*, is defined by two implications and one unit clause.

2.2. Semantics

The formal semantics of logic programming were originally defined by van Emden and Kowalski [22]. The denotation, or meaning, of an n-ary procedure p is **D**(p), a set of n-tuples of ground terms. This definition is similar to the definition of a relation, and in fact the denotation of a procedure is often called a relation. There are three ways of defining **D**; all three methods define the same set.

$\mathbf{D}^1$(p), the *operational* semantics of an n-ary procedure p, is defined to be the set of all n-tuples $< t_1 \cdots t_n >$ such that the predicate $p(t_1 \cdots t_n)$ is *provable*, given the clauses of the program as axioms. Implementations of logic programming systems use a constructive proof procedure to create the tuples of $\mathbf{D}^1$. For example, a goal statement such as

$$\leftarrow p(X,a).$$

is in fact a request that the system prove $p(X,a)$. A constructive proof not only satisfies the request, it generates a set of terms $\boldsymbol{X}$ such that $p(x_i,a)$ is provable for any x_i belonging to $\boldsymbol{X}$. In response to the above query, an interpreter would consruct the set

$$\{ < x_i, a > \mid x_i \in \mathbf{X} \}$$

which is the subset of $\mathbf{D}^1$(p) in which the atom a is the second term in the tuple.

In the *model-theoretic* semantics, the meaning of p is $\mathbf{D}^2$(p), the set of all n-tuples $< t_1 \cdots t_n >$ for which $p(t_1 \cdots t_n)$ is *true*. Since the first order predicate calculus is complete and consistent (*i.e.* any true statement can be proven, and any statement proven is in fact true), $\mathbf{D}^2 \equiv \mathbf{D}^1$.

The *fixed point* semantics $\mathbf{D}^3$(p) is derived from the program through a transformation that maps clauses into ground clauses, from which tuples of

```
author(fp,backus) ← .
author(df,arvind) ← .
author(eft,kling) ← .
author(pro,pereira) ← .
author(sem,vanemden) ← .
author(db,warren) ← .
author(sasl,turner) ← .
author(xform,standish) ← .

date(fp,1978) ← .
date(df,1978) ← .
date(eft,1978) ← .
date(pro,1978) ← .
date(sem,1976) ← .
date(db,1981) ← .
date(sasl,1979) ← .

title(db,efficient_processing_of_interactive...) ← .
title(df,an_asynchronous_programming_language...) ← .
title(eft,value_conflicts_and_social_choice...) ← .
title(fp,can_programming_be_liberated...) ← .
title(pro,dec-10_prolog_user_manual) ← .
title(sasl,a_new_implementation_technique...) ← .
title(sem,the_semantics_of_predicate_logic...) ← .
title(xform,irvine_program_transformation_catalog) ← .

loc(arvind,mit,1980) ← .
loc(backus,ibm,1978) ← .
loc(kling,uci,1978) ← .
loc(pereira,lisbon,1978) ← .
loc(vanemden,waterloo,1980) ← .
loc(turner,kent,1981) ← .
loc(warren,edinburgh,1977) ← .
loc(warren,sri,1982) ← .

journal(fp,cacm) ← .
journal(sasl,spe) ← .
journal(kling,cacm) ← .
journal(sem,jacm) ← .

tr(db,edinburgh) ← .
tr(df,uci) ← .

paper(P,D,I) ← date(P,D) & author(P,X) & loc(X,I,D).
paper(P,D,I) ← tr(P,I) & date(P,D).
paper(xform,1978,uci) ← .
```

Most binary literals $p(X,Y)$ in this program can be read as "the p of X is Y", *e.g.* *author*(*fp*,*backus*) means "the author of the FP paper is Backus" and *date*(*eft*,1978) means "the date of the EFT paper is 1978". But read $tr(x,y)$ as "x is a tech report from y". Implications $p \leftarrow q \ \& \ r$ are read "p if q and r". The first clause in the procedure for *paper* is "A paper P with date D was written at institution I if the date of P is D and the author of P is X and the location of X was I in year D."

Figure 1. A Logic Program

ground terms are formed. The method of using fixed points to define the semantics of recursive programs in general is defined in Manna's book [37], and the definition of the transformation for logic programs and a proof that $\mathbf{D}^1 \equiv \mathbf{D}^2 \equiv \mathbf{D}^3$ can be found in the article by van Emden and Kowalski [22].

The denotation of a procedure may be an infinite set of tuples. This is often the case when the procedure is intended to model a *function*. An *n*-ary function

$$f: \Lambda^n \rightarrow \Lambda^m$$

is represented in a logic program by an *(n+m)*-ary procedure, for which *n* arguments are used for inputs. The remaining *m* argument positions are uninstantiated variables when the procedure is called, and the net effect of executing the procedure is that the variables are bound to the output values of the function. As an example, consider the addition function of integer arithmetic. A procedure call of the form *sum*(*a*,*b*,*Z*) means to bind the variable *Z* to the sum of integers *a* and *b* (where the integers are represented by terms). The denotation of *sum* is an infinite set of 3-tuples:

<0,0,0>
<0,1,1>
<1,0,1>
<1,1,2>

Deterministic functions have exactly one tuple for each distinct combination of inputs, while *nondeterministic* functions may have more than one such tuple. When represented as relations in this way, all functions are invertible, *i.e.* the relation represents not only the function but also its inverse.

The operational semantics of most logic programming systems is defined by the *resolution* rule, a deductive inference method defined for formulas of first order predicate calculus written as clauses [52]. By restricting the syntax to Horn clauses, resolution proof can be made into a practical system. The remainder of this section is a discussion of resolution, and the next section takes up the subject of controlling the order of the inferences in an interpretation.

The resolution rule states that from two clauses

$$\{ p_1, \cdots p_{n-1}, q, p_{n+1} \cdots \}$$
$$\{ r_1, \cdots r_{m-1}, \neg q, r_{m+1} \cdots \}$$

which contain literals q and $\neg q$ with the same predicate symbol, same arity, and no variables in common, it is possible to derive a new clause

$$\{ p_1, \cdots p_{n-1}, p_{n+1}, \cdots r_1, \cdots r_{m-1}, r_{m+1} \cdots \}$$

that has all literals except q and $\neg q$ from the original two clauses. The derived clause is known as a ***resolvent.***

The literals q and $\neg q$ from the original two clauses must be ***unifiable***; that is, it must be possible to substitute terms for variables occurring in the literals so that they become identical except for the $\neg$ symbol. For example, $q(f(a))$ and $\neg q(X)$ can be unified, since when $f(a)$ is substituted for X in $q(X)$ the literals are both $q(f(a))$; $q(a)$ and $q(b)$ are not unifiable, since neither literal contains a variable and the arguments are not identical. In order for two literals (or terms) to be unifiable, they must have the same predicate (function) symbol and be of the same arity, and the corresponding arguments must be unifiable. Unifying substitutions are often identified by lower case Greek letters, such as θ. $C\theta$ denotes the clause C after substitution θ has been performed.

If a substitution for a variable is required in order to unify the literals, the substitution is applied to all occurrences of that variable in the resolvent. As an example, consider the two clauses

$$\{ p(X,1), q(Y,X), r(X,Z) \} \quad \{ \neg p(0,W) \}$$

The literals $p(X,a)$ and $\neg p(0,W)$ are unifiable by substituting 1 for W and 0 for X, so the resolvent is

$$\{ q(Y,0), r(0,Z) \}$$

It sometimes happens that one variable will be substituted for another during the unification, as in the following example:

Input clauses:

$$\{\ p(1,X),\ q(1,X)\ \} \quad \{\ \neg q(1,Y),\ r(2,Y)\ \}$$

Resolvent:

$$\{\ p(1,X'\ ),\ r(2,X'\ )\ \}$$

Note that X' is a new variable, and that the resolvent has no variables in common with either input clause.

Figure 2 shows many more examples of successful and unsuccessful resolutions.

The requirement that literals q and $\neg q$ have no variables in common is easily satisfied, since it is always possible to rename the variables in one or both of the clauses before doing the inference. This implies that the ***scope*** of a variable is restricted to the clause in which it occurs. This is an important point, and will be addressed again in Chapter 3 in the discussion of parallel evaluation.

A complete resolution proof of a clause C with respect to a set of axioms A (also expressed as clauses) is the derivation of the null clause from the set of clauses $\{\ \neg C \cup A\ \}$, *i.e.* negate C and then show that the negation leads to a contradiction. Unification and substitution are the operations that make resolution a constructive proof procedure. After the null clause has been derived, it is possible to construct terms for variables of C by using the substitutions performed during the proof (more specifically, use the composition of substitutions created in the sequence of resolutions [43]).

2.3. Control

The operational semantics of almost all logic programming systems is defined by the resolution rule. That is, the meaning of a program is determined by the tuples that can be generated by a resolution proof. What remains to be specified is *control*, the mechanism that determines the order in which the resolutions are performed. The simplest and most common control strategy is presented in this section.

A logic program is started by specifying an initial goal statement $\mathbf{G_0}$. A

Successful Resolutions

Clauses	Resolvent	Substitution
{ p(X), q(X) } { ¬p(a), r(Y) }	{ q(a), r(Y) }	X = a
{ p(X,a), q(X) } { ¬p(b,Y), r(Y) }	{ q(b), r(a) }	X = b, Y = a
{ p(X) } { ¬p(1) }	□	X = 1
{ p(X), q(X) } { ¬p(f(a,B)) }	{ q(f(a,B)) }	X = f(a,B)
{ p(X), q(X) } { ¬p(A), r(A) }	{ q(X), r(X) }	<note one variable substituted for another>
{ p(X), q(X) } { ¬p(A), r(X) }	{ q(X), r(X2) }	<note renaming of variables>

Unsuccessful Resolutions

Clauses	Reason for Failure
{ p(X), q(X) } { p(X), r(Y) }	must unify a positive literal with a negative literal
{ p(X), q(Y) } { ¬r(X), s(Y) }	no predicate symbols match
{ p(X,a), q(X) } { ¬p(b) }	predicate symbols match, but literals do not have same arity
{ p(a) } { ¬p(b) }	arguments not unifiable

Figure 2. Examples of Unification and Resolution

computation is a sequence of goal statements $\mathbf{G}_0, \mathbf{G}_1, \cdots \mathbf{G}_n$, where $\mathbf{G}_i$, $1 \leq i \leq n$, is the resolvent of $\mathbf{G}_{i-1}$ and one of the clauses of the program. The computation halts with success when the null clause is derived, and halts with failure if no inference is possible.

Recall that when clauses are written as implications, literals in the body of the clause are negative literals, the heads of clauses are positive literals, and a goal statement is a clause with only negative literals. In the context of a logic programming system based on resolution, then, the derivation of a new goal statement $\mathbf{G}_{i+1}$ from $\mathbf{G}_i$ involves the following steps:

-- Select any literal L from $\mathbf{G}_i$ (usually L is the leftmost literal in $\mathbf{G}_i$).

-- Find a clause C in the program such that the head of C can be unified with L.

-- Rename the variables in C (since the two clauses being resolved, C and $\mathbf{G}_i$, cannot have variables in common).

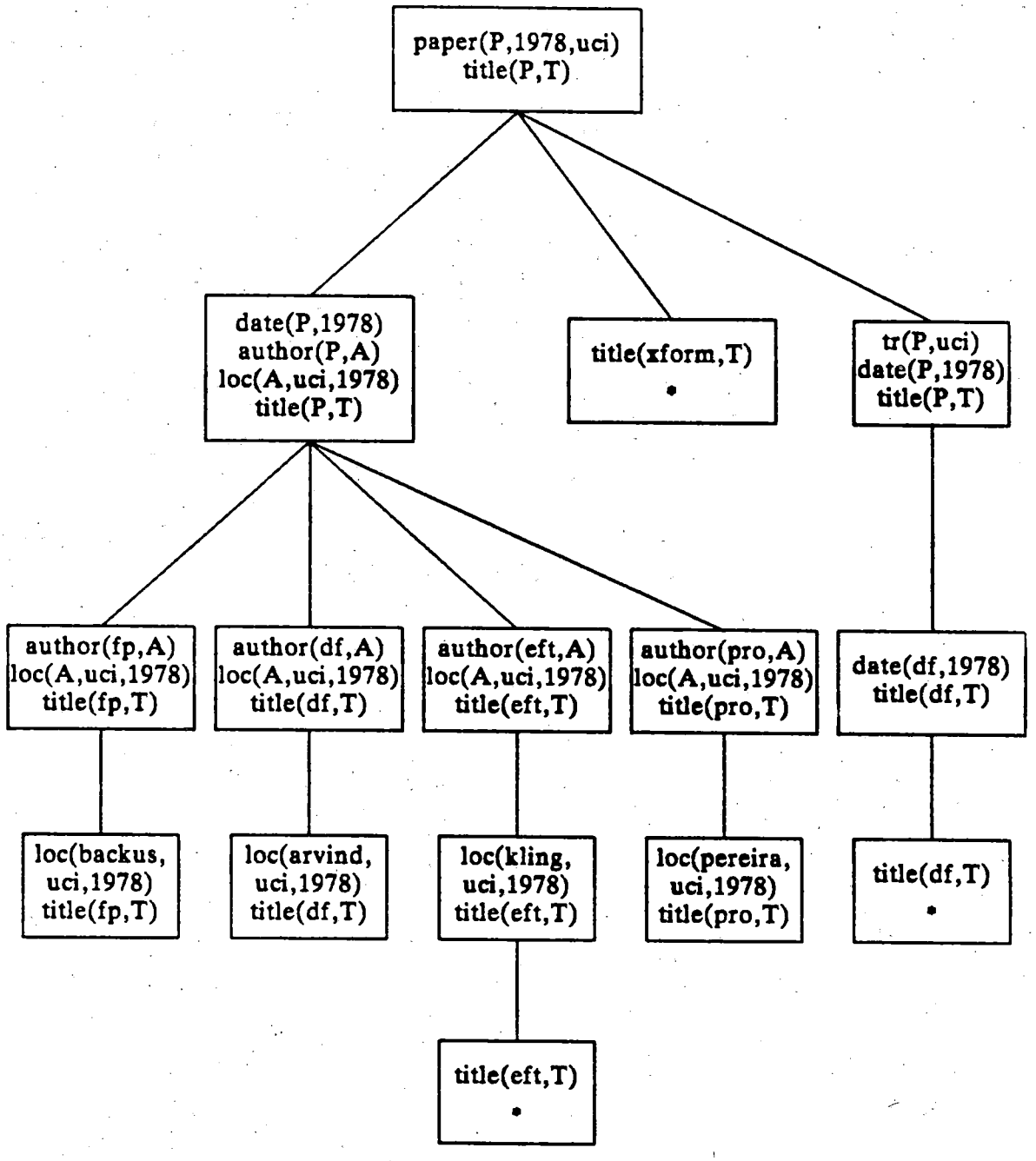
-- Form the new goal statement $\mathbf{G}_{i+1}$ by removing L from $\mathbf{G}_i$ and replacing it by the literals from the body of C.

-- If the unification requires a substitution, apply the substitution to $\mathbf{G}_{i+1}$.

When the selected clause C is a unit clause (when the body does not have any literals) then $\mathbf{G}_{i+1}$ will have one less literal than $\mathbf{G}_i$. $\mathbf{G}_{i+1}$ will be the null clause when $\mathbf{G}_i$ contains exactly one literal, and this literal is resolved with a unit clause.

A *goal tree* is a tree where each node is a clause. Immediate descendants of a node N are clauses that can be derived from N by one inference. Steps in the execution of a logic program can be represented as a goal tree: the root of the tree is the initial goal $\mathbf{G}_0$, interior nodes are the intermediate goal statements, and the leaves of the tree are either empty clauses or are clauses for which no inference is possible. The latter are called ***fail nodes***.

The standard control strategy is a depth first search of a goal tree (see, for example, van Emden [23]). When the interpreter reaches a fail node, it ***back-***



Portion of a goal tree generated during solution of *paper(X,1978,uci)*. Only the leftmost literal within a node was used to derive descendant nodes. Nodes with an asterisk represent successful branches, i.e. the next clause generated on one of these branches is the null clause.

Figure 3. A Goal Tree

tracks to the most recent choice point. Note that if the goal tree contains an infinite branch, then the standard control will not find all occurrences of the null clause: it will miss those that are to the right of the infinite branch, and in fact will go into an infinite loop when it encounters this branch.

Figure 3 shows a goal tree formed from the initial goal

$\leftarrow paper(P,1978,uci)$.

and the program of Figure 1. This tree was formed by using only the leftmost literal from any node as the literal to resolve. This selection rule (use only the leftmost literal from the current goal statement) is used by the standard control method, so the tree in the figure is also the tree that is searched by the standard control.

It is possible to describe the depth first search in terms of the textual representation of the program, without resorting to descriptions of goal trees. In textual terms, the system solves the subgoals of a goal statement one at a time, from left to right. To solve a single goal G, the system searches the program from top to bottom for a clause with a head that can be unified with G. If it finds such a clause, then it recursively solves the goals in the body, from left to right. The same problem used to grow the goal tree of Figure 3 is given again in Figure 4, this time accompanied by an explanation in terms of the text of the example program.

2.4. Prolog

The particular version of Prolog described here is DEC-10 Prolog [46]; there have been numerous other implementations, including some for microcomputers. The syntactic differences between logic programs and DEC-10 Prolog programs are minor:

-- The implication symbol is :- instead of $\leftarrow$.

-- The :- symbol is not used in unit clauses, *i.e.*

$p \leftarrow$.

is written

p.

-- The literals in the body of a clause are separated by commas, not the logic symbol &.

-- Integers are a special type of atomic term.

-- LISP-like lists are allowed as data structures. Lists are enclosed in square brackets. The empty list is [], and the list $[A|B]$ is a list with A as first element (CAR in LISP terminology) and B as the tail (CDR), which is also a list. Lists are really terms: $[a,b,c]$ is simply shorthand for the term $.(a,.(b,.(c,[])))$, which has a period for the function symbol (CONS).

2.4.1. Evaluable Predicates

Previously, the meaning of the ternary predicate *sum* was defined to be the relation containing the infinite set of 3-tuples $<a,b,c>$, where c is the sum of a and b. Within the formalism of logic programming, there are two methods for doing arithmetic. One is that the relation can be given explicitly, as a set of assertions $sum(a,b,c)$, and arithmetic operations will be essentially table searches. Obviously, the entire infinite relation cannot be realized on any physical machine, and the subset that is defined consumes a large amount of space. Alternatively, the relation can be computed, by defining addition and other arithmetic operations axiomatically. In using this method, the symbol 0 is used to represent the integer zero, the terms $s(0)$, $s(s(0))$, *etc.* represent the positive integers, and addition is performed by proving that the sum exists, using the axioms of arithmetic [33].

In Prolog, arithmetic is performed by metalogical ***evaluable predicates***, which are analogous to the built-in (predefined) primitive functions of applicative languages. The underlying computer system performs the arithmetic operations, simply because it is faster. This will have an effect on the semantics of programs, namely some goals that are solvable using one of the formal methods may not be solvable when the machine performs the operation. An example is

```
author(fp,backus) ← .
author(df,arvind) ← .
author(eft,kling) ← .
author(pro,pereira) ← .
author(sem,vanemden) ← .
author(db,warren) ← .
author(sasl,turner) ← .

date(fp,1978) ← .
date(df,1978) ← .
date(eft,1978) ← .
date(pro,1978) ← .
date(sem,1976) ← .
date(db,1981) ← .
date(sasl,1979) ← .

loc(arvind,mit,1980) ← .
loc(backus,ibm,1978) ← .
loc(kling,uci,1978) ← .
loc(turner,kent,1981) ← .

paper(P,D,I) ← date(P,D) & author(P,X) & loc(X,I,D).
paper(P,D,I) ← tr(P,I) & date(P,D) ← .
paper(xform,1978,uci) ← .
```

The above program is part of the logic program of Figure 1. This figure explains the steps in the solution of

$$\leftarrow paper(P,1978,uci)\ \&\ title(P,T).$$

First, the interpreter tries to solve *paper*(*P*,1978,*uci*). When that literal is unified with the head of the first clause for *paper*, the body (after applying the substitution) is

$$\leftarrow date(P,1978)\ \&\ author(P,X)\ \&\ loc(X,uci,1978).$$

Now the interpreter must solve these goals, left to right, in order to complete the solution of *paper*(*P*,1978,*uci*). The first solution for *date*(*P*,1978) unifies *P* with *fp*, so the interpreter moves on the next goal, which is now *author*(*fp*,*X*). The only solution for this binds *X* to *backus*, and the remaining literal in the body of *paper* is now *loc*(*backus*,*uci*,1978). This fails, and the interpreter backtracks. The most recently solved goal was *author*(*fp*,*X*), so the interpreter tries to re-solve it. That also fails, so the interpreter moves back even further, to re-solve *date*(*P*,1978). This can be solved a different way, by unifying *X* with *pro*, so the interpreter moves forward again, now trying to solve *author*(*pro*,*X*). This backtracking and retrying continues until *date*(*P*,1978) is finally solved by unifying *P* with *eft*; then *author*(*eft*,*X*) is solved by binding *X* to *kling*, so finally *loc* is solvable with the arguments *loc*(*kling*,*uci*,1978). That completes the solution of *paper*, and the system moves on to solve the remaining literal from the original goal, which is now *title*(*eft*,*T*) since *P* was bound to *eft* in the solution of *paper*. This will succeed immediately when *T* is bound to *value_conflicts*... (Figure 1).

Figure 4. Example Computation

$\leftarrow sum(X, Y, 5)$.

In other words, what integers X and Y have a sum of 5? This goal has many solutions, all of which can be found if the system searches the relation for tuples that have 5 as the third element. However, the goal fails when a machine is asked to add two uninstantiated variables.

Arithmetic in DEC-10 Prolog is performed by the evaluable predicate *is*. This is a binary predicate, and is one of the predicates that can be used when writing infix expressions. The second argument must be a legal arithmetic expression, constructed from the usual operators and integer terms, and the first argument can be either an integer or a variable. When *is* is called, the expression is evaluated. If the first argument is a variable, it will be unified with the value of the expression. If the first argument is an integer, *is* will succeed only if the integer and the value of the expression are the same. If the second argument contains any variables (*i.e.* it is not a ground term) then the goal fails.

Some examples of goals that use *is*:

$is(X, +(2,1))$
X *is* 2+1

These literals are identical; one is written in the infix style. When called, 2+1 is evaluated to 3, and then X is bound to the atom 3.

3 *is* 2+1

The expression is evaluated, and since 2+1 = 3, the goal succeeds.

Z *is* (2*3)+(4*5)

Z is bound to *26*.

5 *is* 2+1

This will fail.

5 *is* X+3

This also fails, since X must be bound. Note that this does not mean variables cannot be used in the expression. It simply means that the variable must be bound to an integer term as the result of solving some other literal by the time this literal is selected for evaluation.

The idea that a goal will fail if certain argument positions contain uninstantiated variables is expressed in DEC-10 Prolog by *I/O modes*. Each argument in a goal has one of three modes associated with it: in input-only mode, the argument must be a ground term; in output-only mode the argument must be an uninstantiated variable, which will be bound in the solution of the goal; and in the default, don't-care, mode, arguments can be either ground or uninstantiated. When a Prolog procedure implements an *n*-ary function, n argument positions of the predicate symbol will be input-only mode, and the remaining argument positions are output-only. The DEC-10 Prolog compiler uses mode declarations supplied by the user program to generate more efficient code for user-defined functions. The concept of I/O modes will also be used to order goals for parallel solution, as defined in Section 5.1.

In some Prolog systems, addition is performed by a ternary evaluable predicate named *sum*. *sum*(X,Y,Z) will be true if X+Y=Z, or if there is a value can be found for one of X, Y, or Z that will make the expression X+Y=Z true. In these systems, the last example is written *sum*(X,3,5), and this goal succeeds by binding X to 2. *sum* is said to have a *threshold* of two, meaning that all that is necessary for the success of *sum* is that two of the three arguments be bound to integer values [11, 31, 65]. *sum* will fail if fewer than two arguments are bound.

2.4.2. Higher Order Functions

DEC-10 Prolog has constructs that allow programmers to treat clauses as data, allowing procedures to be passed as arguments to other procedures or added to the program as it is executing. A more extensive discussion of the necessity and/or desirability of these extensions can be found in a paper by Warren [64].

Clauses can be represented by terms if the implication symbol and the logical AND symbol (:- and comma in DEC-10 Prolog) are in the set of infix

operators. Thus the clauses

$p :- q.$
$p(a) :- q, r(a).$

can be represented by the terms

$:-(p,q)$
$:-(p(a), ,(q,r(a)))$
$\uparrow$

where the comma above the arrow is a function symbol, not a separator of arguments.

The goal

$:- assert(T).$

adds the term T (which must be bound to a term that has the syntax of a clause) to the program currently in the system. The opposite of *assert* is *retract*:

$:- retract(T).$

finds a clause that unifies with T and then deletes it from the program.

A second method for handling program pieces as data is through the evaluable predicate *call*, which is similar to the EVAL function of LISP:

$:- call(P).$

treats the term P as if it were a goal statement, and calls the interpreter recursively to solve that goal statement.

An evaluable predicate that is useful in conjunction with *call* is $=..$, which constructs terms from a list of components:

$:- T =.. [F|A].$

is a goal that succeeds if T is a term that has function symbol F and argument list A. For example, the goal

$:- T =.. [author,db,warren].$

unifies T with the term *author(db,warren)*.

A procedure that uses both of these constructs is

```
mapcar(F,[],[]).
mapcar(F,[X1|Xn],[Y1|Yn]) :-
        Goal =.. [F,X1,Y1],
        call(Goal),
        mapcar(F,Xn,Yn.)
```

This is the Prolog equivalent of the LISP function MAPCAR, which takes as arguments a function and a list of elements X, and returns a list Y consisting of the results of applying the function to each X. If ***sqr_one***(X,Y) binds Y to the square of the integer X, then the goal

```
:- mapcar(sqr_one,[1,2,3,4],L)
```

unifies L with the list [1,4,9,16].

2.4.3. The Cut Symbol

The standard Prolog control has been described as a depth first search of a goal tree. The *cut symbol*, !, allows the programmer to control the search by pruning unwanted branches from the search tree. Cut is inserted into the body of a clause, along with literals. When the interpreter encounters cut as a goal, it always succeeds. However, if the interpreter ever backtracks to the point where it has to re-solve the cut, the resulting behavior is that the *head* of the clause that contains the cut fails. The net effect is that all further solutions for literals to the left of the cut in the clause, and all clauses in the same procedure following the current clause, are deleted from the goal tree.

As an example, refer to the small program in Figure 5, and observe what happens when a call is made to p. The first clause for p has a body with literals q and r, so the system starts to solve q. The first clause for q has body

```
a, !, b.
```

a and the cut are solved, but b fails. Now the interpreter backtracks, and encounters ! while backtracking. The *head* of the clause that contains this cut is q, so the call to q fails. The call to q was made from the first clause for p, and the

failure of q forces the interpreter to move on to the second clause for p, where the program finally succeeds.

Common uses of the cut symbol are in finalizing choices from nondeterministic procedures and in the definitions of conditional expressions and negation.

Consider this small program, which has two different definitions for a procedure p:

$q(a)$.
$q(b)$.
$p1(X)$:- $q(X)$.
$p2(X)$:- $q(X)$, !.

When solving the goal

:- $p1(X)$, *continue*.

the system eventually selects the first clause for q to solve $q(X)$, and X is bound to a. If *continue* fails, the system backtracks into $p1(X)$, then $q(X)$ is re-solved, and X will be bound to b. However, when there is a cut symbol in the body, as in the definition of $p2$, the second answer is not produced. In solving the goal statement

:- $p2(X)$, *continue*.

the first answer, $X = a$, is produced as before. When the system backtracks into $p2(X)$ after *continue* fails, it encounters the cut symbol, so it does not retry $q(X)$; furthermore $p2(X)$ also fails. Without the cut symbol, p is nondeterministic procedure, since there is more than one solution to a call to $p(X)$. When a cut symbol is added, this procedure behaves like a deterministic procedure, in that it produces one answer and then fails when asked to produce more answers.

A conditional expression in a functional language has the general form of

$f(X) =$ if $p(X)$ *then* $g(X)$ *else* $h(X)$

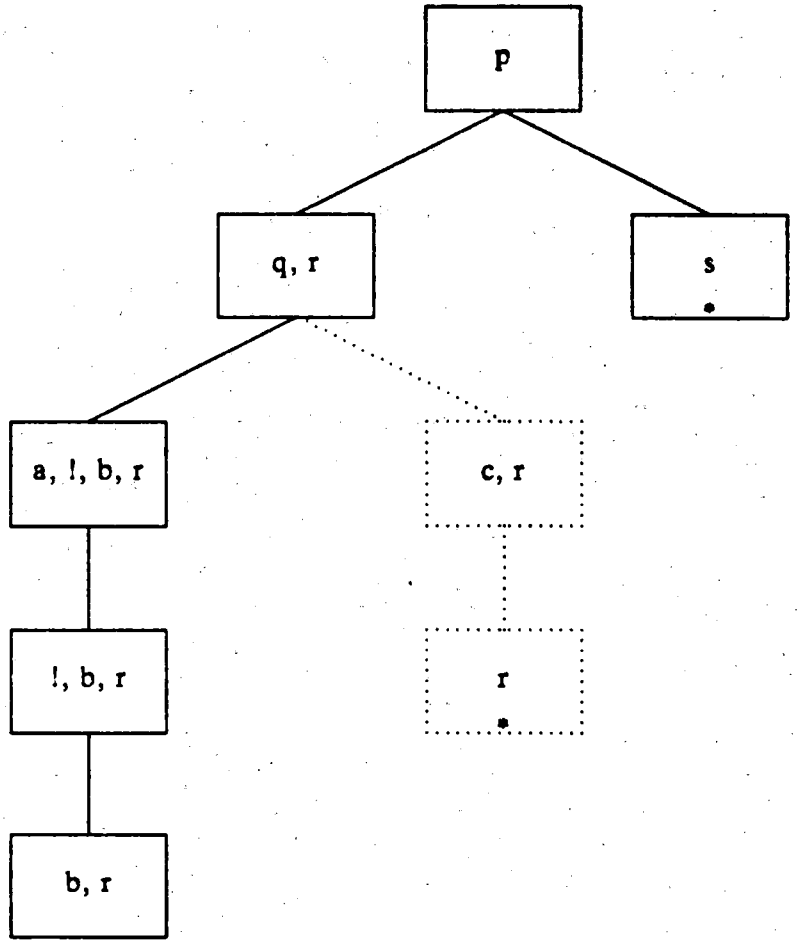
If $p(X)$ is true, the value of $f(X)$ is given by $g(X)$, otherwise it is defined by $h(X)$. Recall that in Prolog, n-ary functions can be defined by $(n+1)$-ary predicates, in a way that the result of applying the function is unified with a variable passed in

Program:

```
p :- q, r.
p :- s.
q :- a, !, b.
q :- c.
a.
c.
r.
s.
```

Initial Goal:

```
:- p.
```



This figure shows the effect of the cut symbol (*!*) on a depth first search. When the leftmost branch ends in a fail node (no solution for *b*), the interpreter backs up to where *!* is the first goal in the list. The result is that the head of the clause containing the *!* fails. In this example, the *!* is part of a clause for *q*, so the call to *q* fails. The only solution found by the interpreter is the one on the far right.

Figure 5. Pruned Goal Tree

the extra argument position. In Prolog, the above expression is written as two clauses:

$f(X, Y)$:- $p(X)$, !, $g(X, Y)$.
$f(X, Y)$:- $h(X, Y)$.

When the function is called, for example by the goal

:- $f(10, Y)$.

the interpreter selects the first clause for f, and then calls $p(10)$. If the call $p(10)$ succeeds, Y is bound by the call $g(10, Y)$. If $p(10)$ fails, the interpreter backtracks to the second clause for f, and the value of Y is computed by the call $h(10, Y)$.

The cut symbol is necessary for those occasions when $g(X, Y)$ fails. The desired control behavior is that when $p(X)$ is true, then $f(X, Y)$ is defined by $g(X, Y)$; this means that if $g(X, Y)$ fails, then $f(X, Y)$ should also fail. This situation is analogous to the definition of conditional expressions in FP, where f is undefined when p is true but g is undefined. An interpreter must not backtrack to try $h(X, Y)$ in those cases where $p(X)$ succeeds but $g(X, Y)$ fails. This behavior is enforced by the cut symbol.

The third common use of the cut is in the definition of a procedure for negation:

not(G) :- *call*(G), !, *fail*.
not(G).

where *fail* is a goal that cannot be solved. This is ***negation as failure***, first defined by Clark [10]. Recall that an implication

p :- q, r

is equivalent to the Horn clause

$\{ p, \neg q, \neg r \}$

and that the literals of the body of a clause are actually negative literals. Thus one cannot simply write

$:- \neg p(X).$

for the request "prove p(X) is false", since a negated literal in the body of a clause would actually be a positive literal, and by definition a goal statement (or the body of a clause) must contain only negative literals.

Negation as failure means that if one fails to prove a statement, then one can assume the statement is false. Operationally, an implementation of negation as failure requires that if a goal G is solvable, then $not(G)$ should fail, but if G fails, $not(G)$ should succeed.

Referring to the above Prolog definition of negation, when $not(G)$ is called, with some goal G as the argument, the interpreter first tries to solve G, via the statement call(G). If this fails, then the second clause for $not(G)$ is tried, and since this is a unit clause, it succeeds. In other words, when G fails, $not(G)$ succeeds.

In the other case, when G succeeds, the interpreter moves on to the cut and *fail* literals. Cut succeeds, and *fail* fails. Because the cut is there, the interpreter does not try to solve G again, and in addition causes the failure of $not(G)$, which is the head of the clause with the cut. Thus when G succeeds, $not(G)$ fails.

Negation as failure is definitely a metalogical construct. In formal logic, there is a major difference between the statements "P is false" and "P cannot be proven", especially when higher order functions are introduced into the system. There are also practical problems in Prolog systems that use this definition of negation. These problems arise when the negated goal contains variables. For example, suppose the predicate $p(t)$ can be proven, but that $p(f)$ cannot. The goals $not(p(t))$ and $not(p(f))$ behave as expected, with $not(p(t))$ failing and $not(p(f))$ succeeding. But notice what happens with the goal $not(p(X))$: the system tries to solve $p(X)$, which it accomplishes by binding X to t, so $not(p(X))$ is considered to be false. But it was just shown that $not(p(X))$ is true when X is the term f, so one can argue that the system should actually succeed in solving $not(p(X))$ by binding X to f. Thus it is not clear at all whether $not(p(X))$ should succeed or fail, and if it should succeed, how to construct the set of legal values

for X.

In spite of these shortcomings, negation as failure is used effectively in many logic programs. A definition of negation as failure that does not rely on the cut symbol will be given in Chapter 4.

2.5. Alternate Control Strategies

Control in a logic program has been characterized in the previous sections as a tree search. The object of the search is to find a null clause at the end of a sequence of resolutions. The unifications used on the path from the root of the tree (which is the starting goal statement) to the null clause define an n-tuple of values for the n variables of the starting goal statement. If there is more than one way of solving the original goal, there will be a number of null clauses at the leaves of the tree, with an n-tuple defined by each path.

Every step in the expansion of the tree involves two choices: selection of a literal in the current node, from which successor nodes will be generated, followed by a selection of one of the successor nodes to be the root of the next tree searched. A standard interpreter performs a depth first search of a restricted space, in which the leftmost literal in a node is the only one ever selected for expansion, and the subtrees generated by expanding this literal are searched left to right.

The meaning of a procedure is a relation, which is an unordered set of tuples of terms. Ideally, a control strategy helps an interpreter construct every tuple in the relation if necessary. In practice, however, a given control method may not be able to order the required resolutions so that all tuples are constructed. In particular, a depth first interpreter never terminates when there is an infinite branch in the search tree; this control method will never construct any tuples defined by finite branches to the right of an infinite branch.

The meaning of a procedure is independent of the control mechanism. The meaning is a relation, an unordered set of tuples, and a control mechanism merely defines an order in which those tuples are created. Some alternatives to the standard depth first search optimize the search by decreasing the size of the

search space. Other alternative control methods generate a larger set of answers by avoiding infinite branches. Some of these alternatives will be discussed in this section. Although the mechanisms explained here are defined in terms of a sequential search of a single search space, some of the principles illustrated can and will be used in the definition of parallel control in Chapter 5.

The first three alternatives have to do with selecting other literals besides the leftmost as the literal to expand. The fourth alternative is intelligent backtracking, a more effective method for backtracking that prunes portions of the search tree that cannot contain solutions.

2.5.1. Selection Based on a Known Number of Solutions

In general, a tree search is more efficient when the branching factor in the tree is smaller. If a search algorithm can expand nodes that generate fewer descendants, then it might save itself needless work by traversing fewer unsuccessful branches. This principle is realized in logic programs in two ways.

The first is based on an interaction between goals determined by variable bindings. As an illustration, consider a program with two procedures with heads $p(X)$ and $q(X)$, both made up of only ground unit clauses. There are N_P clauses for p and N_Q clauses for q. Assume there are N_{PQ} terms that occur as arguments in both p and q; these are the terms that are constructed in response to the goal

$$\leftarrow p(X) \ \& \ q(X).$$

A solution of either literal by itself will bind X to a ground term. If the interpreter selects $p(X)$ at this point, there will be N_P descendant search trees, each with a root of the form $q(a)$, and the remaining steps consist of a search through the descendants looking for one of the N_{PQ} occurrences of the null clause. On the other hand, if the interpreter selects $q(X)$, there will be N_Q descendants with roots $p(a)$, but still exactly N_{PQ} null clauses. Whether the interpreter must generate all answers (*i.e.* find all null clauses) or just one (*i.e.* find the leftmost null clause), the efficiency of the search is determined by the proportion of the number of null clauses to the number of branches, and this proportion

is better when fewer branches are generated. Thus when it is known in advance that N_P is less than N_Q, always select $p(X)$ for resolution first, regardless of whether $p(X)$ is the leftmost literal in the goal statement.

Again, it is important to note that the order of selection effects the efficiency of the search, and possibly the order in which the answers are reported, but not the final result. The paths that end with a null clause in either tree lead to the construction of the same set of values for X.

A second case where generating the fewest number of descendants pays off is when all the descendant paths lead to fail nodes. Continuing the example, this occurs when N_{PQ} is zero. When $p(X)$ is selected, the interpreter searches N_P branches before backtracking to the parent. If $q(X)$ is selected, N_Q branches must be searched before the interpreter discovers that there are no null clauses in the tree. Again, if N_P is less than N_Q, less useless work is done if $p(X)$ is selected as the literal on which to base the resolutions.

This general strategy, of selecting literals that are known beforehand to have the fewest number of solutions, is used to optimize queries in the CHAT-80 relational database system [62, 63].

2.5.2. Selection by Number of Uninstantiated Variables

It may be possible to limit the size of the search space even when the interpreter does not have prior information about the number of solutions for each literal, by assuming that literals with fewer uninstantiated variables will generate fewer branches.

Consider a database of ground unit clauses of the form $p(X,Y)$, where there are N different terms for X and M different terms for Y, for a total of N×M unit clauses. Figure 6 gives the branching factor of a goal statement in which $p(X,Y)$ is selected for expansion. The branching factor is a function of the pattern of variable instantiation in $p(X,Y)$. The table shows that the branching factor is reduced when the number of uninstantiated variables is reduced.

Goal Pattern	Number of Uninstantiated Variables	Branching Factor
p(X,Y)	0	N × M
p(a,Y)	1	M
p(X,b)	1	N
p(a,b)	2	1

Figure 6. Branching Factor as a Function of Variable Instantiation

An example of where this observation improves the efficiency of a logic program can be seen in a clause based on the program of Figure 1:

query(*P*,*I*) ← *author*(*P*,*X*) & *loc*(*X*,*I*,*D*).

Given a depth first interpreter and an initial goal statement

← *query*(*eft*,*I*).

the derived goal statement will be

← *author*(*eft*,*X*) & *loc*(*X*,*I*,*D*).

There is just one way to solve the first subgoal, and that solution binds X to a term that leads to only one solution for the second subgoal; the final answer has X bound to *kling*, I to *uci*, and D to *1978*.

If, on the other hand, the initial goal statement is

← *query*(*P*,*uci*).

then the derived goal statement is

← *author*(*P*,*X*) & *loc*(*X*,*uci*,*D*).

The only answer to this query is the same as the one produced by the first example. There are eight ways to solve the first subgoal, since any of the unit clauses in the procedure for *author* are unifiable when no arguments are bound in the call, but only one of those unifications leads to a solution for the second subgoal. If an interpreter could first solve the ***rightmost*** subgoal (*loc*(*X*,*uci*,*D*) in this derived clause), it would find just one solution for that goal, and that solution leads immediately to a solution for the leftmost literal. In other words, the number of misleading branches can be reduced from seven to zero by selecting a literal that has one instantiated variable instead of zero instantiated variables.

This strategy of first solving goals that have the fewest number of uninstantiated variables was first mentioned by Kowalski [33]. The IC-Prolog interpreter implements the strategy by allowing users to write a number of versions of the same clause, and then annotate these clauses so that the interpreter selects the most efficient one at runtime, depending on the pattern of variable instantiation

in a goal [11]. The strategy will also be used by the parallel interpreter described in Chapter 5.

2.5.3. Coroutines

Consider these definitions of the procedures *concat* and *sqr*:

concat([],*List*,*List*).
concat([*Car*|*Cdr*],*L*1,[*Car*|*L*2]) ← *concat*(*Cdr*,*L*1,*L*2).

sqr([],[]).
sqr([*X*1|*Xn*],[*Y*1|*Yn*]) ← *Y*1 *is* *X*1 * *Y*1 & *sqr*(*Xn*,*Yn*).

The goal

← *concat*([1,2],[3,4],*L*) & *sqr*(*L*,*S*).

is a request to construct a list L that is the concatenation of the lists [1,2] and [3,4], and a list S such that every element of S is the square of the corresponding element of L. The only solution in the goal tree with this goal at the root gives the answers $L = [1,2,3,4]$ and $S = [1,4,9,16]$ (Figure 7). After the first step in the computation, the derived goal statement is

← *concat*([2],[3,4],*L'*) & *sqr*([1|*L'*],*S*).

where the variable L from the original goal has been bound to the term $[1|L']$. Bindings such as these, where a variable is bound to a term that contains other variables, are known as *partial bindings*.

The normal depth first control completely solves the call to *concat*, binding L to [1,2,3,4], before the solution of *sqr* is started. A *coroutine* control interleaves the steps in the solutions, by having *concat* make a "piece" of the list L through a partial binding, and then having *sqr* use this piece. The literal *concat*([1,2],[3,4],*L*) is called the *producer* of L, *sqr*(*L*,*S*) is a *consumer* of L, and the variable L is called the *communication channel* between the two literals.

The series of derivations made for the above example by a coroutine control is shown in Figure 8. Successive goal statements are derived until a partial binding is created for L. At that point, the consumer literal is selected, and derivations continue until a call to the consumer has an uninstantiated variable in the

concat([1,2],[3,4],L) & sqr(L,S).	
concat([2],[3,4],L') & sqr([1|L'],S).	L = [1|L']
concat([],[3,4],L'') & sqr([1,2|L''],S).	L = [1,2|L'']
sqr([1,2,3,4],S).	L = [1,2,3,4]
X1 is 1*1 & sqr([2,3,4],S').	S = [X1|S']
sqr([2,3,4],S')	S = [1|S']
X2 is 2*2 & sqr([3,4],S'').	S = [1,X2|S'']
sqr([3,4],S'').	S = [1,4|S'']
X3 is 3*3 & sqr([4],S''').	S = [1,4,X3|S''']
sqr([4],S''').	S = [1,4,9|S''']
X4 is 4*4 & sqr([],S'''').	S = [1,4,9,X4|S'''']
sqr([],S'''').	S = [1,4,9,16|S'''']
□	S = [1,4,9,16]

Figure 7. Sequence of Derivations in Depth-First Control

concat([1,2],[3,4],L) & sqr(L,S).	L = [1|L']
concat([2],[3,4],L') & *sqr([1|L'],S).*	
concat([2],[3,4],L') & *X1 is 1*1* & sqr(L',S').	
concat([2],[3,4],L') & sqr(L',S').	L' = [2|L'']
concat([],[3,4],L'') & *sqr([2|L''],S'').*	
concat([],[3,4],L'') & *X2 is 2*2* & sqr(L'',S'').	
concat([],[3,4],L'') & sqr(L'',S'').	L'' = [3,4]
sqr([3,4],S'').	
*X3 is 3*3* & sqr([4],S''').	S = [1,3,9|S''']
sqr([4],S''').	
*X4 is 4*4* & sqr([],S'''').	S = [1,3,9,16|S'''']
sqr([],S'''').	S'''' = []
□	S = [1,4,9,16]

The literal selected for use in the next resolution step is in italics. Derivations are made based on the consumer literal ***concat*** until the communication channel (variable *L*) is partially bound; then derivations based on the consumer literal ***sqr*** are made until the first argument is a variable.

Figure 8. Sequence of Derivations in Coroutine Control

argument position for L. Then the producer is selected, and another piece of L is created, and so on. To summarize, a depth first interpreter creates the entire list L, and then calls *sqr* to square every element in L, making S. The coroutine interpreter interleaves the interpretation of the two calls, creating and squaring the first element, then creating and squaring the second element, until the last element has been squared.

A most interesting use of coroutines is in the definition of infinite data structures. The clause

$$inf(N,[N|L]) \leftarrow M \text{ is } N+1 \ \& \ inf(M,L).$$

describes an infinite list of integers. The goal

$$\leftarrow inf(1,X) \ \& \ use(X,Y).$$

is a request to unify X with the infinite list of integers starting with 1, and then "use" this list; it results in an infinite loop when an attempt is made to solve it with a depth first interpreter, since that interpreter tries to create the entire list of integers starting from 1. A coroutine interpreter would create the sequence of integers only up to the last integer required by the goal $use(X,Y)$.

IC-Prolog [11, 12] allows the user to designate literals within a clause as producers and consumers, and is an implementation of coroutines. Infinite data structures are used in many elegant programs written in SASL [57], LUCID [2] and other applicative programming languages.

2.5.4. Intelligent Backtracking

Consider the set of unit clauses

$p(a) \leftarrow .$
$p(b) \leftarrow .$
$q(1) \leftarrow .$
$q(2) \leftarrow .$
$r(b,1) \leftarrow .$
$r(b,2) \leftarrow .$

Given the goal statement

$\leftarrow p(X)$ & $q(Y)$ & $r(X,Y)$.

a depth first interpreter first solves $p(X)$, binding X to a, then solves $q(Y)$, binding Y to 1, and then tries to solve $r(a,1)$. When the latter fails, the interpreter backtracks. The most recent choice point is in the selection of the clause for solving $q(Y)$; when this is redone, another solution is found, binding Y to 2, and the next goal is $r(a,2)$, which also fails.

Both of these calls to r fail because the solution of $p(X)$ binds X to a value that cannot be used to solve $r(X,Y)$. When the interpreter backs up only as far as $q(Y)$, it cannot fix this erroneous choice, and by re-solving $q(Y)$ it is wasting resources.

An interpreter that uses *intelligent backtracking* analyzes the cause of a failure, and backtracks to the source of values that cause the failure. An interpreter designed and implemented by Pereira and Porto performs this kind of analysis [47, 48]. In the example given above, it finds that any goal of the form $r(a,X)$ fails because of the presence of the term a in the first argument position. Since X was bound to a in the call to $p(X)$, the interpreter backs up past the call to $q(X)$, all the way to a choice point in the solution of $p(X)$. When $p(X)$ is solved again, binding X to b this time, the entire goal list can be solved, without the wasteful attempt to solve $r(a,2)$.

Other cases where intelligent backtracking can be helpful are in goals such as

$\leftarrow p(A)$ & $q(B)$ & $r(A)$.

When $r(A)$ fails, $q(B)$ can be skipped on backtracking since it does not produce any values that can effect the solution of $r(A)$. This is a case where it is not necessary to analyze the exact cause of the failure; it is only necessary to notice that a new solution of $q(B)$ cannot help solve $r(A)$, since $r(A)$ and $q(B)$ have no variables in common. Behavior similar to this limited form of intelligent backtracking will be seen in the parallel control described in Chapter 5.

2.6. Sources of Parallelism

A parallel control method is one where an interpreter can divide a problem into independent parts, and then distribute those parts to other interpreters. In a multiprocessor system, this can lead to a faster solution of the problem if the other interpreters are running on physically different computers.

One possible parallel control is based on a parallel search of a goal tree. When a node has more than one descendant subtree, an interpreter can continue searching one of the subtrees itself, and distribute the other subtrees to other interpreters. The expected speedup in execution will be obtained if one of the interpreters derives the null clause more quickly than an interpreter that performs a simple depth first search. The amount of time required in such a system (ignoring intercomputer communication time) will be proportional to the shortest path from the initial goal to a null clause, whereas the amount of time required by a depth first interpreter is proportional to the sum of the path lengths of every branch to the left of the first branch that ends in a null clause. For deterministic functions, the single answer is most often at the end of the leftmost (or only) branch in the search tree. In these cases, a parallel search will not speed up the execution at all, even when the function is inherently parallel.

Another possibility for parallel control stems from coroutines. Interpreters running on separate processors could perform the derivations based on producer and consumer literals in parallel. For a program with one consumer and one producer, the maximum speedup will be a factor of two. For larger and more complicated programs, greater time savings may be realized. This form of parallelism is static: the amount of parallelism, *i.e.* the number of processors that can be used to solve a problem, is simply a function of the structure of the program. A more dynamic form of parallelism is found in the unraveling interpreter of the Irvine Dataflow system [1, 27], where the amount of parallelism is a function of both the structure of the program and the size of the problem. For example, in a program for matrix multiplication, one coroutine might be defined to generate row/column pairs while another coroutine consumes the pairs and computes inner products. The parallelism is independent of the size of the matrices: there will

always be two coroutines. When matrix multiplication is done by an unraveling interpreter, all inner products can be computed at (roughly) the same time, and the amount of parallelism is a function of the number of inner products that need to be computed.

The parallel control to be defined in the remaining chapters is a form of dynamic parallelism, similar to the unraveling interpreter, that exploits the parallelism inherent in the definition of deterministic functions. At the same time, this control method is applicable to nondeterministic functions and more general relations, where more than one correct value must be computed.

2.7. Chapter Summary

This chapter has introduced logic programming as a formal system with three components: a syntax that is a usable subset of first order predicate calculus, a denotational semantics in which the meaning of a program is defined to be a relation, and an operational semantics, defined by the resolution proof procedure, that allows one to construct relations. Control was shown to be important for efficiency, in that it effects the order in which answers are found, but control does not effect the correctness of a program, in that alternative control methods do not compute different relations. The exception is that some control methods are more defined, meaning more of the relation can be constructed. A number of interesting alternatives to the simplest and most common control method were discussed. These alternatives illustrate some principles that will be used in the definition of the parallel control defined in the remaining chapters.

Research on logic programming and the Prolog language that is closely related to the dissertation research will be discussed in Chapter 7. A deeper treatment of other topics not so closely related can be found in the following papers.

Resolution and unification were first defined by Robinson, with the intended application of automatic theorem proving [52]. An overview of first order predicate calculus, an algorithm for transforming a general well formed formula into a set of clauses, and a lengthy discussion of resolution can be found in Nilsson's

book [43]. Martelli and Montanari, in a recent paper, define three efficient unification algorithms, and discuss *occur checking*, a topic not covered in this chapter [38].

The DEC-10 Prolog system is one of the most widely used implementations of Prolog [46]. One of its unique features is a compiler that generates object programs that are comparable in execution speed to compiled LISP programs [59, 60]. Papers on implementing Prolog have been written by Roberts [51], Colmerauer [13], and, most recently, van Emdem [23], among others.

The cut symbol (also known as "slash") provokes some lively discussions about the meanings of programs and programming style; a recent contribution to this discussion is by van Emden [24]. Negation as failure was first defined by Clark [10]. The paper by Dahl on the CHAT database also has a complete discussion of the problems of this method for defining negation in resolution based logic programming systems [16].

Pereira has written a number of papers on the subject of control in logic programs. His work with Porto on intelligent backtracking will be discussed in a later chapter. In addition, Pereira has defined a language called Epilog that allows programmers to define control by using Horn clauses, thus effectively allowing the definition of special-purpose control constructs for special situations that arise in the user's program [49].

Two languages that are closely related to Prolog, yet not based on the resolution rule, are LPL [28] and Relational Programming [36]. LPL was defined in Haridi's thesis, and more recently Haridi and Ciepielewski have investigated possible sources of parallelism in this language [9]. In the relational programming system of MacLennan, entire relations are computed at the same time, and operations are performed on relations as a whole instead of on individual tuples within the relation, which is the case in Prolog.

A number of large and useful applications have been written in Prolog. Among these applications are the natural language query processor of the CHAT relational database system [16, 62, 63], Warren's problem solving program [58],

and Kibler and Porter's episodic learning program [32]. The use of Prolog as a metacompiler was described by Warren [61], and a comparison of definite clause grammars (grammars in which the rules are very similar to the Horn clauses of Prolog programs) and augmented transition networks for processing natural language was given in the paper by Pereira and Warren [45]. Finally, short, informal descriptions of new systems and applications are periodically published in the international *Logic Programming Newsletter* [35].

CHAPTER 3

The AND/OR Process Model

In the parallel control method defined in this dissertation, a logic program is solved by a set of *processes* that communicate via *messages*. A process is a data structure, consisting of state information and a program segment. In the AND/OR process model, there are two types of processes. An *AND process* is created to solve a goal statement, a conjunction of one or more literals. An *OR process* is created by an AND process to solve exactly one of those literals. A process starts in an initial state, and through a series of discrete transformations it is mapped into a final state. Each transformation is "triggered" by exactly one input message, and any transformation may cause one or more output messages to be sent to another process.

This chapter introduces the AND/OR Process Model by defining the basic requirements of AND and OR processes and the kinds of messages they generate. The processes defined in this chapter are sequential in nature: the tasks carried out by subprocesses are done one at a time, and for any given program the resulting computation is equivalent to a depth first interpretation, in terms of the sequence of operations performed. The purpose of this chapter is to show that logic programs can be interpreted by dividing them into smaller pieces that can be solved by independent interpreters. The next two chapters will define parallel processes, processes that solve their piece of the problem by creating more than one subprocess simultaneously.

3.1. Oracle

The decomposition of a logic program into a set of AND and OR processes is based on the notion of an *oracle*, which is an interpreter or machine that solves some problem in one step relative to the interpreter that consults it [30].

The use of oracles in defining independent computations can be illustrated using the following definition of a function that computes the sum of the squares of its two inputs:

$$ssq(X,Y,Z) \leftarrow product(X,X,X2)\ \&\ product(Y,Y,Y2)\ \&\ sum(X2,Y2,Z).$$

One way to compute the sum of the squares of the integers one through four is by the following goal statement:

$$\leftarrow ssq(1,2,A)\ \&\ ssq(3,4,B)\ \&\ sum(A,B,C).$$

The first five derivations performed by a depth first interpreter are:

0: $\leftarrow ssq(1,2,A)\ \&\ ssq(3,4,B)\ \&\ sum(A,B,C).$
1: $\leftarrow product(1,1,X2)\ \&\ product(2,2,Y2)\ \&$
$sum(X2,Y2,A)\ \&\ ssq(3,4,B)\ \&\ sum(A,B,C).$
2: $\leftarrow product(2,2,Y2)\ \&\ sum(1,Y2,A)\ \&\ ssq(3,4,B)\ \&\ sum(A,B,C).$
3: $\leftarrow sum(1,4,A)\ \&\ ssq(3,4,B)\ \&\ sum(A,B,C).$
4: $\leftarrow ssq(3,4,B)\ \&\ sum(5,B,C).$

After four steps, the interpreter has solved $ssq(1,2,A)$ and bound the variable A to the term *5*. All four steps are part of the solution of $ssq(1,2,A)$; no resolutions in this sequence are based on any other literal from the initial goal statement. The last goal statement shown above contains every literal except $ssq(1,2,A)$ from the initial goal; $ssq(1,2,A)$ has been *resolved away.*

The most important thing to notice about this sequence is the set of variables that are instantiated in goal statement 4: the only variables instantiated during this sequence of resolutions are those that occur in the literal that was resolved away. No other variables can be instantiated. In this example, the variables of the original goal are A, B, and C; only A was in $ssq(1,2,A)$, the literal that was resolved away, and A is the only variable that can possibly be bound by these resolutions. Furthermore, the possible bindings for A come from the tuples of **D**(ssq), the denotation of *ssq*.

For the general case, consider an interpreter that resolves away a literal L from a goal statement $\mathbf{G}^1$ by N resolutions, in the process generating a sequence

$\mathbf{G}^1 \cdots \mathbf{G}^N$ of goal statements. This interpreter could solve L in *one inference step* by consulting an oracle to provide a tuple from **D**(L), constructing a positive literal L' with the terms from this tuple, and then generating $\mathbf{G}^N$ by resolving L' with $\mathbf{G}^1$. An interpreter that can consult more than one oracle has the potential for exploiting parallelism. In order to do this, the interpreter must analyze a goal statement and identify two or more literals that can be solved independently and simultaneously by oracles; this topic will be discussed in Chapter 5

An overview of an interpretation in the AND/OR Process Model is that OR processes are essentially oracles. They are created to answer questions about exactly one literal. AND processes consult one oracle for each literal in the body of the clause they solve, and coordinate the answers from the oracles until a set of answers has been found that satisfies all literals simultaneously.

3.2. Messages

As a computation in the AND/OR Process Model proceeds, a tree of processes will be created. The initial goal statement is used to define the process at the root of the tree. As a result of some transformations, a process may create new processes as its descendants; these new processes will solve parts of the problem their parent was created to solve. The process tree is not the same as the goal tree defined earlier: the subproblems solved by descendants are not the same as the subproblems of the goal tree, and the parallel control to be described in the next two chapters is not a parallel search of a goal tree.

All messages sent during a computation are either from a process to one of its immediate descendants, or from a process to its parent. Messages are never sent between siblings or any other "family tree" relation. Messages sent to descendants are *start*, *redo*, or *cancel*, and messages sent to the parent are *success* or *fail*.

The start message is self-explanatory. When a process has reached a state where it has one or more independent subproblems to solve, it creates descendant processes (with appropriately defined initial states to be defined later) to solve them, and sends them all start messages.

A success message is sent to the parent when a process has solved the task given to it by its parent. That task is represented as a set of literals. The success message contains a copy of this set of literals, with variables instantiated. For example, if the subproblem is to solve the literal $p(X)$, and it can be solved by binding X to 0, then the success message will be the term $success(p(0))$.

A fail message is sent when a process cannot solve its problem. After sending the fail message, the process is transformed into the final state and terminates.

When a process has received an answer from a descendant, and later finds it cannot use that answer, it can send a redo message to the descendant, telling it to solve its subproblem in another way. This means the parent needs a different set of bindings for the variables in the subproblem.

Finally, a process may reach a state where it will never use any success messages that a descendant may send, in which case it sends the descendant a cancel message. A process that reads a cancel message is transformed directly into the final state and terminates.

3.3. OR Processes

As defined earlier, an OR process is created to solve exactly one literal. The basic requirements for an OR process will be described in this section. A detailed description of a *parallel* OR process, one that can have many descendants operating in parallel, is the subject of the next chapter.

An OR process created to solve a literal L must search the entire program for a clause with a head that can be unified with L. A sequential OR process searches the program linearly, from top to bottom, stopping when it encounters a clause with a head that matches L. If there are no such clauses, the process sends its parent a fail message and terminates.

There are two cases to consider when the OR process finds a clause with a matching head, depending on whether this clause is a unit clause or not. If L matches a unit clause, then the OR process can immediately construct a success message for its parent. For example, if L is $p(a,X)$, and there is a unit clause

$p(Z,b) \leftarrow .$

in the program, then the OR process can send *success*$(p(a,b))$. If L matches the head of an implication, the OR process creates a descendant AND process to solve the body of the implication. For example, if L is $p(a,X)$, and the program contains the implication

$p(V,W) \leftarrow q(V)\ \&\ r(W).$

then an AND process is created to solve the goal statement

$\leftarrow q(a)\ \&\ r(X).$

When the descendant AND process sends a success message (such as *success*$(q(a)\ \&\ r(b))$, denoting the fact that X was bound to b), then the OR process constructs a success message for its own parent.

If an OR process receives a redo message from its parent, it must solve its problem another way. Again, there are two possibilities, depending on whether or not the previous answer was created from a unit clause. If the previous answer was formed from a unit clause, the sequential OR process must resume its search for another clause to unify with L; if there are no more clauses that match L, then the process sends a fail message to its parent. If the previous answer was obtained from a nonunit clause, *i.e.* it was based on a success from a descendant, then that descendant is sent a redo message, and the OR process waits for a response from the descendant.

When an OR process receives a fail message from an AND descendant, it must find another clause that matches L. This could lead to the creation of a new descendant (if L matches the head of another implication), an immediate success (if L matches a unit clause), or failure (if there are no more clauses with heads that match L).

An OR process sends a cancel message to its descendant only when it receives a cancel from its own parent.

3.4. AND Processes

An AND process must solve all of the literals in the goal statement given to it by its parent. The literals are solved by creating OR processes for each one. A sequential AND process solves the goal statement much the same as an interpreter that performs a depth-first search. An OR process is created for the leftmost literal of the goal statement. If the OR process sends a success message, bindings in the answer are applied to the remaining unsolved literals, and an OR process is created for the very next literal in the goal statement. If the OR process sends a fail, then the most recently solved literal must be redone, *i.e.* a redo message is sent to the OR process created to solve that literal.

An AND process can send its parent a success after all descendant OR processes have sent successes. A sequential AND process fails if the first literal cannot be solved, *i.e.* if the OR process for the leftmost literal sends a fail message. When an AND process receives a redo from its parent, it in turn must send a redo to one of its own descendants; in a sequential AND process, this will be the process created to solve the rightmost literal.

3.5. Interpreter

An interpreter that executes a logic program by decomposing it into AND and OR processes has been implemented in DEC-10 Prolog. Since both AND and OR processes can be either parallel or sequential in nature, there are actually four different interpreters. The measurements and examples used in this section are from interpreter APOP (*A*nd *P*arallel - *O*r *P*arallel), but in fact all four interpreters produce the same kinds of output. All four interpreters share the same kernel of scheduling procedures, performance measuring routines, message passing primitives, and other low level supporting code. Details of the implementation of the kernel and of parallel processes are given in Appendix I.

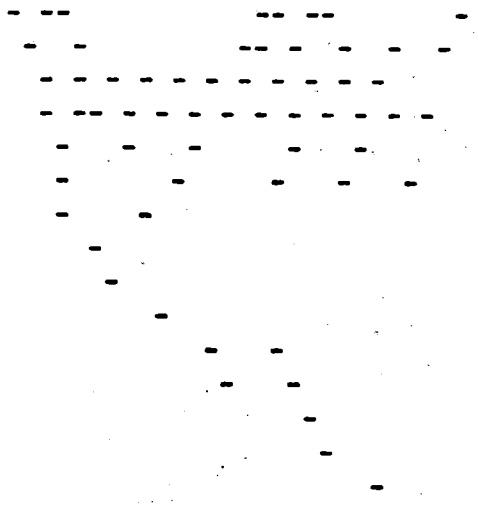
After it solves a problem, the interpreter prints out the number of processes created, and, for each process, the number and size of each kind of message sent. Associated with each process and each message is a "time stamp", represented as an integer. The interpreter is able to use this information to create plots, such as

the one shown in Figure 9, that show the relationship between the transformations performed on the processes. The vertical axis represents the number of processes; the transformations of one process are plotted on one line. The horizontal axis represents time. The interpreter records the fact that each transformation takes one time unit. When a message bearing time stamp T triggers a transformation that causes other messages to be sent, those new messages will have time stamp T+1. If the interpreter transforms process number P at time T, then a dash will be plotted at coordinates (P,T). Note that if P sends a message to Q as part of the transformation plotted at (P,T), there will be a dash at (Q,T+1) as a transformation of Q is triggered by this message. The plot in Figure 9 was produced by the solution of

$\leftarrow$ *paper*(*P*,1978,*uci*).

with interpreter APOP.

The plots provide an estimate of the amount of parallelism possible. Wherever there are two dashes plotted for the same time (same column), there is the possibility that two processing elements could be performing the corresponding state transitions in parallel. The plots are *not* to be construed as simulation results, giving a realistic timing of a parallel execution. In physical terms, such plots could only be realized on a system that has an infinite number of processing elements, each dedicated to solving just one process, and where each processor is capable of passing a message to any other processor in a constant amount of time. This interpreter was built to see if there is parallelism to be found in logic programs. The answer is "yes" if the plots show more than one dash in a column. The problem of mapping processes onto processing elements, of "distributing the dashes" so that a parallel solution is performed when problems are actually solved by a physical network of processing elements, will be discussed in Chapter 6.



Maximum number of dashes per column = 4

15 processes executed 62 steps in 28 time units: 2.22

Messages Summary: 69 messages sent, using 573 characters.

Process ID	Number of Successes	Size of Successes	Number of Fails	Number of Redos	Number of Starts	Number of Cancels
1	3	(91)	1	3	1	0
2	3	(85)	1	2	2	0
3	1	(35)	1	4	5	4
4	1	(58)	1	5	6	5
5	4	(93)	1	0	0	0
6	4	(93)	1	0	0	0
7	1	(23)	1	0	0	0
8	0	(0)	1	0	0	0
9	0	(0)	1	0	0	0
1	0	(0)	1	0	0	0
1	1	(26)	0	0	0	0
1	1	(19)	0	0	0	0
1	0	(0)	1	0	0	0
1	0	(0)	1	0	0	0
1	0	(0)	1	0	0	0

Interpreter output showing plot of solution of *paper(P,1978,uci)*. Process 2 is the parallel OR process created to solve that literal; the state transitions are explained in detail in Chapter 4. Processes 3 and 4 are parallel AND processes created to solve the bodies of the implications in the procedure for *paper*. The transitions of process 4 are described in detail in Chapter 5.

Figure 9. Interpreter Output

3.6. Programming Language

Many of the extensions to the formalism of logic programming included in most Prolog systems are meaningful only in single processor, sequential systems. Most notable are *assert* and *retract*, which modify the database of clauses in the program, and the cut symbol, which is used to guide the global search process.

The language supported by the parallel interpreter also extends the formalism, but only with constructs that make no assumptions about the number of processors available to interpret the program or about whether the processors have access to a common memory. The eventual hardware is presumed to be a collection of asynchronous, autonomous processing elements, each with its own local memory and its own copy of the program being interpreted. The AND/OR Process Model may eventually be implemented on a multiprocessor in which processors share a common memory, but at this time it is best to make the worst case assumption that the processors will be completely independent. It will be easier to optimize a model that makes no assumptions about shared memory when it is implemented on a system with shared memory; it will be much harder to implement a model that assumes a common memory on a system that does not have common memory.

The extensions to logic programming supported by the interpreter are:

-- The evaluable predicate *is* for performing arithmetic operations (Section 2.4.1).

-- Definition of $\leftarrow$ and $\&$ as infix operators, and the evaluable predicates =.. and *call*, thus allowing higher order functions.

-- Negation as failure.

-- Conditional expressions.

The last two extensions, negation and conditional expressions, are implemented in Prolog in terms of the cut symbol (see Section 2.4.3). The same behaviors (*e.g.* a goal that fails when its argument is a goal that succeeds) can be implemented in the AND/OR Process Model, but by using specially defined processes instead of the cut symbol.

Negation as failure is implemented by a special OR process, created whenever the literal to be solved is of the form $not(G)$. This OR process will create an AND descendant to solve the goal G passed as a parameter. If the descendant returns a fail message, then the OR process sends $success(G)$ to its own parent; if the descendant sends $success(G)$ then the OR process sends fail to its parent and cancel to the descendant.

Conditional expressions can be written in DEC-10 Prolog, using $\rightarrow$ and ; as infix predicate symbols:

$f :- p \rightarrow q ; r.$

The Prolog interpreter translates clauses of this form into two separate clauses, one of which has a cut symbol(Section 2.4.3):

$f :- p, !, q.$
$f :- r.$

One way of implementing conditional expressions in the AND/OR model also involves translation into two new clauses, but neither contains a cut symbol:

$f \leftarrow p \ \& \ q.$
$f \leftarrow not(p) \ \& \ r.$

Another method for implementing conditionals, by using special AND processes, is presented in Chapter 6.

3.7. Chapter Summary

The AND/OR Process Model provides a framework for interpretation of logic programs that allows an interpreter to identify subcomputations that can be performed by independent interpreters. The independent interpreters are processes that communicate with their parent process via messages. The description presented in this chapter was a description of the minimum behavioral requirements of each kind of process. A global perspective of the computation performed by these minimal processes would show the same computation carried out by a standard depth first interpreter, since the same sequence of inferences is generated, and in the same order.

The depth first interpretation is defined in terms of a search of a global tree of goal statements, strongly implying a von Neumann architecture for the underlying hardware. The AND/OR Process Model, on the other hand, presents a method for interpretation by small, asynchronous, and logically independent processes that communicate only through messages. Thus the first step in the design of a highly parallel architecture for logic programs has been taken: it has been shown how a logic program can be executed by independent interpreters. The next step is to show how these interpreters can exploit parallelism by creating a number of processes that carry out subtasks simultaneously.

CHAPTER 4

Parallel OR Processes

An OR process is the embodiment of an oracle, an independent interpreter created to solve a goal statement of exactly one literal. An OR process created to solve an n-ary literal $p(X_1 \cdots Xn)$ is expected to construct the set $\mathbf{D^1}$(p), *i.e.* it must construct sets of n-tuples $<t_1 \cdots t_n>$ such that $p(t_1 \cdots t_n)$ is provable.

OR processes do not attempt to construct the entire relation all at once. In this respect, they behave in the same manner as sequential interpreters, since they respond with the first tuple that satisfies the initial goal. After reporting this first response, they are suspended, and only a request for additional answers causes the process to send additional answers.

If a procedure $p(X_1 \cdots X_n)$ is defined by more than one clause, sequential OR processes (and sequential, depth first interpreters) construct $\mathbf{D^1}$(p) by first obtaining all tuples defined by first clause, then all tuples defined by the second clause, and so on. When $\mathbf{D^1}$(p) is finite, the process fails after obtaining the last tuple defined by the last clause. The issue of infinite relations, and the effect of infinite branches on both sequential interpreters and OR processes, will be discussed further in the summary at the end of this chapter.

Relations have been defined to be *unordered* sets of tuples, so the ordering of tuples defined above is not necessarily part of the meaning of a predicate. In particular, a parallel control structure could construct $\mathbf{D^1}$ by interleaving tuples defined by the various clauses in the procedure. The parallel OR processes defined in this chapter attempt to construct answers based on all clauses simultaneously. $\mathbf{D^1}$ is assembled as the messages from the descendant processes arrive, and the order of tuples depends only on the timing of the success messages received.

4.1. Operating Modes

When an OR process is first created, it assumes that its parent AND process is waiting for an answer. The first tuple constructed by the OR process should be sent (via a success message) to the parent. After this, however, the tuples should be saved and not sent to the parent until that process sends a redo message. The OR process acts as a message center, deciding when to transmit results and when to store them.

An OR process is in *waiting mode* when its parent is waiting for an answer, and is in *gathering mode* when the parent is busy, using the answer sent previously. Processes will switch back and forth between these two operating modes. The rules for changing from one mode to the other are based on the order of success and redo messages received, and on the number of tuples that have been constructed but not yet sent to the parent.

4.2. State Transitions

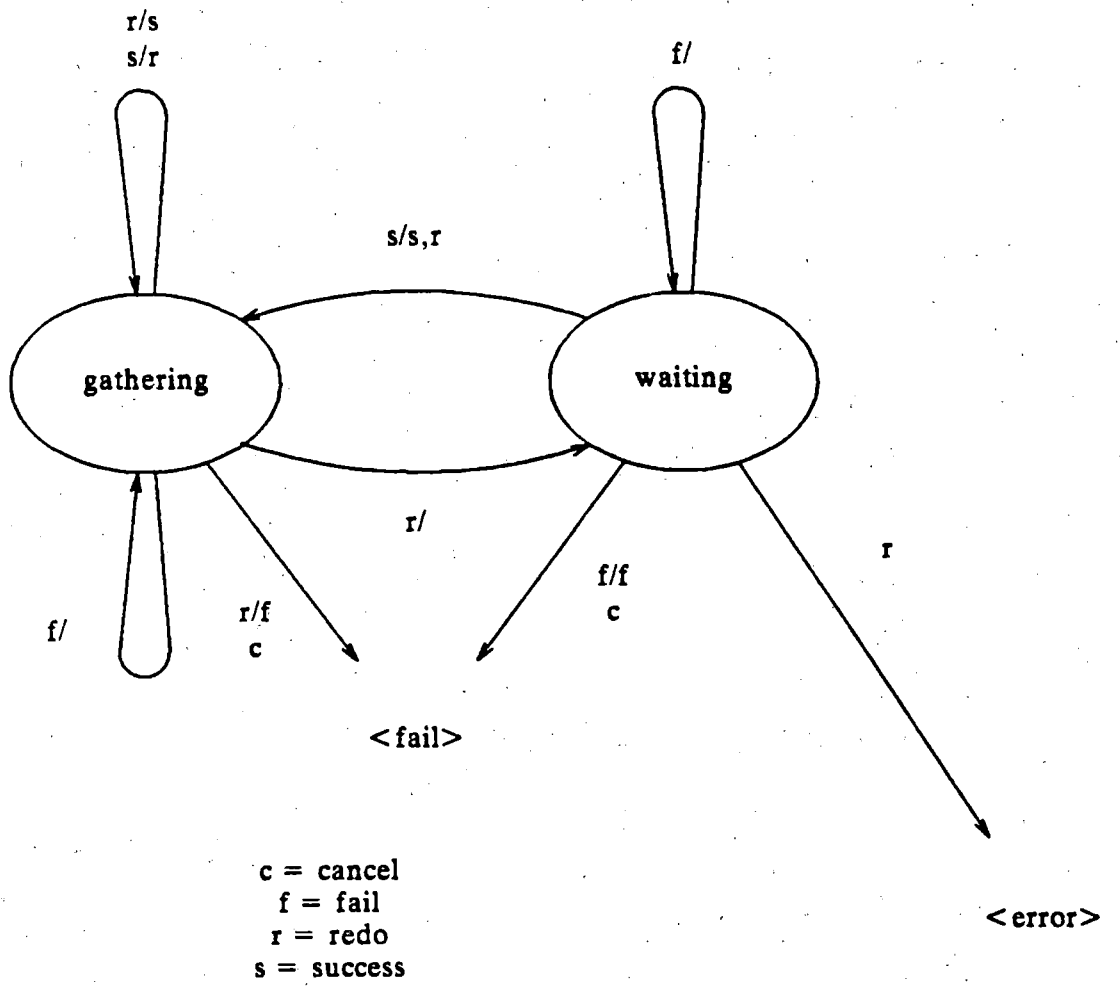
The diagram of Figure 10 summarizes the conditions under which a process changes states. The following sections will describe in detail how a parallel OR process reacts to the messages it receives. Every transition in Figure 10 will be explained. State transitions in the figure have labels of the form X/Y, meaning that the transition was triggered because the OR process received message X, and that as a result of the transition it is transmitting message Y. The labels can be 'f' (for fail), 's' (for success), 'c' (for cancel), or 'r' (for redo). Throughout these discussions, the following symbols will be used:

-- L will stand for the literal to be solved by the OR process.

-- $\mathbf{S}^i$ is the input state (the state of the process before a message is processed) and $\mathbf{S}^o$ is the output state.

-- WL is the waiting list, a list of answers not yet sent to the parent, and SL is the list of answers that have been sent.

-- DL is a list of IDs of descendant processes.



A parallel OR process works in one of two modes: gathering or waiting. The transitions between modes are summarized here. A label X/Y on an arc means message type X was the input message, and message type Y is output (see text for explanation of multiple arcs).

This figure originally appeared in "Parallel Interpretation of Logic Programs", by Conery and Kibler [15].

Figure 10. Operating Modes of Parallel OR Process

4.2.1. Start Message

In its initial state, an OR process has no descendants, and *WL* and *SL* are both the empty list. When the process receives the start message from its parent, it creates descendant AND processes for every implication with a head that matches ***L***, adds the IDs of these processes to *DL*, and constructs answers to send the parent for every unit clause that matches ***L***.

The following states may result:

-- If no clause has a head that unifies with ***L***, then the OR process fails immediately. $\mathbf{S}^o$ will be *done* (the final state), and a fail message is sent to the parent.

-- If ***L*** unifies with the heads of $N > 0$ implications, but no unit clauses, then N descendant AND processes are created, and the process will be in *waiting* mode in $\mathbf{S}^o$. *SL* and *WL* will remain empty lists.

-- If ***L*** unifies with the heads of $N \geq 0$ implications and $M > 0$ unit clauses, then create N descendant AND processes (as above), and also construct answers from the M unit clauses. Let $M1$ be one of those answers, and *MR* be the remaining answers. Send *success*($M1$) to the parent. The OR process will be in gathering mode in $\mathbf{S}^o$, with *WL* equal to *MR*, and *SL* equal to [$M1$].

4.2.2. Success Message

Whenever a parallel OR process receives a success message from a descendant, it responds by sending that descendant a redo, causing it to immediately start working on its next answer. Further processing depends on whether the OR process was in gathering or waiting mode in $\mathbf{S}^i$.

Transition marked s/s *from waiting mode*

When a *waiting* OR process receives a success message from one of its descendants, it creates an answer A and sends *success*(A) to its parent. The process switches to gathering mode in $\mathbf{S}^o$, with A appended to *SL*. *WL* is unchanged.

Transition marked s/ *from gathering mode*

If a *gathering* OR process receives a success message, it constructs answer A, but does not send it; A is appended to WL, SL is unchanged, and the process remains in gathering mode.

4.2.3. Fail Message

Whenever an OR process receives a fail message from a descendant, the ID of the failed process is removed from the descendant list DL.

Transition marked f/ *from waiting mode*

If there are still descendants working (*i.e.* DL is not the empty list after removing the failed descendant), then no further action is required; remain in waiting mode.

Transition marked f/f *from waiting mode*

If DL is now the empty list, then send fail to the parent, since the parent is waiting for some response, and there is now no way to construct another answer.

Transition marked f/ *from gathering mode*

No further action is required. Remain in gathering mode, and do not send a fail message yet, even if DL is now the empty list, since the parent is currently busy.

4.2.4. Redo Message

By definition, an OR process is in *waiting* mode if its parent is waiting for an answer. A redo message in this case denotes a system error condition.

A *gathering* OR process handles a redo from its parent in one of three ways, depending on the states of WL and DL:

Transition marked r/s *from gathering mode*

If WL, the list of answers not yet sent to the parent, is not empty, then select one answer A from WL. Send *success*(A) to the parent, append A to SL, and remove A from WL. The OR process remains in gathering mode.

Transition marked r/f *from gathering mode*

If *WL* is empty, then check the list of descendants *DL*. If *DL* is also empty, then there is no way to make another answer. Send the parent fail, and terminate.

Transition marked r/ *from gathering mode*

If *WL* is empty but *DL* is not, meaning there is still a chance that an active descendant can produce further answers, then go into waiting mode. *SL* and *WL* are not changed.

4.3. Example

Figure 11 (the four page figure at the end of the chapter) shows the states of the parallel OR process created to solve the literal

paper(*P*,1978,*uci*).

The states are from the printout produced by the trace mechanism in the interpreter. This process is Process 2 from the plot of Figure 9 (Chapter 3).

The first transition occurs when the process receives the start message from process 1. There are three clauses that have heads that can be unified with *paper*(*P*,1978,*uci*). One of them is the unit clause *paper*(*xform*,1978,*uci*), so an answer based on that literal is sent to the parent, and the process is in gathering mode in the next state. Processes 3 and 4 are created to solve the bodies of the other two clauses. Note that *SL* contains the answer sent and *WL* is empty.

The second transition is triggered by a redo message from the parent. There are no answers in *WL*, so the process goes into waiting mode until a message arrives from either descendant.

Transition <3> occurs when one of the descendants, process 4, sends a success message. Since the parent is waiting, this answer is sent immediately (and also appended to *SL*). Process 4 is sent a redo message, and the OR process goes back to gathering mode.

While the process is still in gathering mode, the other descendant sends a success. This answer is appended to *WL*, the descendant is sent a redo message,

and the process remains in gathering mode.

Transition <5> is triggered by a redo message from the parent. There is an answer ready for it in *WL*, so this answer is sent immediately. The process remains in gathering mode.

> Note: the success and the redo read in transitions <4> and <5> may be processed in either order, and the state of the OR process will still be the same.

A redo message is received from the parent (transition <6>), and *WL* is empty, so the OR process goes into waiting mode.

The next transition occurs when one descendant sends a fail message. The record of the descendant (3) is removed from *DL*, and the OR process remains in waiting mode.

> Note: the order of arrival of the fail and redo messages that trigger transitions <6> and <7> is also not important.

Finally, the last remaining descendant, process 4, sends a fail. There are now no more active descendants, and the OR process can not make any more answers. The parent is sent a fail message, and the output state is *done*.

4.4. Chapter Summary

Parallel OR processes have the same I/O behavior as their sequential counterparts defined in the previous chapter. They are independent interpreters, created to solve goal statements of one literal. If the goal is solvable, they respond with a success message containing one tuple from the denotation of that literal; otherwise they send back a fail message. Additional answers are not sent until the parent process sends a redo message. The main difference between sequential and parallel OR processes is that parallel processes are message centers, coordinating the actions of multiple concurrent descendants, whereas sequential processes need to monitor at most one descendant process at any one time.

Sequential OR processes are sensitive to the order of the clauses in a program. The order of the answers sent to a parent is a function of the relative

order of the clauses. A side effect of this ordering is that it is possible to construct a procedure for p for which $\mathbf{D}^1 \neq \mathbf{D}^2$. A simple example is provided by a procedure that contains the clause

$$p \leftarrow p.$$

When a depth first interpreter encounters such a clause in a procedure for p, it generates an infinite subtree in the search space. This interpreter will never find any answers to the right of the infinite subtree, answers which may be defined by later clauses for p. Sequential OR processes will also be trapped in an infinite computation by this clause: the body is used to start an AND process, and then the AND process starts an OR process to solve p, and then that OR process starts another AND process for the same clause, and so on.

Parallel OR processes, on the other hand, have the power to find more answers than either sequential OR processes or depth first interpreters. Parallel OR processes create the same nonterminating AND processes for the same clauses that sequential OR processes do, but the parallel processes are able to obtain answers from other clauses. This does not guarantee that OR processes construct all of $\mathbf{D}^1$(p). It is still possible to construct a set of clauses such that the null clause can be derived through a series of resolutions, but for which the control strategies of the AND/OR process model are not capable of deriving any null clause. These pathological cases will be described at the end of the next chapter, since they concern the method for solving literals in the body of a clause in parallel.

Key to the State Information of Parallel OR Processes

Most of the state information for an OR process is self-explanatory. The operating mode (waiting or gathering) is given on the first line. The goal given on the second line is a copy of the original goal the OR process was created to solve; the variables in this copy are never bound. The list of active descendant AND processes keeps three pieces of information about each descendant: its process ID number, and a copy of the complete clause, both head and body, of the clause used to create that process.

The numbers **<N>** stand for transition numbers.

<1>

OR Process 2 (gathering) after 'start' from Process 1, T = 1
  Goal: paper(P,1978,uci)
  Parent ID: 1
  Descendant List [Process,Head,Body]:
    [3, paper(P',1978,uci), [tr(P',uci),date(P',1978)]]
    [4, paper(P'',1978,uci), [date(P'',1978), author(P'',A''),loc(P'',uci)]]

  Answers Sent: Answers Waiting to be Sent:

  paper(xform,1978,uci)

Figure 11. States of a Parallel OR Process

<2>

OR Process 2 (waiting) after 'redo' message from Process 1, T = 4
Goal: paper(P,1978,uci)
Parent ID: 1
Descendant List [Process,Head,Body]:
[3, paper(P',1978,uci), [tr(P',uci),date(P',1978)]]
[4, paper(P'',1978,uci), [date(P'',1978), author(P'',A''),loc(A'',uci)]]

Answers Sent: Answers Waiting to be Sent:

paper(xform,1978,uci)

<3>

OR Process 2 (gathering) after 'success([date(eft,1978),
author(eft,kling),loc(kling,uci)])' from Process 4, T = 14
Goal: paper(P,1978,uci)
Parent ID: 1
Descendant List [Process,Head,Body]:
[3, paper(P',1978,uci), [tr(P',uci),date(P',1978)]]
[4, paper(P'',1978,uci), [date(P'',1978), author(P'',A''),loc(A'',uci)]]

Answers Sent: Answers Waiting to be Sent:

paper(eft,1978,uci)
paper(xform,1978,uci)

Figure 11 Continued

<4>

OR Process 2 (gathering) after 'success([tr(df,uci),date(df,1978)])'
from Process 3, T = 15
Goal: paper(P,1978,uci)
Parent ID: 1
Descendant List [Process,Head,Body]:
[3, paper(P',1978,uci), [tr(P',uci),date(P',1978)]]
[4, paper(P'',1978,uci), [date(P'',1978), author(P'',A''),loc(A'',uci)]]

Answers Sent: Answers Waiting to be Sent:

paper(eft,1978,uci) paper(df,1978,uci)
paper(xform,1978,uci)

<5>

OR Process 2 (gathering) after 'redo' from Process 1, T = 17
Goal: paper(P,1978,uci)
Parent ID: 1
Descendant List [Process,Head,Body]:
[3, paper(P',1978,uci), [tr(P',uci),date(P',1978)]]
[4, paper(P'',1978,uci), [date(P'',1978), author(P'',A''),loc(A'',uci)]]

Answers Sent: Answers Waiting to be Sent:

paper(df,1978.uci)
paper(eft,1978,uci)
paper(xform,1978,uci)

Figure 11 Continued

<6>

OR Process 2 (waiting) after 'redo' from Process 1, T = 20
  Goal: paper(P,1978,uci)
  Parent ID: 1
  Descendant List [Process,Head,Body]:
    [3, paper(P',1978,uci), [tr(P',uci),date(P',1978)]]
    [4, paper(P'',1978,uci), [date(P'',1978), author(P'',A''),loc(A'',uci)]]

  Answers Sent: Answers Waiting to be Sent:

  paper(df,1978,uci)
  paper(eft,1978,uci)
  paper(xform,1978,uci)

<7>

OR Process 2 (waiting) after 'fail' from Process 3, T = 23
  Goal: paper(P,1978,uci)
  Parent ID: 1
  Descendant List [Process,Head,Body]:
    [4, paper(P'',1978,uci), [date(P'',1978), author(P'',A''),loc(A'',uci)]]

  Answers Sent: Answers Waiting to be Sent:

  paper(df,1978,uci)
  paper(eft,1978,uci)
  paper(xform,1978,uci)

<8>

OR Process 2 after 'fail' from Process 4, T = 26
done

Figure 11 Continued

CHAPTER 5

Parallel AND Processes

Sequential AND processes, as defined in Chapter 3, simply mimic sequential interpreters by solving their subgoals one at a time, from left to right. A parallel AND process is one that can solve more than one literal at any time. AND parallelism involves creating more than one OR process simultaneously, and then coordinating the responses to success and fail messages from these descendants until all literals have been successfully solved.

The brute force method for AND parallelism is to immediately create a process for every literal on the first step. There are three reasons why this will not be effective; all three reasons are based on the fact that the solution of one literal often binds variables that are arguments in other literals.

The first drawback to brute force parallelism is that the AND process must ensure that solutions for the different literals bind common variables to the same terms, and this may be quite difficult in the general case. For example, given the goal statement

$\leftarrow p(A,B)\ \&\ q(B,C)\ \&\ r(C,A).$

the AND process has to find tuples $<A,B,C>$ that satisfy all three predicates at the same time.

A second argument against solving all literals at once is that by waiting until variables in literals are bound (via the solution of other literals), the OR process created to solve those literals may be more efficient: there are often fewer solutions, and fewer fruitless choices made in constructing those solutions (Section 2.5).

Finally, and of most practical importance, some literals *fail* if an attempt is made to solve them before a sufficient set of variables are instantiated; these are the literals with thresholds or mode declarations (Section 2.4.1). For example, in

the goal statement

$\leftarrow$ *length*(*L*,*N*) *& X is* 2**N*.

the goal of multiplying N by two fails unless N is instantiated to an integer via the solution of the first literal.

An effective method for achieving AND parallelism is thus a problem of correctly ordering the literals, of deciding which literals must be done sequentially and which can be done in parallel. The implementation of AND parallelism defined in this chapter has three major components. There is an *ordering algorithm* that automatically decides, based on the current state of the goal list, the order in which the literals should be solved. The *forward execution* component actually creates the descendant OR processes; it handles success messages, and determines which (if any) literals can be solved as a result. The third component is known as *backward execution*, and handles fail and redo messages to decide which literal(s) must be re-solved before continuing forward execution.

5.1. Ordering of Literals

The basis for the ordering of literals in the body of a clause is the sharing of variables. Whenever two or more literals have a variable in common, one of the literals will be designated the *generator* for the variable, and it will be solved before the others. The solution of the generator literal is intended to create a value for the corresponding variable. After the generator has been solved, the other literals that contain that variable, the *consumers*, may be scheduled for solution. A generator will be defined for every variable in a goal statement. It is possible that the solution of a generator will not bind the variable, so that consumers still have a variable in common; this situation is discussed in section 5.2.

Generators and consumers are similar to the lazy producers and eager consumers of IC-Prolog [11]. The term "generator" is used here, since their action is more closely related to generators in other languages (see, for example, Alphard [66]), in that they produce a *sequence* of independent terms, as opposed to parts of a single complex term through a series of partial bindings. Note that a literal can be the generator of some variables and a consumer of others. This is

especially true when the literal is a function call, when some of the arguments are input arguments and the others must be uninstantiated variables that will be bound by the execution of the function.

5.1.1. Dataflow Graphs

Generator and consumer relationships can be shown pictorially as a *dataflow graph*. In these graphs there will be one node for each literal in a clause, including the head. There is a set of directed arcs for each variable in the clause. In each set, the arcs go from the generator literal to each literal that consumes the variable. An *immediate predecessor* of a literal ***L*** is defined to be a literal that is a generator for one of the variables in ***L***. A predecessor, in general, is either an immediate predecessor or a predecessor of an immediate predecessor.

The head of the clause is a special case. It is the generator of every variable that is bound when the process is created. The ordering algorithm of the next section requires this information, so the head of the clause (*i.e.* the literal being solved by the parent OR process) is included in the state information of a parallel AND process (see Appendix I for details). The head literal is also the consumer of variables that are not instantiated when the clause is called. In some of the pictures of dataflow graphs, arcs will be drawn from the generators of those variables to the head to indicate this fact.

5.1.2. The Ordering Algorithm

There are a number of rules one can use to identify generators. The first, mentioned above, is that the head of a clause is the generator for all variables that are instantiated when the clause is invoked.

Second, some of the literals in the body may have I/O modes (Section 2.4.1). These may be evaluable predicates, for which the system already knows the modes, or they may be user-defined functions, in which case the user must specify a mode declaration. A good example is the evaluable predicate *is* from DEC-10 Prolog, which has the mode declaration

:-mode is(?,+).

This declaration shows that *is* has two arguments. The question mark means that terms in that argument position can be either variables or nonvariable terms. The plus sign means that the corresponding argument *must* be a ground term: it must not contain any uninstantiated variables by the time the literal is selected for solution. Another possibility (not shown in this example) is a minus sign, which means that the corresponding argument must be an uninstantiated variable, and that as a result of the procedure call the variable will be instantiated. If there is a minus sign in the mode declaration for a predicate, then the ordering algorithm knows that a literal with this predicate symbol *must* be the generator for any variables used in that argument position. A plus sign means the literal can *never* be the generator for any variables occurring in the corresponding argument position. The above mode declaration indicates that a literal $is(X,Y)$ can never be a generator for any variables occurring in an expression in the second argument position, but might be the generator of a variable that is the first argument.

The two rules just described, that the head is the generator of variables that are bound when the procedure is called and that mode declarations cannot be violated, are the only two strict rules. By themselves, however, they are not sufficient to designate generators for every variable in the body of a clause. There are a number of heuristic rules that could be used in conjunction with the first two rules in order to both make sure all variables have generators and the resulting ordering is relatively efficient. A number of these heuristics were given in Section 2.5. The only heuristic currently implemented in the ordering algorithm is the *connection rule*, which is a special case of the rule that calls for selection of the literal with the largest number of instantiated variables. Briefly, when the connection rule is applied, it attempts to find a literal that consumes variables for which generators are already known, and which can be designated the generators of other variables that do not yet have generators. The connection rule is stated more concisely as step 3.a of the ordering algorithm (Figure 12).

The ordering algorithm uses the following variables:

B The set of literals in the body. Initialized to contain every literal in the body; when a literal is designated as a generator, it is removed from *B*.

S The set of variables for which generators have been specified.

U The set of variables for which generators are not known yet. Note that the union of *S* and *U* contains every variable in the clause.

The algorithm:

1. Identify as many generators as possible using mode declarations; remove those literals from *B*, and initialize *S* to be the variables generated by these literals.
2. Add to *S* the variables instantiated in the head of the clause; the head is the generator of these variables. Initialize *U* to be the set of all remaining variables.
3. Repeat until *B*=[] or *U*=[]:
   a. Make a set *LS* with every literal in *B* that has at least one variable in *U* and one variable in *S* {note: if *S* is empty, *LS* will also be empty}.
   b. If *LS* is empty after step (a), find the leftmost literal in *B* that has a variable in *U* and add it to *LS* {*LS* contains just this one literal}.
   c. For every literal *L* in *LS*, assign *L* as the generator of any variables in *U* that occur in *L*. Remove these variables from *U* and add them to *S*. Remove *L* from *B*.

Figure 12. The Literal Ordering Algorithm

Finally, if none of the three rules described above identifies a generator for a variable, the leftmost literal in which the variable occurs is designated as the generator for that variable. This rule, called the "leftmost rule", is included to make sure that every variable will have a generator.

Note that since mode declarations are known before a clause is called, the second rule can be applied when clauses are first loaded by the interpreter. The other rules can only be applied at runtime, once the pattern of variable instantiation in the clause is known. Logically, any literal can be the generator of any variable that appears as one of its arguments. The only exceptions are determined by evaluable predicates or user defined procedures with mode declarations. The ordering algorithm is used primarily to ensure that mode constraints are not violated, and secondarily to produce an efficient ordering.

The Prolog code and a more detailed description of the ordering algorithm are given in Appendix I. This algorithm was used to produce every dataflow graph pictured in this dissertation.

5.1.3. Examples of Literal Orderings

The ordering algorithm will be illustrated by four examples, each showing a different pattern of variable instantiation in the body of a clause. The dataflow graphs produced for these examples are shown in Figure 13.

Disjoint Subgoals

$f(X,Y) \leftarrow g(X)\ \&\ h(Y).$

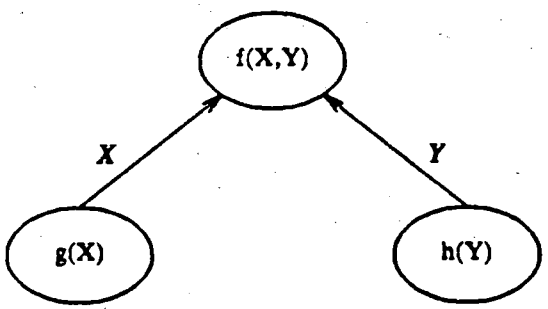
In this example the two literals in the body are clearly independent. The graph pictured is for the case when neither X nor Y is instantiated when the process is created. The leftmost rule was used to designate $g(X)$ as the generator of X and $h(Y)$ as the generator of Y. Note that if there are N_G solutions for $g(X)$ and N_H ways of solving $h(Y)$, then $\mathbf{D}^1$(f) will contain $N_G \times N_H$ pairs of X and Y values. The remaining pairs, after the first, will be created in response to redo messages; the method used to enumerate all pairs is described later in the section on backward execution.

Clause:

$f(X,Y) \leftarrow g(X)\ \&\ h(Y).$

Call:

$\leftarrow f(X,Y).$



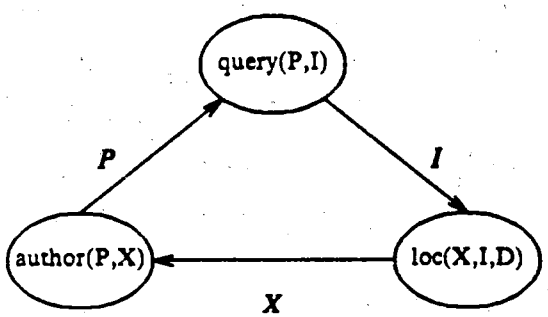
(13a)

Clause:

$query(P,I) \leftarrow author(P,X)\ \&\ loc(X,I,D).$

Call:

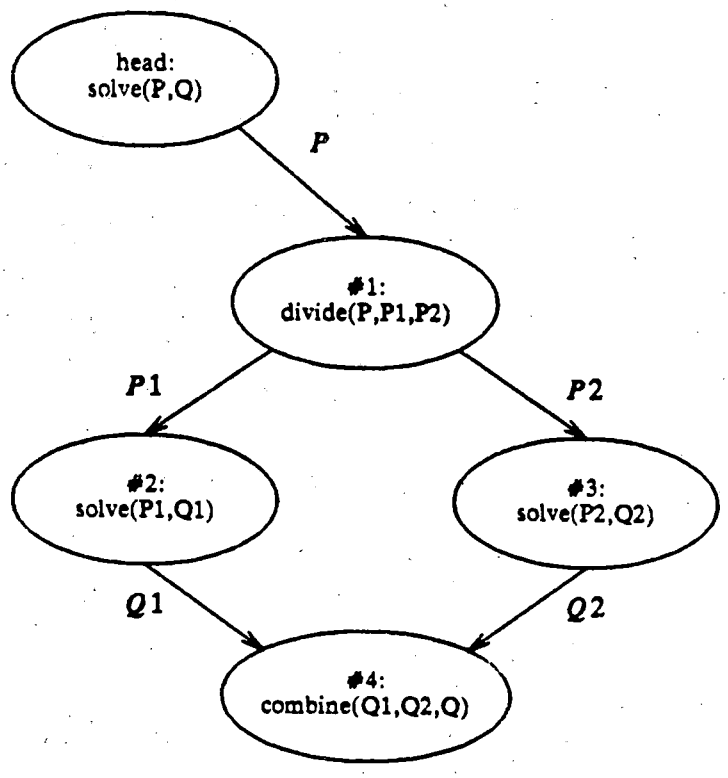
$\leftarrow query(P,uci).$



(13b)

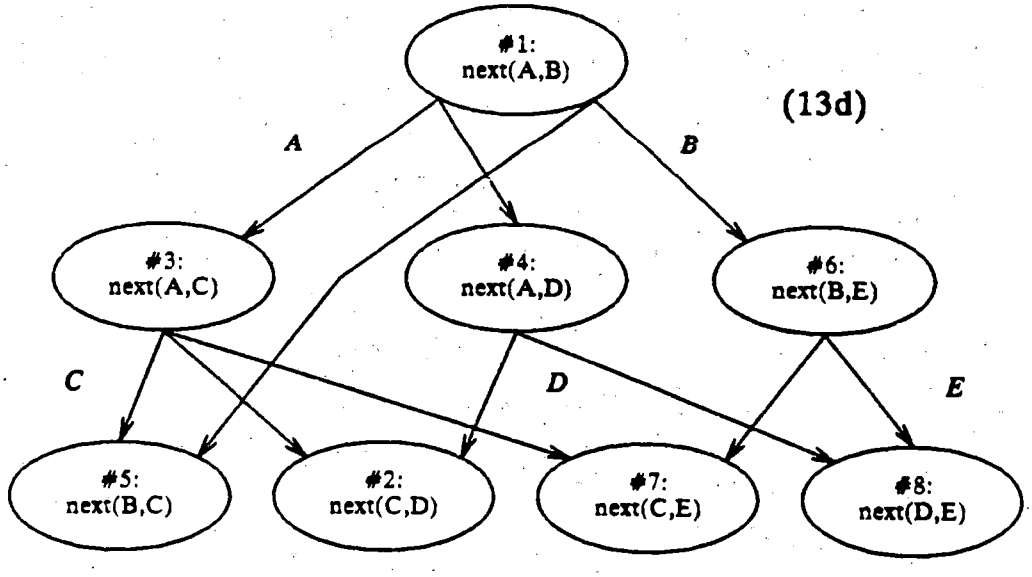
Figure 13: Dataflow Graphs

solve(*P*,*Q*) ← *divide*(*P*,*P*1,*P*2) &
solve(*P*1,*Q*1) &
solve(*P*2,*Q*2) &
combine(*Q*1,*Q*2,*Q*)



(13c)

color(*A*,*B*,*C*,*D*,*E*) ←
next(*A*,*B*) & *next*(*C*,*D*) & *next*(*A*,*C*) & *next*(*A*,*D*) &
next(*B*,*C*) & *next*(*B*,*E*) & *next*(*C*,*E*) & *next*(*D*,*E*)



(13d)

Figure 13 (Continued)

Shared Variable

$query(P,I) \leftarrow author(P,X)\ \&\ loc(X,I,D).$

The two subgoals have the variable X in common, and no call to *query* can ever cause X to be instantiated when the clause is selected. When the AND process is created, if I is instantiated but P is not, then the connection rule specifies that $loc(X,I,D)$ should be the generator of X. Otherwise, $author(P,X)$ is designated, either through the connection rule (if only P is instantiated, as in the picture in Figure 13b) or the leftmost rule (if neither or both are instantiated). This is an example of where the connection rule implements the optimal ordering described in Section 2.5.2, based on the number of instantiated variables in each literal.

Deterministic Function

$$\begin{aligned} solve(P,Q) \leftarrow\ & divide(P,P1,P2)\ \& \\ & solve(P1,Q1)\ \& \\ & solve(P2,Q2)\ \& \\ & combine(Q1,Q2,Q). \end{aligned}$$

This clause illustrates the general form of a deterministic function expressed as a clause. On every call, P will be bound to a term representing the input problem, and as a result of the call Q will be bound to a term representing the output of the function application. The optimal ordering of subgoals is: divide problem P into independent subproblems P_1 and P_2; then solve P_1 and P_2 in parallel via the recursive calls, instantiating Q_1 and Q_2; when both are done, construct answer Q from partial answers Q_1 and Q_2. This sequence of events is implied by the picture in Figure 13c; exactly how it is achieved is described in the next section, on forward execution. This graph can be produced by repeated application of the connection rule, so mode declarations are not required. In general, however, mode declarations may be required when producing the ordering for functions.

Map Coloring

```
color(A,B,C,D,E) ←
        next(A,B) & next(C,D) & next(A,C) &
        next(A,D) & next(B,C) & next(B,E) &
        next(C,E) & next(D,E).

next(red,blue) ← .          next(red,green) ← .
next(blue,green) ← .        etc.
```

The goal of this procedure (Figure 13d) is to see if there is an assignment of one of four colors to the regions of the map, such that no two adjacent regions have the same color. The calls to *next* will succeed only if the arguments have been (or can be) instantiated to terms representing different colors. There is one call to *next* for each border in the map. This formulation of the map coloring problem as a logic program was originally given by Pereira and Porto in their papers on intelligent backtracking [47, 48].

When this procedure is called, none of the variables in the head will be instantiated. The literal ordering shown in the figure was produced by first using the leftmost rule to designate *next*(*A*,*B*) as the generator for both *A* and *B*, *i.e.* the solution of this literal assigns colors to regions *A* and *B*. The connection rule was used to identify the three literals in the middle row as generators of the other three variables. That leaves the remaining four literals as consumers. The role of consumer in this problem is too verify that colors assigned by generators are valid for the rest of the map.

Not unexpectedly, when this problem is interpreted, the generators in the middle row create a combination of values that is unacceptable to some of the consumers on the bottom row. There are a number of difficult problems presented by this example, as the AND process tries to coordinate the four generators in order to create, eventually, every five-tuple of colors that satisfy the constraints of this goal list. Many of the problems arise from the relative timing of the arrival of fail and success messages. The general principles will be explained in the section on backward execution. A detailed trace of the parallel solution of this problem is in Appendix II.

5.2. Forward Execution

Literals in the body of a parallel AND process will always be in one of three states: *blocked*, *pending*, or *solved*. A literal is in the solved state after an OR process has been created for it, and that process has sent back a success message. A literal is in the pending state when an OR process has been created for it, but the process has not yet sent back any message. Finally, a literal is in the blocked state when an OR process has not yet been created for it.

Forward execution is essentially a graph reduction procedure. Whenever the AND process receives a success message from a descendant, it means the corresponding literal can be resolved away from the body of the clause; in the dataflow graph, the node for the literal and all arcs leaving it are removed from the graph. The AND process succeeds after a success message has been received from every descendant, *i.e.* after the graph has been completely reduced. Recall that a success message from an OR process created to solve a literal L has the general form *success* $(L\theta)$, where $L\theta$ is a copy of L with (possibly) some variables bound. The graph reduction step is accomplished by resolving $\neg L\theta$ with the current set of literals in the body of the clause. If L is a generator of a set of variables, then some of those variables may be instantiated in $L\theta$. Envision values flowing from L to the consumers, as the resolution of $L\theta$ with the remaining literals causes those variables to be bound in the resolvent.

The criterion for deciding when to start an OR process for a literal is that a literal is ready to be solved only when all of its predecessors have been solved, *i.e.* when the corresponding node in the dataflow graph has no incoming arcs. If the graph is acyclic, and each literal can be solved, then eventually a process will be started for every literal. A more formal presentation of the forward execution algorithm is given in Figure 14. Figure 15 shows the parallel solution of two sample goal lists as sequences of graph reductions.

Figure 14 shows that the ordering algorithm will be applied after every success message is received. This is necessary for those cases when a generator binds its variable V to a non-ground term containing a new variable V'. If there is more than one consumer of V, they will then have a common variable in V'.

1. When the start message is received, initialize a list B to be the complete set of literals to be solved.
2. Repeat until B is the empty list:
   a. Apply the ordering algorithm to the literals in B to make a dataflow graph G.
   b. Start an OR process for every literal in G that has no incoming arcs and that does not already have a process.
   c. Wait for a message from an OR descendant.
   d. If the message is *fail*, call the backward execution algorithm (Section 5.3).
   e. If the message is *success*($L\theta$), resolve B with $\neg L\theta$, making a new body B {note: B now has one less literal, and bindings in θ have been applied to all remaining literals}.

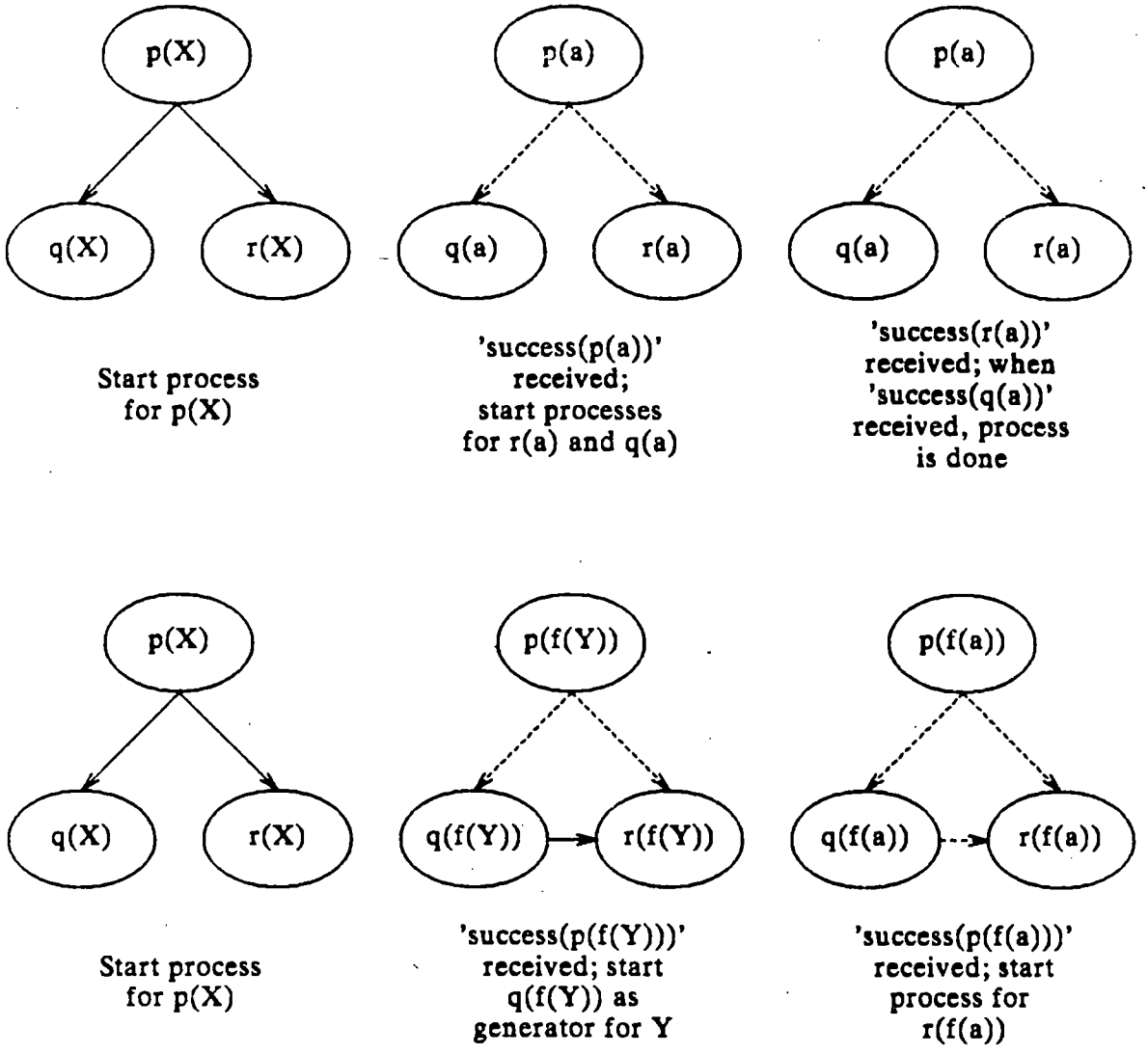
Figure 14. Forward Execution Algorithm

Since literals with variables in common are not solved in parallel, and since every variable must have a generator, the ordering algorithm must be called again to specify a list of list generator for V' . When the generator binds V to a ground term, which contains no variables, then step 2.a can be omitted. The current implementation of the interpreter makes the simplifying assumption that generators bind their variables to ground terms, and all subsequent discussions in this chapter will be based on this assumption.

The combination of the ordering algorithm and the forward execution strategy is sufficient for parallel solution of clauses that define deterministic functions. The distinguishing characteristics of these clauses are that the literals in the body are also deterministic functions, meaning they all have mode declarations, and for every combination of inputs there is just one output value for each output variable. Barring system failure, deterministic functions are guaranteed to succeed when given legal inputs.

Matrix multiplication is a good example of such a function. One way of writing this function as a logic program is shown in Figure 17. The head of the procedure is *mm*(*A*,*B*,*C*). When called, A and B will be bound to terms representing matrices, and after the call, C will be instantiated to their product. A row in a matrix is represented as a list of integers, and a matrix is a list of rows (*i.e.* a list of lists; see Figure 17).

The top level of the function is simply a call to transpose one argument, followed by a call to a procedure that actually multiplies the matrices. BT is the transposed version of B; it is a list of columns instead of a list of rows. After transpose succeeds, the problem is to distribute all possible pairs of rows of A with columns of BT to the inner product function. This is done by the two auxiliary functions *mmt* and *mmc*. The internal structure of these two procedures is identical: there are two literals in the body of each; one literal is a call to a lower level function with the first element of the input list, while the other literal is a recursive call with the remainder of the list. The dataflow graphs for both functions show that the literals are independent, and can be solved simultaneously. The inner product function shown here is sequential in nature, since the results of



This figure shows two possible sequences of graph reductions during forward execution for the goal statement

$\leftarrow p(X)\ \&\ q(X)\ \&\ r(X).$

Nodes and arcs drawn with dotted lines have been removed. In the first sequence, $p(X)$ generates the ground term a, and $r(a)$ and $q(a)$ can be solved in parallel. In the second sequence, X is bound to $f(Y)$, making the remaining literals $r(f(Y))$ and $q(f(Y))$. The literal ordering algorithm must be called to decide on a generator for Y; then that generator will be solved before the other literal.

Figure 15: Sequences of Graph Reductions

the multiplications are be summed serially.

Analysis of the bodies of *mmt* and *mmc* shows that since the recursive call can be done at the same time as the call to the lower level function, the time required to solve a problem of size n is proportional to the time required to solve the largest subproblem, rather than proportional to the sum of times to solve both subproblems. The time required to compute the product of the two matrices is thus the time required to distribute the last of the row/column pairs to process that performs an inner product, plus the time required to do that inner product. For the multiplication of $n \times n$ arrays, this time is $O(n+n+n)$, or $O(n)$ [27].

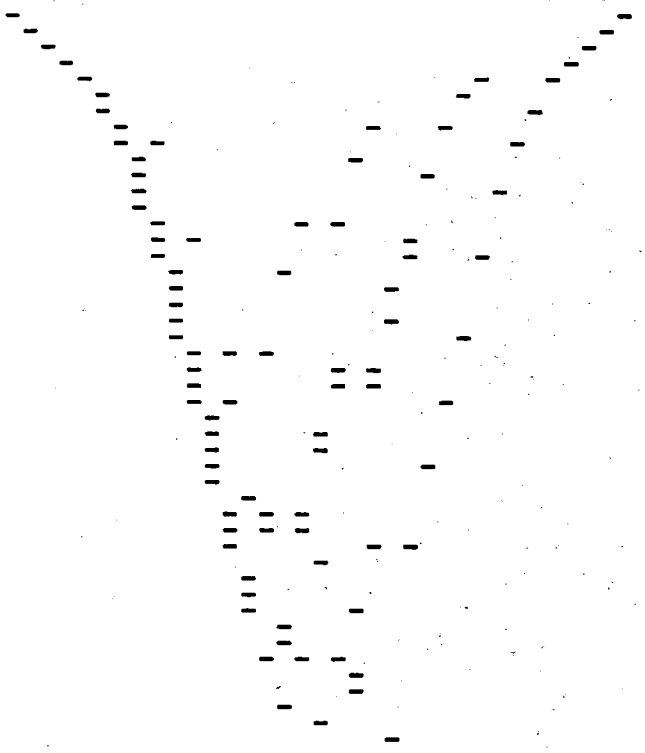
Figure 16 shows the time plot for the call to *mmt* in the multiplication of two 2×2 matrices. The corresponding table shows the number of steps required, and the simulated time used. The results support the claim that parallelism in deterministic functions can be exploited by the AND parallelism of the AND/OR Process Model.

5.3. Backward Execution

The purpose of backward execution is to coordinate the actions of the generators in their production of terms for the variables of the goal list. If there are n variables in its goal list, an AND process is expected to construct as many n-tuples of terms as possible. A subset of these n-tuples belong to the relation defined by the clause the AND process is interpreting.

A straightforward model for generating tuples is provided by the nested loops of a procedural language, such as Pascal. For example, a nested loop implementation of the map coloring problem of Section 5.1.3 is of the form

```
for A := Red to Blue do
    for B := Red to Blue do
        for C := Red to Blue do
            for D := Red to Blue do
                for E := Red to Blue do
                    if Next(A,B) and . . . and Next(D,E) then
                        Writeln('success(A,B,C,D,E)');
```



Maximum number of dashes per column = 5

46 processes executed 91 steps in 35 time units: 2.60

Interpreter Measurements:

N	Number of Processes	Number of Steps	Time	Steps/ Time	Number of Messages	Message Size
1	13	25	21	1.19	13	382
2	46	91	35	2.60	46	1630
3	121	241	49	4.91	121	4822

Figure 16. Plot for 2 × 2 Matrix Multiplication

```
/* To multiply two matrices, transpose the second, then form all inner products. */

mm(A,B,C) ← transpose(B,BT) & mmt(A,BT,C).

/* Multiply all rows of A with entire matrix B                                   */

mmt([],_,[]).
mmt([A1|An],B,[C1|Cn]) ← mmc(A1,B,C1) & mmt(An,B,Cn).

/* Multiply all columns of B with row A                                          */

mmc(_,[],[]).
mmc(A,[B1|Bn],[C1|Cn]) ← ip(A,B1,C1) & mmc(A,Bn,Cn).

/* Form the inner product of two vectors                                         */

ip([],[],0).
ip([A1|An],[B1|Bn],C) ← ip(An,Bn,X) & C is X+A1*B1.

/* To transpose a matrix, call 'columns' to divide it into two parts: the        */
/* first column and the rest of the columns; then transpose the rest.            */

transpose([[]|_],[]).
transpose(M,[C1|Cn]) ← columns(M,C1,Rest) & transpose(Rest,Cn).

columns([],[],[]).
columns([[C11|C1n]|C],[C11|X],[C1n|Y]) ← columns(C,X,Y).

/* Mode declarations, required for proper ordering                               */

mode(is,[?,+]).                    mode(mmt,[+,+,-]).
mode(mm,[+,+,-]).                  mode(mmc,[+,+,-]).
```

Figure 17. Matrix Multiplication Program

An abstract description of the working of this model is as follows. Initial values (*red*) are assigned to all variables. The initial tuple <*red,red,red,red,red*> is tested by the boolean expression in the body of the loop. The next tuple is created by assigning the innermost variable, *E*, its next value. Eventually, the last value (*blue*) is assigned to the innermost variable. The next tuple is obtained by *resetting* the variable *E* to its first value while making the second value of the next-innermost variable *D*. In general, whenever there are no more values for a variable, the previous (outer) variable is given a new value, and whenever a variable is set to a new value, all later variables (all those closer to the body of the loop) are reset to their initial values.

This simple model for generating tuples has been adapted for use in parallel AND processes as a way of coordinating the transmission of redo messages to descendant OR processes. It is not a very elegant model of tuple generation, but it has the twin virtues of being straightforward and complete, meaning it constructs all possible tuples as long as the domains of the generated variables are finite.

Nested generators can also be used to describe the overall behavior of the sequential Prolog interpretation. Since the same predicate that *tests* adjacent colors is also used to *generate* colors, the Prolog implementation has the advantage that it never constructs any obviously wrong tuples. In the Pascal implementation, all 5^4 5-tuples of colors are generated, the first 3^4 of which are of the form <*red,red,C,D,E*>. In Prolog, *next*(*A,B*) is the generator of *A* and *B*, and it never instantiates both *A* and *B* to the same color, thus effectively preventing the construction of a large number of useless tuples.

The parallel implementation to be described in this section retains the advantages of the Prolog interpretation, since it also uses the same predicate for generating and testing, and it has further efficiencies that are closely related to the intelligent backtracking of Pereira and Porto. To summarize, nested loops are inelegant but simple and correct, and when implemented in logic programs there is the potential for cutting out a large amount of useless work.

5.3.1. Data Structures for Backward Execution

Adoption of the nested loop model for constructing tuples of terms in a parallel AND process requires a *linear ordering* of literals and implementation of the *reset* operation. These items and data structures required for backward execution will be defined in this section; the actual sequence of events carried out in backward execution will be described in the next section. Examples will refer to the clause and dataflow graph of Figure 18.

Many of the data structures require a means for identifying a particular literal in the body of a clause. The technique used is to refer to a literal by a term of the form #N, where N is the place the literal occupies in the text of the clause. With respect to the example of Figure 18, the term #2 refers to *author*(*P*,*A*), the second literal in the body.

The linear ordering is actually an ordering of all literals, not just the consumers. The only constraint on the relative order of any two literals is that a generator must always come before all literals that consume its variable. In the current implementation, the linear ordering is obtained via a level-order traverse of the dataflow graph. The linear ordering of the literals of Figure 18 is [#1,#3,#2].

The reset operation must effectively restart a generator, so that a variable takes on the same set of values once again. The generator does not have to produce the values in the same order after a reset; the only requirement is that a variable is bound to all same values again. Also, a reset may occur before the generator has created all possible values.

Resets are implemented using lists of answers. The AND process maintains a list of used answers and unused answers from each generator. The normal sequence of events is that the answer in the first success message from a generator is put on the list of used answers. If the AND process needs a second answer from that process, it sends a redo, and the next answer is appended to the list of used answers. When a reset is called for, all answers but one from the used-list are copied to the unused-list. The remaining answer becomes the new current

Original clause:

$$paper(P,D,I) \leftarrow date(P,D)\ \&\ author(P,A)\ \&\ loc(A,I,D).$$

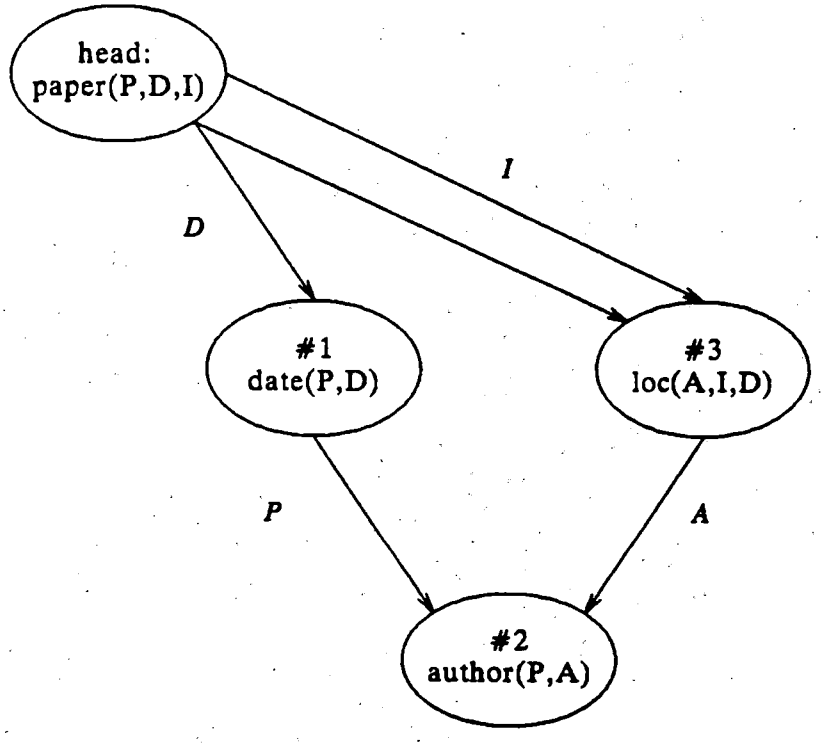
Goal:

$$\leftarrow paper(X,1978,uci).$$

Body of clause after head unified with goal (head generates D, I):

$$\leftarrow date(P,1978)\ \&\ author(P,A)\ \&\ loc(A,uci,1978).$$

Dataflow graph ('date' generates P, 'loc' generates A):



Linear ordering: [#1,#3,#2]

Redo lists:
[#1,head]
[#2,#3,#1,head]
[#3,head]
[head,#1,head]

Figure 18. Dataflow Graph for Example AND Process

value of the variable. As the AND process again requires additional answers, it takes them from the unused-list instead of sending a redo message to the OR process for the generator literal. Only when the unused-list is empty (all answers having been transferred to the used list) does the AND process send another redo message.

Backward execution also often requires cancel messages to be sent to descendant OR processes. After a parent sends a cancel message, it can ignore any subsequent messages received from the descendant. This situation may arise when a descendant sends a message, but the message has not yet been processed by the time the parent decides to send the cancel message. In the discussion below, *replacing* a process P means sending P a cancel message, creating a new process P' for the same literal, and using the process ID of P' in place of P.

When an AND process receives a fail message from one of its OR descendants, one of the generators that precedes the failed literal L must produce a new value. If that generator cannot generate a new value (*i.e.* it returns a fail message in response to the redo message), one of the other generators that produces values consumed by L must be sent a redo. The decision of which generators will be sent redo messages, and in what order, is determined by a ***redo list*** associated with each literal. The redo list for a literal L contains L and every predecessor of L, sorted according to the linear order (with literals that are earlier in the linear order occurring later in the redo list). Redo lists are created at the same time the linear ordering is made. Redo lists for the literals of the example problem are shown in Figure 18.

Finally, an AND process maintains a list called the ***failure context*** to keep track of the failed literals and decide exactly which generator should be sent the next redo message. The failure context is initially the empty list, and as fail messages are received, literal numbers are added to this list.

5.3.2. Processing of Backward Execution

The backward execution algorithm will be introduced in this section. The algorithm is quite complicated. Instead of discussing the complete algorithm, which contains provisions for handling a number of special cases, only the subset of rules sufficient for obtaining the first solution of the example problem of Figure 18 will be presented first. After the processing for this solution has been described, some further details will be given. The complete set of rules for backward execution, and examples of some special cases, can be be found in the Appendices.

An overview of backward execution is that when a fail message is received, the backward execution algorithm is called to trace out a path in the dataflow graph that extends back from the failed literal. This path should eventually include every predecessor of the failed literal, if required. The failure context reflects the current state of this path. When a generator is encountered on this path, it is sent a redo message, and then every generator occurring later in the linear ordering is reset. If the AND process tries to extend the path beyond a literal with no predecessors, or to include the head of the clause in the path, it fails.

The desired backward path is simply the redo list for the failed literal. This list contains every predecessor of the literal, *i.e.* it contains every generator that could possibly effect the set of values consumed by the literal. An AND process will always be able to determine which generator to re-solve when it first receives a fail message. However, once the backward execution algorithm has embarked on a backward path, subsequent failures of literals not on this path can cause difficulties. This is known as a *multiple failure*; rules for handling multiple failures are described later; examples are found in the solution of the map coloring problem in Appendix II.

When a fail message is received, the AND process appends the literal number of the failed literal to the failure context list. Then the AND process searches for a redo list R such that the new failure context is a prefix of R. The first literal in the resulting suffix identifies which predecessor of the failed literal

should be sent a redo message.[1]

Referring to the figure, if the process for #2 sends a fail message, the failure context is set to list [#2]. [#2] is a prefix of the redo list [#2,#3,#1], and the suffix after this match is [#3,#1], so the OR process for literal #3 is sent a redo message. Next, if the process for #3 fails (meaning that generator cannot produce any more values), the failure context becomes [#2,#3], the suffix after the match is [#1], and the process for literal #1 will be sent a redo message.

Whenever a generator is sent a redo message, the corresponding literal is moved from the solved state to the pending state, since the process for the generator is again trying to construct an answer. When any generator is sent a redo message, a number of other literals will be effected. First, all generators that are later in the linear ordering are reset; this is the step that correlates most directly with the nested loop model. Generators that are reset after being solved are still considered to be solved, since there will be one answer that can be used as the current value of the variables generated. Second, some consumer processes will have to be canceled. If a literal consumes a variable that is generated by any generator that was either sent a redo or reset, then that literal must be canceled, since it consumes values that are being changed. The processes for these literals will be replaced when all of their predecessors are once again in the solved state. Note that a literal might have a new process created immediately, in the case that all of its generators were simply reset.

The processing of success messages has to be modified slightly, in order to accommodate the failure context. When a success message is received, and the forward execution algorithm starts a set of new processes, the literals corresponding to the new processes must be removed from the failure context list. Thus the failure context list grows and shrinks as generators are sent redo messages and then respond with additional answers. The failure context shrinks all the way

---

[1] This operation is concisely expressed in Prolog with the concat procedure: concat(A,B,C) means that C is the concatenation of lists A and B, or, equivalently, that A is a prefix and B a suffix of the list C. If F is the current failure context, and R is the redo list, then concat(F,[X|Y],R) asks "is F a prefix of R? If so, unify X with the first element of the list that must be concatenated with F to make R."

back to the empty list when a new process is started for the literal that originally failed. A more concise presentation of the backward execution algorithm is in Figure 19.

5.4. Example

In the example execution plotted in Figure 9 (Chapter 3), process 4 was created to solve the body of the clause

paper(*P*,1978,*uci*) ← *date*(*P*,1978) & *author*(*P*,*A*) & *loc*(*A*,*uci*,1978).

The steps in the interpretation of that process, from the original ordering of literals through the generation of the first success message, will be explained in detail in this section. The explanation refers to the detailed trace of the individual state transitions generated by the interpreter shown in Figure 20 at the end of the chapter.

5.4.1. Ordering

The clause used to define the initial state of process 4 is

paper(*P*,*D*,*I*) ← *date*(*P*,*D*) & *author*(*P*,*A*) & *loc*(*A*,*I*,*D*).

When the process is created, variables *D* and *I* are bound to 1978 and *uci*, respectively. The ordering algorithm does the following for this clause:

- There are no mode declarations, so step (1) has no effect.
- The variables generated by the head are { *D*, *I* }. The variables that do not have generators are { *P*, *A* }.
- The first pass through the list of literals finds two literals that are connected to { *D*, *I* }. *date*(*P*,*D*) contains *P*, a variable with no generator yet, and *D*, a variable generated by the head, so it is designated as the generator of *P*. Similarly, *loc*(*A*,*I*,*D*) becomes the generator of *A*. At this point all variables have generators (*U* is the empty list), so the ordering algorithm terminates.

  The dataflow graph for this clause is in Figure 18.

1. When a fail message from process ↑L is received, change the state of #L from pending to blocked {note: the process that was created for #L failed and no longer exists; L is now blocked until a predecessor is re-solved}
2. Append #L to the failure context.
3. Unify the updated failure context with a prefix of one of the redo lists. The failure context is of the form [#F1,...#L], and the matched redo list is of the form [F1,...#L|X].
4. The unification of the previous step may succeed when the failure context is exactly the same list as one of the redo lists, *i.e.* X is the empty list. If this is the case, the AND process fails.
5. If the list X from step (3) is not empty, it must be of the form [#G|Xn]. #G is the generator that is to be redone. If #G is the head of the clause, the AND process fails. Otherwise, send ↑G a redo message, and change the status #G from solved to pending.
6. Whenever the OR process for a literal #G is sent a redo message, the AND process may have to reset or cancel some literals to the right of #G in the linear ordering:
   a. For every generator later than #G in the linear ordering, perform a reset operation. These generators remain in the solved state, since their consumers can immediately (re)use the first value. The variables generated by these generators and the variables generated by #G are called the modified variables.
   b. For every literal #L later than #G in the linear ordering, cancel ↑L and change the state of #L to blocked if it consumes any modified variable. It does not matter if #L is a generator or not or if it was previously solved or pending; if it consumes a modified variable, its process ↑L must be canceled.
7. It is possible that some of the OR processes canceled in the previous step can be replaced immediately (since if the variables they consume were reset, the corresponding generators are still in the solved state), so the forward execution algorithm is invoked to start a set of new OR processes.
8. When a new process is started for a literal #N that is currently in the failure context, remove #N and any literals to the right of it from the failure context {note: this step must be taken by the forward execution algorithm as well, after a success message causes a set of new processes to be created}.

Figure 19. The Backward Execution Algorithm

5.4.2. Forward Execution

After the ordering algorithm has been applied, the data structures in the AND process are initialized:

-- The only literal in the solved state is *head*; all others are blocked.

-- The linear ordering, obtained by a level order traverse of the dataflow graph, is [#1,#3,#2].

-- The failure context list is set to the empty list.

-- The redo lists are created for each literal, including the head:

#1: [#1,*head*]
#2: [#2,#3,#1,*head*]
#3: [#3,*head*]
head: [*head*,#1,*head*]

The first step in the forward execution phase is shown in transition <1>. An OR process can be started for a literal when the predecessors of that literal have been solved.[2] This test succeeds for #1 and #3, so OR processes are created to solve the literals *date*(*P*,1978) and *loc*(*A*,*uci*,1978). The IDs of these OR processes are 6 and 7, respectively.

Transition <2> occurs when *success*(*date*(*prolog*,1978)) arrives from process 6. This is from the process created to solve literal #1, so #1 is added to the solved list. Since #3 is also a predecessor of #2, and #3 is not yet solved, no new processes are created on this step.

The next transition occurs when *success*(*loc*(*kling*,*uci*,1978)) arrives from ↑3 (process 7). #3 is added to the solved list, and now a process (number 9) is created for #2. Note that after bindings from the first two answers have been applied, #2 is *author*(*prolog*,*kling*).

---

[2] As currently implemented, the redo list of a literal doubles as the set of predecessors of the literal. The redo list for literal N is always of the form [#N|Rem], and the operation of determining whether all predecessors of N have been solved is equivalent to seeing if Rem is a subset of the list of solved literals.

5.4.3. Backward Execution

The linear ordering shows that the generator of *A* comes after the generator of *P*. Throughout the backward execution phase, then, the expected behavior will correspond to the nested loops

for P := FirstPaper(1978) **to** LastPaper(1978) **do**
  for A := FirstAuthor(UCI) **to** LastAuthor(UCI) **do**
    if Author(P,A) **then** . . .

Transition <4> starts the backward execution processing. ↑2 fails, so #2 is appended to the failure context, making [#2]. The redo sequence which has [#2] as a prefix is [#2,#3,#1,head], so ↑3 is sent a redo message. #3 is removed from the list of solved literals (since it is now working on a second answer). The current state of the process is: literal #1 solved, #2 blocked, #3 pending, with one used answer from both ↑1 and ↑3.

Transition <5> is triggered by a fail from ↑3, meaning there is no additional binding for *A* that satisfies *loc*(*A*,*uci*,1978). #3 is appended to the failure context, making [#2,#3]. This new list is matched with [#2,#3,#1,head], meaning #1 is sent a redo. Note that #3 is reset here: it is later than #1 in the linear ordering, and does not consume *P*. The effect of the reset is to bind *A* to *kling*. The current state is now: #1 pending, #2 blocked, #3 solved, with (still) one used answer from both generators.

Transition <6> takes place when a success message arrives from ↑1 with the second binding for *P*. #1 is added to the list of solved literals (making that list [#1,#3,head]), and a new process can be created for #2 (which is now *author*(*eft*,*kling*)). Note the effect of the success of ↑1 on the failure context: when the new process for #2 was started, it and everything to the right of it were removed from the failure context, changing that list from [#2,#3] to [].

When ↑2 sends success (transition <7>), all literals have been solved. The message

success ([*date*(*eft*,1978), *author*(*eft*,*kling*), *loc*(*kling*,1978,*uci*)])

is sent to the parent.

5.5. Handling Redo Messages

A parallel AND process handles a redo from its parent in the same way it handles a fail from one of its descendants. The "failed literal" is the head, which is the consumer of any variables that were not instantiated when the clause was called. The redo message is used to start a failure context, and redo messages are sent to descendants until a new tuple is created.

It is possible to make a redo list for the head, just like any other literal: a list is constructed that contains all predecessors of the head when the head is considered to be a consumer. When the redo message is received, the failure context is set to [head], and then a generator is selected as before, using this head redo list.

The head redo list for the example of Figure 18 is [head,#1,head]. This starts with the identifier of the literal itself (recall that the redo list for a literal #N of the body starts [#N,...]), and then has literal #1, the generator of P, since the head of this clause consumes P. The second occurrence of ***head*** in this list means that the head is a predecessor of #1, and is included for the cases when the AND process should fail because the failure path started by a redo message is traced all the way back to the head of the clause.

The head literal is removed from the failure context under exactly the same conditions other literals are removed, namely when all predecessors of the head are solved.

To summarize, a redo message will start the AND process in backward execution, tracing a path back through the dataflow graph. After some number of redo messages have been sent to descendants, a new tuple will be created. On the resumption of forward execution, the AND process will reach a a state where the head (in its role as a consumer) should be removed from the failure context, while other literals test the current tuple of values.

Returning to the state transitions in Figure 20 at the end of the chapter, all transitions after <7> show how the AND process responds to a redo message. Transition <8> is triggered when the parent (process 2) sends the first redo

message, requesting the second tuple. The failure context is set to [head], and the head redo sequence is [head,#1,head], meaning #1 should be sent a redo. As a result of sending that redo message, ↑1 is set to work constructing the third value for P, #3 is reset and #2 is canceled. The state of the process is: #1 pending, #2 blocked, #3 solved, with one used answer from ↑3 and two used answers from ↑1.

Transition <9> is triggered by the success message from ↑1 containing the third value of P. #1 is once again added to the solved list, and a new process for #2 is started. Note also that the solved list is [#1,#3,head] and the tail of head redo list is a subset of the solved list, so *head* is removed from the failure context. The current state of the process is: #1 and #3 solved, #2 pending, and an empty failure context.

The third value of P cannot be used to solve #2, so the latest process for that literal sends a fail message (transition <10>). The failure context becomes [#2], and backward execution resumes. Normally, ↑3 would be sent a redo message. However, the AND process knows that process has already failed (since the ↑P field shown in the figure is 0), so #3 is immediately added to the failure context, and the redo message is sent to ↑1. The current state is: #1 pending, #2 blocked, #3 solved (since even though ↑3 has failed, there are answers that are used via resets), with ↑1 working on a fourth value for P.

Transitions <11> and <12> are basically the same as <9> and <10>, since the fourth value of P also causes a process for #2 to fail.

Finally, a fail message arrives from ↑1, indicating that there are no more ways to solve *date*(P,1978). When ↑1 fails, the fail list will become [#2,#3,#1], a prefix of [#2,#3,#1,head], indicating that *head* is the literal to be sent a redo, so the AND process fails.

5.6. Discussion

Returning to the example of Figure 18, consider what happens if literal #3 is the *first* literal to fail. The state of the AND process at this time would be: literals #1 and #3 pending, and literal #2 blocked. When the fail message arrives

from ↑3, the failure context becomes [#3], which is the prefix of the redo list [#3,head]. The suffix is [head], meaning the head of the clause was the generator that created the value that caused #3 to fail. This is a case where the AND process fails. In the previous example, when #3 failed the failure context was the list [#2], and the AND process responded by sending the process for literal #1 a redo message. The reason for the different responses (fail message to the parent versus redo message to ↑2) to the same message (fail from ↑3) lies in the interpretation of the failure context. In one case, when the failure context is empty, #3 failed on its own, *i.e.* it failed because it could not generate any values for the variable A. In the other case, when the failure context is [#2], #3 failed because it could not generate any *additional* answers for one of its consumers, and since that consumer had other predecessors, the AND process needed to send a redo to one of those predecessors and reset #3.

An interesting question arises when considering how to process redo messages. As described above, in the current implementation, redo messages are handled by starting a failure context for the head of the clause, then sending a redo message to the generator for one of the uninstantiated variables contained in the head. Alternatives are to send a redo message to the last generator in the linear ordering, or maybe send a redo to the process for the last literal in the linear ordering, whether it is a generator or not. The difference between either of these alternatives and the method implemented is that the latter is based on the fact that the AND process is expected to generate a set of tuples, and not a multiset; *i.e.* a redo message is a signal to create a ***new*** tuple, different from any previous answer. Sending a redo to a generator of a variable in the head of the clause makes sure the next tuple will have at least one different value.

The following is an example of a program which, when interpreted by Prolog, produces a multiset of tuples for the denotation of the procedure *p*:

$p(A) \leftarrow q(A)\ \&\ r(B).$
$q(0) \leftarrow .$
$r(1) \leftarrow .$
$r(2) \leftarrow .$

The first answer to the query

$\leftarrow p(A).$

is found by unifying $q(A)$ with the unit clause $q(0)$, and then unifying $r(B)$ with $r(1)$. The first answer is thus $p(0)$. If told to backtrack, Prolog will re-solve $r(B)$, this time by binding B to 2, and it once again reports success for $p(0)$. The operational semantics for p is the multiset {<0>,<0>}, in which one tuple occurs twice. A parallel AND process would also produce multiple instances of the same answer if the rule for handling redo messages were to send a redo to either the last literal or the last generator in the linear ordering. However, the rule is to send a redo to a generator of a head variable. For this example, it means sending a redo message to the process for $q(A)$, and thus the second proof of $p(0)$ is not performed.

The reasoning behind the choice to have AND processes create sets instead of multisets is that a set of tuples is closer to the spirit of the definition of the semantics. Interpreters that create multisets do so merely as a side effect of the operational semantics. In short, just because there is more than one way to prove that a tuple belongs in D is no reason to include more than one copy of that tuple in D.

As mentioned previously, multiple failures are quite difficult to process. At first, it would seem reasonable to maintain a number of failure context lists, each beginning with the ID of a failed literal. However, this leads to a number of difficult questions:

- If one failure context determines that a generator is to be reset during some state transition, and then a different failure context decides that the same generator should be reset during a later state transition, then should the generator be reset twice? Or is one reset sufficient?
- What should happen when two failure contexts reach a common predecessor? The generator at that node will be sent a redo when the first failure context includes that node, but the generator should probably not be sent another redo message when it is added to the second failure context.

The current method for handling multiple failures is to *postpone* the processing of additional failures while in the midst of tracing out a failure path for the first literal to fail. When a fail message arrives from a process ↑N such that #N cannot be added to the current failure context,[3] then #N is considered to be a *postponed failure*, and is ignored temporarily. When the failure context shrinks to the empty list, the AND process will start a new failure context for one of the postponed failures, as long as the process for the postponed failure was not canceled during the processing of a previous failure. A deeper discussion of postponed failures, and an example of the processing of a postponed failure, can be found in Appendix II.

In the preceding discussions of resets and redo messages, it may appear that there is an underlying assumption that each generator is responsible for generating values for only one variable, which is the case in the procedural language implementation of nested loops. However, no such assumption needs to be made in adopting this model for parallel AND processes. If a literal $g(X,Y)$ is the generator for both X and Y, it is not necessary to assume that this generator creates all possible Y values before generating a second value for X and resetting Y. The AND process simply uses the $<X,Y>$ pairs returned by the OR process for $g(X,Y)$, and lets the OR process worry about creating all possible pairs. Whenever the process for $g(X,Y)$ is sent a redo message or reset, all consumers of either variable are canceled.

A goal for backward execution is to be able to generate as many tuples of values as possible. When the domains of the variables are finite, the nested loop model completely specifies the set of tuples. When one or more domains are infinite, then nested loops are not able to generate all tuples. In particular, consider two variables, I and F, where the domain of I is infinite, the domain of F is finite, and the values of the variables are denoted $\{i_1, i_2, \cdots\}$ and $\{f_1, f_2, \cdots f_m\}$. In the goal list

---

[3] In other words, the call to concat specified in a previous footnote fails for all redo lists.

$\leftarrow gf(F)$ & $gi(I)$ & $pf(F)$ & $pi(I)$.

gf is the generator of F, gi the generator of I, and I is the innermost variable. Suppose pf succeeds only for the second value of F, but pi succeeds for any value of I. Since no tuple with f_1 can succeed (*i.e.* $pf(f_1)$ fails), then Prolog (and the equivalent nested loop program in a procedural language) will never solve this goal list. All tuples created will be of the form $<f_1,i_n>$, with the interpreter stuck in an infinite loop generating the i_n. Parallel AND processes have a chance of succeeding in this example, since when $pf(F)$ fails a redo message is sent to the generator for F, while the generator for I is reset. Thus there are cases, even when infinite domains are involved, where parallel AND processes construct all successful tuples. Parallel AND processes are still not perfect, however. If the consumer is $p(F,I)$, the linear ordering may specify that the generator for I is to be sent a redo message before the generator of F, and thus the parallel AND process is also caught in an infinite loop.

The intelligent backtracking interpreter of Pereira and Porto can also avoid infinite loops, since the generators of infinite domains may be skipped on backtracking. Their interpreter may even succeed when parallel AND processes fail, because their interpreter analyzes the *cause* of a failure. If $p(f_1,i_1)$ fails because the unification of $p(f_1,i_1)$ does not succeed when F is bound to f_1, their interpreter knows to backtrack into the solution of the generator of F. At present, all a parallel AND process knows is that $p(f_1,i_1)$ failed, that this literal has two predecessors, and that one of them must be redone; which one is determined solely by the linear ordering.

5.7. Chapter Summary

Parallel solution of the body of a clause is essentially an attempt to create a dataflow graph from the body, and then solve the literals in the order specified by the graph. This attempt is successful when the literals all succeed, which is often the case when the clause implements a deterministic function. However, in nondeterministic functions and relations, it is not always the case that literals can be solved on the first attempt. When a literal fails, an interpreter must re-

solve a previously solved literal, hoping the next solution creates new variable bindings that allow the failed literal to be solved.

Backward execution is the name of the mechanism in parallel AND processes that determines which literals must be re-solved in response to failures. The mechanism is quite complicated, and requires a large overhead in terms of data structures to represent the state of each literal and the state of the process as a whole. Fortunately, the overhead does not interfere with forward execution; it is only when literals fail that the rather awkward backward execution mechanism is invoked.

There are a number of improvements that can be made in the definition and implementation of backward execution. Many descendant processes may be canceled needlessly, sequential processing of multiple failures is very conservative, and the nested loop model itself may not be the best abstract model of tuple generation. The philosophy has been to define a method that is sufficient to coordinate the literals that bind variables to values, so that eventually as many tuples of values are created as possible. The long term goal of the research is the design of a non von Neumann computer architecture for parallel execution of logic programs. Rather than spending time in fine tuning the backward execution mechanism, it is time to move on to the next step, and show how the parallel processes may be efficiently implemented on a non von Neumann system.

Key to State Information of Parallel AND Process

In the following trace, all lines except the descriptions of the current status of the literals are self-explanatory. For each literal, the current status is indicated by a data structure of the form

#N:[Curr,Orig,RL,Gens,MP,U,UU]

where *N* is the literal number, *Curr* is the current form of the literal (including variable bindings, if any), *Orig* is the original form of the literal (before any bindings), *RL* is the redo list, *Gens* is the list of variables generated by the literal, *MP* is the ID of the OR process created to solve the literal (if 0, there is no such process, *i.e.* the literal is in the blocked state), and *U* and *UU* are lists of used and unused answers from *MP*. The AND process does not attempt to keep the list of literals in any particular order. In fact, literals toward the top of the list are literals that were most recently used for some operation.

Variable names beginning with capital letters are "real" variables, which may eventually be bound. Variables of the form $var(N) are metavariables, required by the ordering algorithm and other procedures that reason about variables without binding them. In this process, the variables are *P, D, I,* and *A*; the corresponding metavariables are $var(0), $var(1), $var(2), and $var(3).

<1>

```
AND Process 4 after 'start' from Process 2, T = 2
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [head]
  Failure Context: []
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #1:[date(P,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6, [], [] ]
    #3:[loc(A,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 7, [], [] ]
    #2:[author(P,A), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 0, [], [] ]
```

Figure 20. States of a Parallel AND Process

<2>

```
AND Process 4 after 'success(date(pro,1978))' from Process 6, T = 4
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#1,head]
  Failure Context: []
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #1:[date(pro,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6, [date(pro,1978)], [] ]
    #3:[loc(A,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 7, [], [] ]
    #2:[author(pro,A), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 0, [], [] ]
```

<3>

```
AND Process 4 after 'success(loc(kling,uci,1978))' from Process 7, T = 4
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#3,#1,head]
  Failure Context: []
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #2:[author(pro,kling), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 9, [], [] ]
    #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 7, [loc(kling,uci,1978)], [] ]
    #1:[date(pro,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6, [date(pro,1978)], [] ]
```

Figure 20 Continued

<4>

```
AND Process 4 after 'fail' from Process 9, T = 7
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#1,head]
  Failure Context: [#2]
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #2:[author(pro,A), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 0, [], [] ]
    #3:[loc(A,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 7, [loc(kling,uci,1978)], [] ]
    #1:[date(pro,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6, [date(pro,1978)], [] ]
```

<5>

```
AND Process 4 after 'fail' from Process 7, T = 9
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#3,head]
  Failure Context: [#2,#3]
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 0, [loc(kling,uci,1978)], [] ]
    #2:[author(P,kling), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 0, [], [] ]
    #1:[date(P,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6, [date(pro,1978)], [] ]
```

Figure 20 Continued

<6>

```
AND Process 4 after 'success(date(eft,1978))' from Process 6, T = 11
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#1,#3,head]
  Failure Context: []
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #2:[author(eft,kling), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 11, [], [] ]
    #1:[date(eft,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6,
          [date(eft,1978), date(pro,1978)], [] ]
    #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 0, [loc(kling,uci,1978)], [] ]
```

<7>

```
AND Process 4 after 'success(author(eft,kling))' from Process 11, T = 13
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#2,#1,#3,head]
  Failure Context: []
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #2:[author(eft,kling), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 11, [author(eft,kling)], [] ]
    #1:[date(eft,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6,
          [date(eft,1978), date(pro,1978)], [] ]
    #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 0, [loc(kling,uci,1978)], [] ]
```

Figure 20 Continued

<8>

```
AND Process 4 after 'redo' from Process 2, T = 15
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#3,head]
  Failure Context: [head]
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 0, [loc(kling,uci,1978)], [] ]
    #2:[author(P,kling), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 0, [], [] ]
    #1:[date(P,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6,
          [date(eft,1978), date(pro,1978)], [] ]
```

<9>

```
AND Process 4 after 'success(date(df,1978))' from Process 6, T = 17
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#1,#3,head]
  Failure Context: []
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #2:[author(df,kling), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 13, [], [] ]
    #1:[date(df,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6,
          [date(df,1978), date(eft,1978), date(pro,1978)], [] ]
    #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 0, [loc(kling,uci,1978)], [] ]
```

Figure 20 Continued

<10>

```
AND Process 4 after 'fail' from Process 13, T = 19
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#3,head]
  Failure Context: [#2,#3]
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 0, [loc(kling,uci,1978)], [] ]
    #2:[author(P,kling), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 0, [], [] ]
    #1:[date(P,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6,
          [date(df,1978), date(eft,1978), date(pro,1978)], [] ]
```

<11>

```
AND Process 4 after 'success(date(fp,1978))' from Process 6, T = 21
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#1,#3,head]
  Failure Context: []
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
    #2:[author(fp,kling), author($var(0),$var(3)),
          [#2,#3,#1,head], [], 15, [], [] ]
    #1:[date(fp,1978), date($var(0),$var(1)),
          [#1,head], [$var(0)], 6,
          [date(fp,1978), date(df,1978),
           date(eft,1978), date(pro,1978)], [] ]
    #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
          [#3,head], [$var(3)], 0, [loc(kling,uci,1978)], [] ]
```

Figure 20 Continued

<12>

```
AND Process 4 after 'fail' from Process 15, T = 23
  Parent ID:      2
  Linear Ordering: [#1,#3,#2]
  Head Redo Seq:   [head,#1,head]
  Literals Solved: [#3,head]
  Failure Context: [#2,#3]
  Literal Status -- #N:[Curr,Orig,RL,Gens,MP,U,UU]:
     #3:[loc(kling,uci,1978), loc($var(3),$var(2),$var(1)),
           [#3,head], [$var(3)], 0, [loc(kling,uci,1978)], [] ]
     #2:[author(P,kling), author($var(0),$var(3)),
           [#2,#3,#1,head], [], 0, [], [] ]
     #1:[date(P,1978), date($var(0),$var(1)),
           [#1,head], [$var(0)], 6,
           [date(fp,1978), date(df,1978),
            date(eft,1978), date(pro,1978)], [] ]
```

<13>

```
AND Process 4 after 'fail' from Process 6, T = 25
done
```

Figure 20 Continued

CHAPTER 6

Multiprocessor Implementation of AND/OR Processes

The overall goal of this dissertation research is the design of a multiprocessor computer architecture that exploits parallelism in the execution of logic programs. The research takes the language first approach, summarized in the introduction, in which the designer starts with an abstract model of computation, specifies a programming language based on the model, defines a method for interpreting programs of the language in parallel, and finally starts the design of a computer that supports the parallel execution model. Previous chapters described the research in the early steps of this top down process, research which culminated in the definition of a method for interpreting logic programs that automatically divides the program into independent pieces for parallel solution. This chapter presents some implementation considerations, with the goal of showing that the abstract model can form the theoretical framework for a practical multiprocessor.

At this level, a logic program appears to the system as a collection of independent processes that communicate via messages. When a process receives a message, it will be transformed into another state, and possibly generate messages for other processes. Using operating system terminology, a process is *running* when a PE is transforming it from one state to another, it is *blocked* when there are no messages for it, and it is *ready* when there is a message for it, but no PE is (yet) transforming it into its next state.

The transformation of a process from one state to another will be considered an indivisible operation. That is, once a PE is set to work on a particular transformation, it will complete the transformation before taking on any other tasks. Another description of the operation of the model at this level is the provided by the concept of a *workpool*. At any point in time, the system has a pool of work that needs to be done. There is a network of PEs, each of which has

access to items in the pool. The system operates by having PEs take a piece of work from the pool, apply some operation, and put the updated piece of work back in the pool. In the AND/OR Process Model, the pool of work is defined by the sets of processes and messages. A piece of work is defined to be a process and a single message destined for it, and the operations performed by the PEs are the state transformations defined by the process/message pairs.

There have been a number of architectures defined around the workpool concept, so naturally they are candidates for architectures to support the AND/OR Process Model [56]. A basic feature of these architectures is that PEs either share a common memory, or fetch work from the memory over a shared bus. For a variety of reasons to be explored below, at this time these architectures do not appear to be optimal for the AND/OR Process Model.

This chapter starts with a survey of issues involved in distributing packets of work to the PEs of a system, and concludes with some arguments for why the AND/OR Process Model should be implemented on a network of independent processors, each with its own substantial amount of local memory, large enought to keep a complete copy of the logic program. The local memory must also store processes the PE is working on, and messages bound for those processes. The topology of the processor interconnection network (*e.g.* R-ary N-cube, ring, bus) is not important for this discussion.

6.1. Issues

Ideally, as soon as a process goes into the ready state, some PE will be assigned the task of performing the corresponding state transition. The mechanism that decides which process executes on which PE is important. Dimensions for comparing process allocation scheme are decentralization, locality, and evenness. Decentralization means that there should not be a central PE or authority that decides where a process should be executed. A centralized mechanism is a bottleneck when there are a very larger number of processes, and is a vulnerable point in terms of system reliability. Locality means that processes that communicate should be close to each other physically, so that messages from one to the

other will not have to travel very far in the network, no matter what the topology is. Finally, evenness refers to the goal that all PEs should have the same amount of work to do at all times.

In the AND/OR Process Model, a large number of processes, and an equally large number of messages, will be created during the solution of parallel problems. When multiplying two square matrices of size n, $O(n^3)$ processes and $O(n^3)$ messages are created (Figure 16). In any finite system, a point will be reached where there are more processes and/or messages than can be stored in the system as a whole or on a single PE. The underlying hardware will become bogged down, or may even deadlock. Analogous situations occur in conventional systems, where data blocks are moved between main memory and disk, and the system becomes bogged down by thrashing, and in dataflow systems, where the matching store overflows when there are too many tokens waiting for partners [54].

Three ways to address this problem are to try to minimize the overhead, in terms of storage requirements for processes and messages; design a method for gracefully moving blocked processes and dormant messages between main memory and secondary storage; and use mechanisms for inhibiting parallelism, so that when the system starts to become overloaded, it should switch to a mode where fewer processes are created [44]. Specific comments about all three of these methods for managing large problems in the context of the AND/OR Process Model will be discussed in the next sections.

6.2. Implementation

6.2.1. Process Migration

Burton and Sleep have proposed a method of distributing processes to PEs whereby it is up to an idle PE to take the initiative to find work to do [8]. There is no notion of *assigning* a newly formed process to a PE. Rather, a PE assumes that it must completely solve any problem that is given to it, and later, if a neighboring PE indicates that it is idle, then a part of the current problem may be sent to the neighbor. By way of contrast, in many multiprocessor systems

there is a mechanism for assigning tasks to processors that will perform the task. An example is the assignment function of the Irvine Dataflow system, which is a hashing function that maps activity names into processor IDs [27].

This distribution model is, obviously, decentralized. Locality is enforced by having a process move at most once, from the PE that created it to one of its immediate neighbors.

An even distribution of work under this distribution plan depends on strategies for deciding which parts of a problem to keep and which parts to let go. As an example of how AND/OR processes could be evenly distributed, consider an AND process for solving a typical problem involving tail recursion, where the Xs are bound and the Ys are unbound in a call to p:

$$p([X1|Xn],[Y1|Yn]) \leftarrow q(X1, Y1) \ \& \ p(Xn, Yn).$$

In a parallel AND process, both goals on the right hand side are solved in parallel, and thus two OR processes are created and sent start messages at the same time. The PE solving the AND process could keep the OR process that solves $q(X1, Y1)$, and let the OR process for $p(Xn, Yn)$ migrate to a neighbor. Xn is a list of terms; usually each term in this list has the same general structure as $X1$. Using this policy of sending the "tail" part of the tail recursion, the problem could unfold along a "line" of PEs, each of which would solve one of the goals $q(Xi, Yi)$, and the work would be apportioned evenly. Since the only message passing in the system is between parent/descendant pairs, the locality of message transfers is not effected by this policy.

There is a tradeoff here, since in these problems the term Xn is a *list* of terms, each of the form $X1$. Thus the goal $p(Xn, Yn)$ is likely to be larger (in terms of the number of bytes required to write it) than the goal $q(X1, Y1)$. The tradeoff is that in order to spread the work evenly, the larger goals must be passed from one PE to the next.

Another potential difficulty is related to the topology of the underlying network. The above scenario of unfolding a "line" of work is very likely when the topology is a ring, in which PE_i has only two neighbors, PE_{i-1} and PE_{i+1}.

However, in a hypercube, it is not clear what path from PE_1 to PE_n corresponds to a "line", or how the paths from the PEs would interact. Defining a set of policies that help each PE decide which processes to send to its neighbors, and analyzing the tradeoffs involved in the context of various topologies, is the subject of future work and simulations.

The biggest advantage to this scheme for allocating work is that when large problems are being solved, the system will reach a point where (assuming a relatively even distribution plan has been developed) each PE will be actively working on a problem. Since each PE is busy, no requests for work will be transmitted, and no more subproblems will be passed around the network. Thus one immediate advantage is that message traffic is not strictly a function of problem size. When a large problem is solved, there will be a large amount of traffic as subproblems are initially spread around, but eventually a point will be reached where each PE is busy working on its own part of the overall problem. If an assignment function is used, new tasks are mapped onto processors independently of the amount of message traffic, and as the problem grows the number of messages grows along with it.

6.2.2. Process Representation

Many implementations of logic programming languages use a technique called *structure sharing* that allows a compact representation for derived clauses [7]. The basic idea is that in the representation of a derived goal statement, instead of making copies of literals in the the body of a called procedure, a "stack frame" is created, with pointers to the literals in existing clauses, and a structure similar to the runtime stack of Pascal-like languages is formed. One stack frame represents one resolvent. The stack frames contain pointers to the input clauses and a pointer to an environment which has information about variable bindings. One important difference between Pascal stacks and structure sharing is that the stack frames are not removed upon exit from the logic program procedure. At any point in the computation, the value of a term in a goal list is obtained by traversing this structure. Structure sharing is not always the

best implementation technique; a comparison of the relative advantages of structure sharing versus string copying can be found in a paper by Mellish [39].

All previous descriptions of the AND/OR process model have stressed the logical independence of processes, and implied that when a new process is created, copies of terms from the parent process are used to form the body of the new process. An implementation, however, could use a scheme very much like structure sharing in order to minimize the amount of space used by processes. When a new process is created, a "frame" for it would contain pointers to existing clauses in the parent process, and then a pointer to the new process would be added to the list of ready processes. The only time it is necessary to obtain the full string representation of a clause is when a process leaves the PE that created it. At this time the structure would be traversed in order to obtain the complete external representation of the process to be sent off to a neighboring PE. The expected large size of processes and messages means that some form of structure sharing or some other representation technique that takes advantage local memory will be useful for representing processes within any given PE. An architecture that is organized around pools of work may require an entire process/message pair to be delivered to a PE each time a transformation is to be performed, and this could be quite costly.

In Section 5.2, it was mentioned that if a generator creates a ground term (one that contains no variables), then the AND process does not have to reapply the ordering algorithm to modify the dataflow graph that connects literals. If the term contains a variable, the ordering algorithm must be applied again, since there may still be dependencies among literals. Making a graph is time consuming, so an AND process will always check to see if a term from a generator is a ground term. Part of the representation of terms (whether it is based on structure sharing or not) should include information on whether the term is a ground term.

Another improvement in efficiency may be obtained by keeping track of canceled processes. Instead of immediately "garbage collecting" the space occupied by terminated processes, the system should save these processes, and reclaim the

space they occupy only when necessary. In many computations in the AND/OR Process Model, processes are canceled, only to be replaced immediately by new processes created to solve exactly the same problems. A property of logic programs (and functional programs in general) is referential transparency, *i.e.* the value computed by a procedure call is independent of the context in which the call is made. If a process creates a descendant to solve a goal (or goal list) **G**, then cancels **G**, and later starts a different process to solve exactly the same problem, there is no reason the original process for **G** cannot be resurrected to retransmit all solutions for **G**. If the system can restore a canceled process to its previous state, instead of creating a new one, quite a bit of work may be saved.

6.2.3. Secondary Memory

One would assume that as the AND/OR tree of processes is formed, the processes toward the top of the tree will be idle while descendants at the frontier of the tree actively carry out their tasks. Eventually *success* messages work back to the top of the process tree. This assumption is verified by simulations done so far.

This observation can form the basis of an efficient use for a secondary memory. Again, at this level, the topology of the interconnection of PEs or their connection to a secondary memory is not essential to the discussion. One possible arrangement is to have each PE connected to two networks: a message network (N-cube or whatever) and a secondary memory bus.

When a PE's memory starts to become filled with processes, and all of its neighbors are presumably busy (since processes are not migrating), then blocked processes can be written out to the secondary memory. The processes written out should be those at the top of the tree. One can envision a *systolic pipeline* effect here: as the amount of memory devoted to active processes shrinks, freeing space, waiting processes can be brought back into main memory, based on how close they are to the current frontier of the AND/OR tree of processes. The system can anticipate their need, before any active process actually sends a message to one of them. Compare this to systems that use demand paging. If a program

makes a reference to information not currently in main memory, it is blocked until the information is retrieved. There is no way to anticipate which information currently on disk will be needed next, so information stays there until there is a demand for it. In the AND/OR Process Model, the regular structure of process interconnection, and the relative predictability of when a process will be activated, may lead to very efficient use of secondary memory.

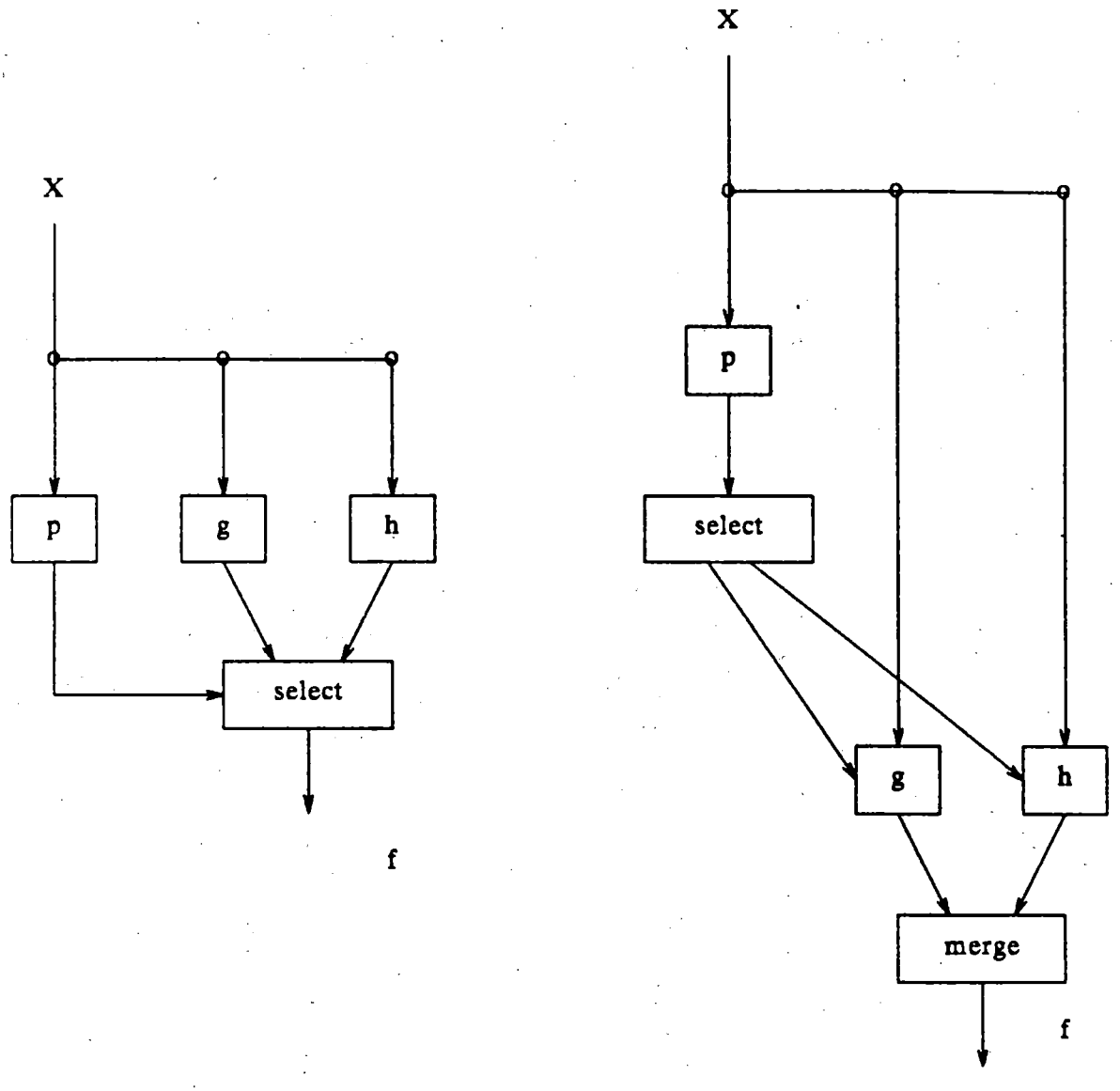
This idea can be extended to situations where memory is filled with only ready processes, after all blocked processes have been moved to secondary storage. During the solution of very large problems, a PE's memory will overflow with processes and messages. The first step in alleviating the congestion is to move out blocked processes, from the top of the AND/OR tree. As the tree continues to grow, a point will be reached where memory will contain only ready processes and their incoming messages. The second stage is to store some subset of these active processes and their messages. Again, the regular structure of the tree of processes will help determine which process/message pairs to move out. Siblings, or processes at the same level in general, do not send messages directly to one another. So, at this stage, processes and messages from the "bottom right" of the tree can be stored, without worrying that processes from the "bottom left" sending them messages.

A global view of the expansion of the AND/OR tree of processes can be characterized as mostly breadth first, as processes create descendants in parallel. When PEs become saturated, and active processes moved out of main memory, the expansion will tend toward depth first, as the leftmost parts of the tree are expanded while the rightmost part stays in secondary memory.

6.2.4. Mechanisms for Inhibiting Concurrency

There are two ways to slow down the creation of new processes when the system starts to become saturated. Each PE can decide when to implement these mechanisms, based on its own current workload.

Figure 21 shows a conditional expression written in ID, along with two possible dataflow schemas that can be generated from the expression [1]. As a result




```
f -> if p(X) then
        g(X)
     else
        h(X);
```

Figure 21: Dataflow Compilations of a Conditional Expression

of the compilation on the left, the predicate P and branches G and H are all evaluated in parallel. The alternative on the right shows a less enthusiastic evaluation, in which P is applied, and then after the result is known, either G or H, but never both, are applied. The schema on the right may lead to a slower computation, since G and H are applied sequentially with respect to P, but fewer processes are created.

In Chapter 3, the same function was written as two clauses in a logic program:

$f(X,Y) \leftarrow p(X)\ \&\ g(X,Y).$
$f(X,Y) \leftarrow not(p(X))\ \&\ h(X,Y).$

A parallel OR process for the literal $f(a,Y)$ creates two AND descendants to evaluate both branches of the conditional expression. In order to inhibit concurrency when necessary, the programming language used in the proposed system should include a conditional operator, such as the -> operator of DEC-10 Prolog:

$f(X,Y)$:- $p(X)$ -> $g(X,Y)$; $h(X,Y)$.

An AND process that encounters this operator in the body would, when the system is busy, create an OR process for $p(X)$; then, depending on the result, it would create a process for either $g(X,Y)$ or $h(X,Y)$. This is in contrast to the rules for forward execution presented in Chapter 5, in which processes for both literals on the right side of each clause are created at the same time.

A second inhibitor of parallelism is simply to switch to a sequential computation when the PEs start to become loaded. However, it would be a mistake to have a PE start using a depth first interpreter when it thinks the system is heavily loaded; instead, the switch to sequential computation should be enforced by a set of policies that control the priorities of the individual processes. The problem with starting to use a depth first interpreter is that parallel processes construct a different (larger) set of answers than sequential interpreters. It would be rather frustrating to use a system that successfully solves a problem when it is not too busy, but then fails (says "not provable") when asked to solve the same problem when it is busy.

As an example of how a general sequential orientation could be given to the computation as a whole, an OR process that creates more than one AND descendant when the system is busy could give the process for the first clause (with respect to the body of the program) a higher priority than the remaining processes.

6.3. Chapter Summary

Programs in execution under the AND/OR Process Model define a pool of tasks that need to be performed. Each task consists of the state of a process and a message destined for it. Multiprocessor implementation of the AND/OR Process Model requires a method for distributing the packets of work to PEs. A possible line of future research is to analyze the simulation results from the APOP interpreter to measure the average size of the packets of work, and then see if the model can be efficiently implemented on any of the existing or proposed workpool architectures.

A different path for future research was outlined in this chapter. Instead of storing the processes and messages in a central pool, from which each PE draws packets of work, processes and messages will reside in the local memories of independent PEs. The advantages to this organization are that large terms (representing process states and success messages) will not have to moved around the system as often, a form of structure sharing might be implemented in order to minimize the memory required to represent processes, and finally it would be straightforward to have the PEs in the system switch to sequential interpretation locally as they and their neighbors become loaded with work during the execution of very large programs.

CHAPTER 7

Conclusion

7.1. Contribution

The major contribution of this dissertation is a method for partitioning a logic program into smaller, independent pieces for parallel solution. Two different sources of parallelism were defined. *OR parallelism* comes from the parallel solution of a procedure that is defined by a number of similar clauses. OR parallelism takes advantage of multiple solutions to a single problem, and essentially replaces backtracking in sequential interpreters. *AND parallelism* is obtained when the literals in the bodies of clauses can be solved simultaneously. AND parallelism is necessary if clauses that represent functions are to be executed in parallel. The interpreter described in this dissertation is the first implementation of AND parallelism for logic programs.

7.2. Related Work

Two other forms of parallelism in logic programs were defined in the original paper on the AND/OR Process Model [15], and yet a fifth form in a paper by Conery, Morris, and Kibler [14]. These other forms are *Search* parallelism, *Stream* parallelism, and *Goal List* parallelism (abbreviated GL).

When a program is very large, it may not be possible to store every clause on every PE. Search parallelism refers to a method for searching across PEs for clauses to use in a resolution step when the program has been partitioned to reside on different PEs. This is an important form of parallelism, especially when logic programs will be used for queries of large databases. If the AND/OR Process Model is to be used to define a database machine, this is an issue that will have to be addressed. So far it been ignored in the implementation of the AND/OR Process Model.

Stream parallelism was the term used to describe the parallelism obtained from coroutines. As described in the chapter on AND parallelism, the biggest difference between the generators in a parallel AND process and the producers in stream parallelism is that generators create a sequence of independent terms, whereas producers create parts of a single term through a series of partial bindings.

Goal List parallelism is simply a parallel search of a goal tree. GL parallelism is similar to OR parallelism, in the sense that the parallelism derives from having a choice of clauses to resolve with a selected literal. The differences are in subproblem size, direction of message transfer, and in opportunities for AND parallelism.

OR processes are oracles, and solve only one literal. The largest problem solved by their direct descendants is proportional in size to the largest body in the procedure defined by the literal. The subproblems created in the GL model are derived goal statements, and can be much larger: the derived goal statement has every remaining literal from the input goal statement plus the literals from the body of the selected clause. If the subproblems are to be sent to independent processing elements, then the size of the subproblems is an important factor. It may be very time consuming to transmit an entire goal stack to other PEs. Ciepielewski and Haridi have defined a structure sharing method that allows independent processes to share a goal stack stored in a common memory (this organization assumes all PEs share the same memory space) [9]. This method will greatly reduce the time required to create subprocesses.

An OR process acts as a message center, deciding when to pass *success* messages to its parent. One advantage of the GL model is that message transfer is unidirectional, and *start* is the only message type required. When a descendant process is created, it becomes totally independent, and there is never a need to communicate with the parent process. Thus the complications arising from waiting and gathering modes are avoided.

An interpreter based on OR processes has an opportunity for exploiting AND parallelism by creating OR processes for more than one literal at a time.

This same opportunity is not available in the GL model. It may be possible to select more than one literal for resolution from a goal statement in the GL model, but the result is only a higher branching factor in the goal tree. The derived goal statements in the subtrees are not any smaller, and the lengths of the derivations are not any shorter. True AND parallelism involves a virtual shortening of the length of a path from the root to a null clause, and this can only be done if sections of a path are derived in parallel by independent interpreters, *i.e.* oracles.

7.2.1. Furukawa, Nitta, and Matsumoto

A system described by Furukawa, Nitta, and Matsumoto is also organized as a set of AND/OR processes [26]. What they refer to as OR parallelism is actually GL parallelism. Their proposed plan for allocation of processes to PEs assigns one process per PE (via an undisclosed assignment function).

7.2.2. Eisinger, Kasif, and Minker

The system of Eisinger, Kasif, and Minker is designed to run on the ZMOB system [21, 50]. The system consists of a problem solver, an extensional database (the set of unit clauses), and an intensional database (set of implications). The problem solver runs on a dedicated PE, and other PEs in the system are charged with gathering information for it. Thus this system is an implementation of search parallelism.

7.2.3. Clark

The IC-Prolog interpreter allows the programmer to annotate clauses to indicate which literals will be the producers of terms [11]. The paper by Clark and McCabe is the first to describe variable bindings with the imagery of dataflow, by talking about values flowing from producers to consumers.

Clark and Gregory extended the language, and showed how clauses can be used to construct networks of communicating processes [12]. The latter interpreter uses Dijkstra's committed choice nondeterminism: when there is a choice of clauses to unify with a selected literal, one is chosen at random; the interpreter will never backtrack to try to undo this choice [19].

IC-Prolog is one of the languages expected to run on the ALICE machine, a general purpose multiprocessor for applicative languages [17]. When running on a multiprocessor machine, IC-Prolog is an implementation of stream parallelism. ALICE is an example of a multiprocessor organized around the workpool concept.

7.2.4. Haridi

Haridi's thesis presents a logic programming system that is not based on resolution [28]. It uses natural deduction as the framework for organizing a proof. A natural deduction proof can also be represented as a tree of inferences. That thesis devotes a chapter to parallelism in that model of computation. The specific form of parallelism is GL parallelism, a parallel search of the natural deduction proof tree. A method for allowing independent processes to share information in that proof tree is described in in a later paper by Ciepielewski and Haridi [9].

7.2.5. Bowen

Bowen has designed a system that is an implementation of van Emden's abstract model for a Prolog interpreter [6, 23]. The form of parallelism obtained is GL parallelism. This system is also expected to run on the ZMOB architecture [50].

7.2.6. Monteiro

Another language that uses Horn clauses to describe concurrent processes is Distributed Logic, or DL for short, described in a paper by Monteiro [41]. The language allows programmers to describe, in a Prolog-like language, the communicating sequential processes of Hoare [29]. The goal is to allow programmers to specify concurrent processes, and their communication paths, as opposed to having programmers specify a function and having the interpreter decide what can or cannot be performed in parallel.

7.2.7. Mellish

Mode declarations are used by the DEC-10 Prolog compiler to generate more efficient code. The techniques for generating DEC-10 machine code from Prolog clauses is defined in Warren's thesis [60]. If the compiler knows in advance that certain arguments will or will not be instantiated, the unification process can be speeded up quite a bit. A method for automatic generation of mode declarations is described by Mellish [39]. By considering the entire program, and not just each clause by itself, this method is much more effective than the ordering algorithm of Section 5.1.2 in figuring out which argument positions correspond to inputs and which are outputs.

7.2.8. Pereira and Porto

The intelligent backtracking scheme of Pereira and Porto is more effective than the system of resets and fail lists described in Chapter 5. When a unification fails, their system analyzes the cause of the failure [47, 48]. By including this type of analysis, AND processes could be more efficient. This would require *fail* messages to carry reasons for failure, *e.g.* the message $fail(p(X,a))$ could mean "literal p cannot be solved with term a in the second argument position."

The shortcomings of chronological backtracking have been described in a number of papers, not all of them in the context of logic programming. Other discussions can be found in a recent paper by Freuder [25] and in Steele's thesis [55].

7.3. Future Research

7.3.1. Process Migration

A policy for deciding which processes should be transferred to neighboring PEs needs to be developed. An example of one rule was given in Section 6.2.1, where it was shown that, by transferring the process that performs a recursive call, work can be spread evenly across PEs. A tradeoff associated with this policy is due to the time it takes to transfer a problem. It may be the case that for a

small problem, it takes longer to transfer it to another PE and wait for results, than to simply keep it on the original PE.

Note that this policy, and others, depend on physical characteristics of the network (topology, bandwidth, *etc.*).

7.3.2. Migration Count

In the Burton and Sleep network, a process is transferred only once, from the PE that creates it to any neighboring PE. Since messages are transmitted only between parents and descendants in the AND/OR model, the longest path would cross two links (this happens when PE_1 creates a process and its descendant, the parent migrates to PE_2 and the descendant to PE_3, and PE_2 can only communicate with PE_2 indirectly, through PE_1). A subject of further experimentation is to see how varying the migration count, or the number of times a process may be transferred to other PEs, effects message transmission times.

As a side note, by setting the migration count to 0, a programmer will be able to test a program as if it were running on a system with a single PE. It will be much easier to debug a program if it "stays put" when it is being worked on. Then, when the programmer is confident the program works, parts can be allowed to migrate to other PEs.

7.3.3. Backward Execution

The method for handling *fail* and *redo* messages presented in Chapter 5 is very conservative. A number of processes are canceled when in fact they are doing work that must be done later, and the sequential processing of fail sequences seems unnecessarily restrictive. The general policy used when writing this first interpreter for parallel AND processes was to be implement a correct "tuple generator", with the emphasis on constructing all tuples. Further parallelism, or more efficient methods, are possible. One example is to process fail sequences in parallel. Also, a more sophisticated analysis of the dependencies between literals may show that a literal does not have to be reset every time the process for an earlier literal is sent a *redo* message.

7.3.4. High Level Language

The literal ordering algorithm (Section 5.1.2) is, in effect, an attempt to infer mode declarations for the literals within the body of one clause. An example of where this algorithm fails is

$$mm(A,B,C) \leftarrow transpose(B,BT) \ \& \ mmt(A,BT,C).$$

The proper sequence solves *transpose* first, generating a binding for BT. The ordering algorithm looks for literals that are connected to the set of variables instantiated in the head, in this case {A,B}. The algorithm uses these variables in order, and thus it finds *mmt* connected by the variable A before it finds *transpose* connected by B.

In this example, an incorrect ordering means the goal will fail. This happens because $mmt(A,BT,C)$ cannot be solved unless A and BT are instantiated. In other cases, an incorrect ordering simply means the AND process will be inefficient. (For a dramatic illustration of how different orderings effect efficiency, see Baxter's "verify and choose" problem solver [4].)

The cases where an AND process actually fails because of an incorrect ordering are those cases where the subgoals are calls to functions, and the calls do not have all input variables bound. One remedy is to use Mellish's automatic mode generation algorithm for a more accurate specification of modes [39]. However, the best solution to this problem is to include a functional notation into the language used by the interpreter. For example, if *transpose* is defined as a function, the clause

$$mm(A,B,C) \leftarrow mmt(A,transpose(B),C).$$

could be translated into the previous clause, and as a result of the translation a correct ordering would be generated.

Two systems that incorporate functional notations into a logic language are the LOGLISP system from Syracuse (Bowen and Kowalski [5]) and Eggert and Schorre's extended syntax for Prolog [20]. The use of functional notation does not necessarily mean that the functions have to be deterministic. They will still

be solved by OR processes, and can still be defined by any number of clauses. The notation simply restricts the input/output sense of variables in the literal, in a manner that does not require the programmer to provide mode declarations or control annotations in the clause.

References

[1] Arvind, K. P. Gostelow, and W. E. Plouffe.
An Asynchronous Programming Language and Computing Machine.
Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, December, 1978.

[2] Ashcroft, W.
LUCID.
Communications of the ACM, 21(8):613-641, August, 1978.

[3] Backus, J.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
Communications of the ACM, 21(8):613-641, August, 1978.

[4] Baxter, L.
A Prolog Program Illustrating a Verify and Choose Method.
Technical Report, Department of Computer Science, York University, 1981.

[5] Bowen, K. A. and R. A. Kowalski.
Amalgamating Language and Metalanguage in Logic Programming.
Technical Report, School of Computer and Information Science, Syracuse University, April, 1981.

[6] Bowen, K. A.
Concurrent Execution of Logic.
In *Proceedings of the First International Logic Programming Conference*, pp. 26-30. Faculte des Sciences de Luminy, Marseille, Sept, 1982.

[7] Boyer, R. S. and J. Moore.
The Sharing of Structure in Theorem Proving Programs.
In B. Meltzer and D. Michie, Eds., *Machine Intelligence 7*, Edinburgh University Press, 1972.

[8] Burton, F. W., and M. R. Sleep.
Executing Functional Programs on a Virtual Tree of Processors.
In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp. 187-194. ACM, October, 1981.

[9] Ciepielewski, A. and S. Haridi.
Formal Models for Or-Parallel Execution of Logic Programs.
CSALAB Working Paper 821121, Royal Institute of Technology, Stockholm, Sweden, 1982.

[10] Clark, K. L.
Negation as Failure.
In H. Gallaire and J. Minker, Eds., *Logic and Databases*, Plenum Press, 1978.

[11] Clark, K. L., and F. McCabe.
The Control Facilities of IC-Prolog.
In D. Michie, Ed., *Expert Systems in the Microelectronic Age*, Edinburgh University Press, 1979.

[12] Clark, K. L. and S. Gregory.
A Relational Language for Parallel Programming.
In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp. 171-178. ACM, October, 1981.

[13] Colmerauer, A., H. Kanoui, and M. van Caneghem.
Last Steps Toward an Ultimate Prolog.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 947-948. August, 1981.

[14] Conery, J. S., P. H. Morris, and D. F. Kibler.
Efficient Logic Programs: A Research Proposal.
Technical Report 166, Department of Information and Computer Science, University of California, Irvine, April, 1981.

[15] Conery, J. S. and D. F. Kibler.
Parallel Interpretation of Logic Programs.

In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp. 163-170. ACM, October, 1981.

[16] Dahl, V.
On Database Systems Development Through Logic.
ACM Transactions on Database Systems, 7(1):102-123, March, 1982.

[17] Darlington, J., and M. Reeve.
ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages.
In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp. 65-76. ACM, October, 1981.

[18] Davis, R. E.
Generating Correct Programs from Logic Specifications.
Technical Report 79-05-001, Information Sciences, University of California, Santa Cruz, May, 1979.

[19] Dijkstra, E. W.
A Discipline of Programming.
Prentice-Hall, Englewood Cliffs, N.J., 1976.

[20] Eggert, P. R. and D. V. Schorre.
Logic Enhancement: A Method for Extending Logic Programming Languages.
In *Conference Record of the Symposium on LISP and Functional Programming*, pp. 74-80. ACM, August, 1982.

[21] Eisinger, N., S. Kasif, and J. Minker.
Logic Programming: A Parallel Approach.
In *Proceedings of the First International Logic Programming Conference*, pp. 1-8. Faculte des Sciences de Luminy, Marseille, Sept, 1982.

[22] van Emden, M. H. and R. A. Kowalski.
The Semantics of Predicate Logic as a Programming Language.
Journal of the ACM, 23(4):773-742, October, 1976.

[23] van Emden, M. H.

An Interpreting Algorithm for Prolog Programs.

In *Proceedings of the First International Logic Programming Conference*, pp. 56-64. Faculte des Sciences de Luminy, Marseille, Sept, 1982.

[24] van Emden, M. H.

Warren's Doctrine on the Slash.

Logic Programming Newsletter, 4:10, January, 1983.

[25] Freuder, E. C.

A Sufficient Condition for Backtrack-Free Search.

Journal of the ACM, 29(1):24-32, January, 1982.

[26] Furukawa, K., K. Nitta, and Y. Matsumoto.

Prolog Interpreter Based on Concurrent Programming.

In *Proceedings of the First International Logic Programming Conference*, pp. 38-44. Faculte des Sciences de Luminy, Marseille, Sept, 1982.

[27] Gostelow, K. P. and R. Thomas.

Performance of a Simulated Dataflow Computer.

IEEE Transactions on Computers, C-29(10):905-919, October, 1980.

[28] Haridi, A.S.

Logic Programming Based on a Natural Deduction System.

PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1981. Report TRITA-CS-8104.

[29] Hoare, C. A. R.

Communicating Sequential Processes.

Communications of the ACM, 21(8):666-677, August, 1978.

[30] Hopcroft, J. E.

Introduction to Automata Theory, Languages, and Computation.

Addison-Wesley, Reading, Mass., 1979.

[31] Katz, E. P.

A Realization of Relational Semantics in an Automatic Programming System.

PhD thesis, University of Southwest Louisiana, 1978.

[32] Kibler, D. and B. Porter.
Episodic Learning.
to appear in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* William Kaufmann Co., August, 1983.

[33] Kowalski, R. A.
Predicate Logic as a Programming Language.
In *IFIPS 74.*

[34] Kowalski, R. A.
Logic for Problem Solving.
Elsevier - North Holland, New York, 1979.

[35] *Logic Programming Newsletter.*
Available from L. M. Periera, Ed., Departamento de Informatica, Universidade Nova de Lisboa, Lisbon, Portugal.

[36] MacLennan, B. J.
Introduction to Relational Programming.
In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp. 213-220. ACM, October, 1981.

[37] Manna, Z.
Mathematical Theory of Computation.
McGraw-Hill, New York, 1974.

[38] Martelli, A. and U. Montanari.
An Efficient Unification Algorithm.
ACM Transactions on Programming Languages and Systems, 4(2):258-282, April, 1982.

[39] Mellish, C. S.
An Alternative to Structure-Sharing in the Implementation of a Prolog Interpreter.
DAI Research Paper 150, Department of Artificial Intelligence, University of Edinburgh, 1980.

[40] Mellish, C. S.
The Automatic Generation of Mode Declarations for Prolog Programs.
DAI Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, August, 1981.

[41] Monteiro, L.
A Horn Clause - Like Logic for Specifying Concurrency.
In *Proceedings of the First International Logic Programming Conference*, pp. 1-8. Faculte des Sciences de Luminy, Marseille, Sept, 1982.

[42] Myers, G. J.
Advances in Computer Architecture.
John Wiley and Sons, New York, 1978.

[43] Nilsson, N. J.
Problem Solving Methods in Artificial Intelligence.
McGraw-Hill, New York, 1971.

[44] Page, R. L., M. G. Conant, and D. H. Grit.
If-then-else as a Concurrency Inhibitor in Eager Beaver Evaluation of Recursive Programs.
In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp. 179-186. ACM, October, 1981.

[45] Pereira, F. C. N. and D. H. D. Warren.
Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks.
Artificial Intelligence, 13:231-278, 1980.

[46] Pereira, L. M., F. C. N. Pereira, and D. H. D. Warren.
Users Guide to DECsystem-10 Prolog.
Technical Report, Department of Artificial Intelligence, University of Edinburgh, September, 1978.

[47] Pereira, L. M. and A. Porto.
Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory.

Report 2/79, Departamento de Informatica, Universidade Nova de Lisboa, October, 1979.

[48] Pereira, L. M. and A. Porto.
An Interpreter of Logic Programs Using Selective Backtracking.
Report 3/80, Departamento de Informatica, Universidade Nova de Lisboa, July, 1980.

[49] Pereira, L. M.
Logic Control with Logic.
Report 2/82, Departamento de Informatica, Universidade Nova de Lisboa, February, 1982.

[50] Rieger, C., R. Trigg, and B. Bane.
ZMOB: A New Computing Engine for AI.
Technical Report 1028, Department of Computer Science, University of Maryland, March, 1981.

[51] Roberts, G. M.
An Implementation of Prolog.
Master's thesis, Department of Computer Science, University of Waterloo, 1977.

[52] Robinson, J. A.
A Machine Oriented Logic Based on the Resolution Principle.
Journal of the ACM, 12(1):23-41, January, 1965.

[53] Sleep, M. R.
Applicative Languages, Dataflow, and Pure Combinatory Code.
In *COMPCON Spring 80*, pp. 112-115. IEEE, February, 1980.

[54] Sowa, M. and T. Murata.
A Dataflow Computer Architecture with Program and Token Memories.
IEEE Transactions on Computers, C-31(9):820-824, September, 1982.

[55] Steele, G. L.
The Definition and Implementation of a Computer Programming Language Based on Constraints.

Technical Report AI-TR-595, MIT AI laboratory, August, 1980.

[56] Treleaven, P. C., D. R. Brownbridge, and R. C. Hopkins.
Data-Driven and Demand-Driven Computer Architecture.
ACM Computing Surveys, 14(1):93-143, March, 1982.

[57] Turner, D. A.
A New Implementation Technique for Applicative Languages.
Software - Practice and Experience, 9(1):31-49, January, 1979.

[58] Warren, D. H. D.
WARPLAN: A System for Generating Plans.
Paper 76, Department of Artificial Intelligence, University of Edinburgh, June, 1974.

[59] Warren, D. H. D., L. M. Pereira, and F. C. N. Pereira.
Prolog - The Language and its Implementation Compared with LISP.
ACM SIGPLAN Notices, 12(8):109-115, 1977.

[60] Warren, D. H. D.
Implementing Prolog: Compiling Predicate Logic Programs.
D.A.I. Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, May, 1977.

[61] Warren, D. H. D.
Logic Programming and Compiler Writing.
Software - Practice and Experience, 10:97-125, 1980.

[62] Warren, D. H. D.
Efficient Processing of Interactive Relational Database Queries Expressed in Logic.
Paper 156, Department of Artificial Intelligence, University of Edinburgh, September, 1981.

[63] Warren, D. H. D. and F. C. N. Pereira.
An Efficient Easily Adaptable System for Interpreting Natural Language Queries.
Paper 155, Department of Artificial Intelligence, University of Edinburgh,

February, 1981.

[64] Warren, D. H. D.
Higher-Order Extensions to Prolog - Are They Needed?.
Paper 156, Department of Artificial Intelligence, University of Edinburgh, September, 1981.

[65] Wise, M. J.
A Parallel Prolog: The Construction of a Data Driven Model.
In *Conference Record of the Symposium on LISP and Functional Programming*, pp. 56-66. ACM, August, 1982.

[66] Wulf, W. A. and M. Shaw.
Abstraction and Verification in ALPHARD: Defining and Specifying Iteration and Generators.
Communications of the ACM, 20(8):553-564, August, 1977.

APPENDIX I

Detailed Definition of the Interpreter

The higher level procedures of the interpreter are discussed in detail in this Appendix. The interpreter was designed to be modular, in the sense that the low level mechanisms (for message passing, process scheduling, *etc.*) are all defined in a kernel file, routines that support one particular parallel model[1] are defined in another file, and, finally, definitions of processes are in a third file. The code given here is for parallel AND and OR processes.

A. Kernel

The special characters '#', '@', and ':' are defined as operators, in order to make it easier to read the complicated terms that represent the states of processes.

```
:-op(300,fx,#).
:-op(350,xfy,@).
:-op(350,xfy,:).
```

The interpreter is started when the user types *solve*(*G*), where *G* is any conjunction of goals; it is necessary to turn *G* into the required internal form and then start an AND process to solve it:

```
solve(G):-initialize,
        top_goal(G),!,
        cycle.
```

cycle is the main loop of the interpreter. It continually pairs up messages and processes and calls the routine that performs the required state transition. After saving the new state of a process, *test_for_result* is called. If it succeeds, the cycle stops. If it fails (meaning there is no result yet), then Prolog retries every goal in the body; however, these are all deterministic, so eventually Prolog

[1] In this case the AND/OR Process Model; another is the Goal List Model [6].

backtracks all the way to *read_message*, and cycle gets another message/process pair:

```
cycle:-
        read_message(Src,N,Msg,T),      % get a message for any process N
        state_of(N,State),              % and the current state of N
        step(State,Src,N,Msg,T,NewS),   % transform into new state . . .
        ptrace(State,Src,N,Msg,T,NewS),
        save_state(NewS),
        test_for_result.
```

The states of processes, and messages in transit between processes, are stored in Prolog's internal database. The above calls to *read_message* and *state_of* are database retrievals that pick up a message at random and then the current state of the destination of that message. States of processes are all terms of the form

$p(N,S,T)$

N is the process ID number, S is the current state, and T is the time of the last transformation. Messages are all terms of the form

$m(S,D,M,T)$

S is the ID of the sending process, D is the ID of the destination, M is the content of the message, and T is the timestamp.

step is a model-dependent procedure that executes exactly one step of a process. The parameter S will be the current state of the process; *step* returns the new state of the process by binding the new state to the parameter *NewS*. The other four parameters define the message that that triggers the transformation defined in *step*. *save_state* records the new state of the process after the state transformation.

```
state_of(N,p(N,S,T)):-          % retrieve state of process N
        recorded(process,p(N,S,T),Key),
        erase(Key),   !.
```

```
save_state(P):
        recorda(process,P,R2),      !.

send_message(Source,Dest,M,T):-
        recordz(msg,m(Source,Dest,M,T),R),
        stats(msg,m(Source,Dest,M)).

read_message(Source,Dest,M,T):-
        recorded(msg,m(Source,Dest,M,T),R),
        erase(R).
```

step consists of a number of clauses, each one of which defines exactly one state transition. The input arguments correspond to a process and a message. The interpreter must scan the list of steps, looking for one that can be applied to the inputs. The clauses are presented here in exactly the same order as they occur in the interpreter, which is the order in which Prolog tries to apply them.

Every step has calls to procedures *wakeup* and *executed*, which update global counters used in timing measurements. These procedures use the old time stored with the process state and update it to reflect the timestamp in the message. Notice also that *every* clause for *step* ends with a cut symbol, meaning that only one step is ever applied for any process-message pair. The cut symbol is required because of the way *cycle* is constructed to do iteration through backtracking.

This first step is for any process that reads a cancel message. It updates the timing data one last time, and returns *done* (the special final state) as the output state:

```
step(p(ID,_,_,T1),                  % implementation of 'cancel'
        PID,ID,cancel,Tm,           % messages for any process
        p(ID,_,done,t(T2,A,S))):-       % in any state
        wakeup(T1,Tm,TX),
        executed(ID,TX,t(T2,A,S)),
        zap_messages(ID),
        cancel_desc(ID,T2), !.
```

B. AND Processes

The *step* procedure for parallel AND processes is described in detail in this section. The state of a parallel AND process is a single complex term:

and(PID,Body,Ans,Lits,HL,Solved,Linear,FL)

where the fields have the following meanings:

PID	is the parent's ID
Body	is the original goal list (with variables uninstantiated)
Ans	is a copy of Body with current values of variables; Ans is used to make the message eventually sent back.
Lits	is the set of the literals of the body. Each item is of the form #N:[Lit,Pred,Gen,Desc,U,UU] where Lit is the "numbered" literal, Pred is the list of predecessors (aka redo sequence), Desc is the process ID of the OR process solving the literal, U is list of solutions used, UU is a list of as yet unused solutions, and Gen is a list of variables generated by Lit.
HL	is an abbreviated Lits structure for the head (among other things, it has the head redo list)
Solved	is a list of numbers of literals solved so far
Linear	is the linear ordering of literal numbers
FL	is the current list of failure contexts

It is possible that a descendant OR process will send a message, and then read a cancel message and terminate. The AND process should ignore any messages sent by these "ghost" processes. *msg_screen* will FAIL (i.e. another step will be tried) if *Src* is either the parent's ID or it is in the *Lits* structure for some literal; if *msg_screen* succeeds it means the process that sent the message is now dead, so the message should have no effect, *i.e.* output state = input state.

```
step(p(ID,and(PID,Body,Ans,LitsIn,HL,Solved,Linear,FL),T1),
     Src,ID,Msg,Tm,
     p(ID,and(PID,Body,Ans,LitsIn,HL,Solved,Linear,FL),T1)):-
     msg_screen(Src,PID,LitsIn),        !.
```

This is the first step taken by a new AND process. The *Solved* list is [*head*], and there will be no descendants; just start processes for all literals that have only [*head*] as predecessors (there has to be at least one as long as there are no

cycles in the dataflow graph). *ready*(*G*,*S*,*B*) is a predicate that is true if goal *G* from body *B* is ready to be solved, given the list of already solved literals *S*; the DEC-10 Prolog procedure *setof* collects the set of all such literals from *B*:

```
step(p(ID,and(PID,Body,Ans,LitsIn,HL,Solved,Linear,FL),T1),
        PID,ID,start,Tm,
        p(ID,and(PID,Body,Ans,LitsOut,HL,Solved,Linear,FL),t(T,A,S))):-
        setof(G,ready(G,Solved,LitsIn),R),
        executed(ID,T1,t(T,A,S)),
        start_all_andp(R,ID,Ans,LitsIn,LitsOut,T),        !.
```

An AND process that receives a success message unifies the the message with the current Answer structure, and starts a new set of OR processes; if there are no more literals to solve, send *success*(*ANS*) to the parent process. *solver*(*N*,*B*,*D*) equates literal number *N* in body *B* with descendant *D*; in this case it is used to find the literal number of the descendant that sent the success message. *add_ans* updates the used-answer list. *fwd_state* updates the status of the literals and creates new processes (code given below):

```
step(p(ID,and(PID,Body,Ans,LitsIn,HL,Solved,Linear,FLi),T1),
        Desc,ID,success(M),Tm,
        p(ID,and(PID,Body,Ans,LitsOut,HL,[N|Solved],Linear,FLo),T2)):-
        wakeup(T1,Tm,TX),
        executed(ID,TX,T2),
        apply_ans(#N:M,Ans),
        add_ans(N,M,LitsIn,LitsTmp),
        fwd_state(Ans,LitsTmp,LitsOut,FLi,FLo,[N|Solved],ID,PID,T2),
        !.
```

When a fail message is received from process for *N*, add *N* to the fail context, send somebody a redo, modify the status of every literal after *N* in the linear ordering, and rebuild the answer template (without values from the generator that was sent the redo message):

```
step(p(ID,and(PID,Body,AnsIn,Li,HL,Si,Linear,FLi),T1),
        Desc,ID,fail,Tm,
        p(ID,and(PID,Body,AnsOut,Lo,HL,So,Linear,FLo),t(T2,A,S))):-
        remove_desc(Desc,Li,Lt1),
        wakeup(T1,Tm,TX),
        executed(ID,TX,t(T2,A,S)),
        send_redo(X,Lt1,Lt,Si,St,ID,T2,V),
        tail(Linear,X,Rem),
        bkwd_state(V,Rem,Body,AnsOut,Lt,Lo,St,So,ID,T2,FLt,FLo), !.
```

In the previous step, if the call to redo_literal fails, it means ***head*** is the literal to be redone, so the AND process fails:

```
step(p(ID,and(PID,Body,Ans,LitsIn,HL,Solved,Linear,FLi),T1),
        Desc,ID,fail,Tm,
        p(ID,done,t(T2,A,S))):-
        wakeup(T1,Tm,TX),
        executed(ID,TX,t(T2,A,S)),
        send_message(ID,PID,fail,T2),     !.
```

When a redo message is read, start a failure context from the head (this is where the list *HL* is used), and find out which literal to redo. Note: assume *FL* is always [] before a redo received, and it is (by definition) [***head***] after redo is sent to a literal:

```
step(p(ID,and(PID,Body,AnsIn,Li,[H1|Hn],Si,Linear,[[]]),T1),
        Desc,ID,redo,Tm,
        p(ID,
        and(PID,Body,AnsOut,Lo,[H1|Hn],So,Linear,FLo),t(T2,A,S))):-
        wakeup(T1,Tm,TX),
        executed(ID,TX,t(T2,A,S)),
        hl([H1|Hn],Li,HLits),
        ext_rl(HLits,head,H1,XX,[head],FC),
        send_redo(XX,Li,Lt,Si,St,ID,T2,V),
        tail(Linear,XX,Rem),
        bkwd_state(V,Rem,Body,AnsOut,Lt,Lo,St,So,ID,T2,[FC],FLo),!.
```

Those are the steps for AND processes; more clauses from the ***step*** procedure are explained in the next section. The remaining clauses in this section are from the major procedures *fwd_state*, *bkwd_state*, and *redo_literal*.

Literal N has been solved; figure out new state information for the AND process. If N was the last literal (if the *Solved* list is as long as the body) send *success*(*Ans*), else start a new set of descendants and update the *Lits* structure.

```
fwd_state(Ans,LitsIn,LitsIn,FLi,[[]],Solved,ID,PID,t(T,A,S)):-
        length(Solved,L1),
        length(Ans,L2),
        L2 is L1-1,              % Solved includes 'head' . . .
        strip(Ans,Msg),
        send_message(ID,PID,success(Msg),T).

fwd_state(Ans,LitsIn,LitsOut,FLi,FLo,Solved,ID,PID,t(T,A,S)):-
        setof(G,ready(G,Solved,LitsIn),New),
        start_all_andp(New,ID,Ans,LitsIn,LitsOut,T),
        fix_fl(New,FLi,FLo),             % remove newly started processes
        !.  %  and any extra []'s from FLI

fwd_state(Ans,L,L,F,F,S,ID,PID,T).              % no new processes . . .
```

bkwd_state figures out which literals to reset and which to cancel, and then starts new processes for canceled literals which can be restarted (since reset generators are still solved generators):

```
bkwd_state(V,Rest,Body,AnsOut,LitsIn,LitsOut,Si,So,ID,T,FLi,FLo):-
        bk(V,Rest,LitsIn,LitsTmp,Si,So,ID,T),
        rebuild(Body,LitsTmp,So,AnsOut),
        % after resets, it may be possible to restart some processes
        restart(ID,AnsOut,LitsTmp,LitsOut,T,So,FLi,FLo).
```

The next procedure is the heart of the ***redo_literal*** procedure. The call has the form *rl*(*FC*,*Lits*,*Desc*,*NFC*,*X*), with *FC* bound to the current failure context, *Lits* bound to the list of literals, and *Desc* the number of the literal that failed. This procedure returns a new failure context in *NFC* and the identity of the literal to redo in *X*.

```
rl(FC,[#N:[Lit,Pred|_]|Ln],Desc,NFC,X):-        % Desc can be appended,
        concat(FC,[Desc,X]_],[N]Pred]),!,       % and X is its predecessor
        append(Desc,FC,NFC).
rl(FC,[#N:[Lit,Pred|_]|Ln],Desc,FC,0):-         % Desc cannot be appended
        concat(FC,_,[N]Pred]),      !.
rl(FC,[L1|Ln],Desc,NFC,X):-                     % try the next Lit
        rl(FC,Ln,Desc,NFC,X).
```

C. OR Processes.

The state of an OR process is a term

or(PID,Mode,Orig,Sent,Wait,Desc)

PID is the process ID of the parent AND process. *Mode* is the current operating mode (gathering or waiting), *Sent* and *Wait* are lists of answers, and *Desc* is the current list of descendant AND processes.

A new OR process knows only about the goal it is supposed to solve; it creates descendant AND processes with the bodies of each head that matches the goal; the descendant list will be a list of pairs of process Id's and goal lists that those processes are working on; the waiting list will be initialized to be the unified heads of assertions (clauses with bodies = *true*); finally, if the waiting list is not empty, send the first answer from this list and go into the gathering mode; otherwise go into the waiting mode:

```
step(p(Id,or(PId,X,Orig,[],[],[]),T1),
        PId,Id,start,Tm,
        p(Id,or(PId,Mode,Orig,Sent,Wait,Desc),t(T2,A,S))):-
        bagof([Orig,Body],clause_for(Orig,Body),Bodies),
        list_sort(Bodies,NonTrue,True),
        executed(Id,T1,t(T2,A,S)),
        start_all(Id,NonTrue,Desc,T2),
        init_lists(True,Sent,Wait),
        maybe_send(Id,PId,Sent,Mode,T2),
        !.
```

The above call to *bagof* puts the body of every clause with a head that matches *Orig* into the list *Bodies*; then *list_sort* splits that list into lists *True* and *NonTrue* (lists of unit and nonunit clauses, repectively). For every body in *NonTrue*, an AND process is started. All unit clauses in *True* are used to make the initial

sets of answers. If *Sent* contains an answer, *maybe_send* makes a success message out of it and unifies *Mode* with *gathering*, otherwise Mode is unified with *waiting*.

If the previous step fails (because *bagof* fails), it means there are no clauses with heads that unify with *Orig*; this normally means the new OR process fails immediately. However, this provides a "hook" for evaluable predicates. If *Goal* is in the list of known predicates or special predicates, then solve it directly and return the answer:

```
step(p(Id,or(PId,X,Goal,_,_,_),T1),
        PId,Id,start,Tm,
        p(Id,or(PId,gathering,Goal,[Ans],[],[]),t(T2,A,S))):-
        Goal=..[Functor|Args],
        primitive(Functor),!,          % succeeds for special functors
        copy(Goal,Ans),
        Ans,!,                         % do it
        executed(Id,T1,t(T2,A,S)),
        send_message(Id,PId,success(Ans),T2),
        !.
```

Send a fail message and terminate if the functor of the goal is not in the list of known special predicates (the call to *primitive* in the previous clause failed) or if the call to the primitive itself failed:

```
step(p(Id,or(PId,X,G,_,_,_),T1),
        PId,Id,start,Tm,
        p(Id,_,done,t(T2,A,S))):-
        executed(Id,T1,t(T2,A,S)),
        send_message(Id,PId,fail,T2),
        !.
```

A waiting OR process handles a fail message by removing the process that sent the message from the list of descendants; if that list is now empty, the OR process itself fails:

```
step(p(Id,or(PId,waiting,G,SList,WList,Desc),T1),
        Son,Id,fail,T,
        p(Id,_,done,t(T2,A,S))):-
        remove(Son,Desc,[]),                % works if removing Son from
        wakeup(T1,T,TX),                    % Desc leaves []
        executed(Id,TX,t(T2,A,S)),
        send_message(Id,PId,fail,T2),
        !.
```

The above call to *remove* failed (more than one descendant left), so just remove the descendant and keep waiting:

```
step(p(Id,M,or(PId,waiting,G,SList,WList,Desc),T1),
        Son,Id,fail,T,
        p(Id,M,or(PId,waiting,G,SList,WList,NewDesc),T2)):-
        remove(Son,Desc,NewDesc),
        wakeup(T1,T,TX),
        executed(Id,TX,T2),
        !.
```

A gathering OR process handles a fail message by removing the sender from the descendant list; a now-empty descendant list does not imply failure, though, since there may be messages in the waiting list (even if there aren't any of those, wait for redo before failing):

```
step(p(Id,M,or(PId,gathering,G,SList,WList,Desc),T1),
        Son,Id,fail,T,
        p(Id,M,or(PId,gathering,G,SList,WList,NewDesc),T2)):-
        remove(Son,Desc,NewDesc),
        wakeup(T1,T,TX),
        executed(Id,TX,T2),
        !.
```

If a waiting OR process receives a success message, it unifies it with the original goal to compute the result. If the result has not yet been sent to the parent, it is sent and then added to the list of answers sent, and the process goes into the gathering mode. If the answer is in the list of answers already sent, it is ignored.

```
step(p(Id,M,or(PId,waiting,G,SList,WList,Desc),T1),
        Son,Id,success(GList),T,
        p(Id,M,or(PId,NextState,G,NextSList,WList,Desc),t(T2,A,S))):-
        wakeup(T1,T,TX),
        executed(Id,TX,t(T2,A,S)),
        member([Son,SonsHead,SonsList],Desc),
        copy(G,Result),
        copy([SonsHead,SonsList],[NewHead,NewList]),
        unify(Result,NewHead,GList,NewList),
        next_state(Result,SList,NextState,NextSList,Id,PId,T2),
        send_message(Id,Son,redo,T2),
        !.
```

A gathering OR process handles a success message by seeing if the message is in either the wait list or the sent list; if not, the message is added to the wait list; in either case, send a redo message:

```
step(p(Id,M,or(PId,gathering,G,SList,WList,Desc),T1),
        Son,Id,success(GList),T,
        p(Id,M,or(PId,gathering,G,SList,NextWList,Desc),t(T2,A,S))):-
        member([Son,SonsHead,SonsList],Desc),
        wakeup(T1,T,TX),
        executed(Id,TX,t(T2,A,S)),
        copy(G,Result),
        copy([SonsHead,SonsList],[NewHead,NewList]),
        unify(Result,NewHead,GList,NewList),
        next_state_2(Result,SList,WList,NextWList),
        send_message(Id,Son,redo,T2),
        !.
```

Waiting OR processes will never get a redo message. There are four cases to consider for a *gathering* OR process that receives a redo message, depending on the states of the waiting and descendant lists:

(1) Both lists are empty; in this case the process fails.

(2) Wait list is empty, descendant list is not empty; change to waiting mode.

(3) Descendant list is empty, but there are some answers in the wait list; move the first answer from the wait list to the sent list, and send it.

(4) Neither list empty; same actions as case (3).

```
step(p(Id,M,or(PId,gathering,G,SList,[],[]),T1),
        PId,Id,redo,T,
        p(Id,_,done,t(T2,A,S))):-
        wakeup(T1,T,TX),
        executed(Id,TX,t(T2,A,S)),
        send_message(Id,PId,fail,T2),
        !.

step(p(Id,M,or(PId,gathering,G,SList,[],Desc),T1),
        PId,Id,redo,T,
        p(Id,M,or(PId,waiting,G,SList,[],Desc),t(T2,A,S))):-
        wakeup(T1,T,TX),
        executed(Id,TX,t(T2,A,S)),
        !.

step(p(Id,M,or(PId,gathering,G,SList,[W1|Wn],Desc),T1),
        PId,Id,redo,T,
        p(Id,M,or(PId,gathering,G,[W1|SList],Wn,Desc),t(T2,A,S))):-
        wakeup(T1,T,TX),
        executed(Id,TX,t(T2,A,S)),
        send_message(Id,PId,success(W1),T2),
        !.
```

The last step catches system error conditions. The head of this step matches any process state and any message, so if one of the earlier steps did not catch the process/message pair, this one will:

```
step(p(ID,_,State,T1),        % this catches any kind (AND or OR)
        Desc,ID,Msg,Tm,
        p(ID,done,t(T2,A,S))):-
        wakeup(T1,Tm,TX),
        executed(ID,TX,t(T2,A,S)),
        write(**System Error: no step succeeds for process ),
        write(ID),nl,
        print(State),nl,
        write( ** Message: ),        write(Msg),
        write( ),write(Desc), nl,
        send_message(ID,PID,fail,T2),      !.
```

D. Ordering Algorithm.

The ordering algorithm has two main procedures. The *static rule* is applied when the user program is first read in. It applies mode declarations and builds

something called the static structure. The *dynamic rule* is applied when a clause is used to create an AND process; it uses the static structure to make the initial state of the AND process.

```
%c "Static Rule" -- applicable to all calls to a clause (not
%c affected by pattern of variable instantiation), uses mode
%c declarations to determine which literals can/cannot be used
%c as generators for each variable.  This procedure also creates
%c the numbered head and body used by other parallel AND stuff.

static(Head,Body,NHead,NBody,GenList):-
        modified(Head,Body,IdBody,NHead,NBody,N),
        slots(N,G),
        numbervars(G,0,N),
        use_modes(NBody,G,GenList),     !.
```

This next procedure makes the copy of the body used by AND processes. It turns variables into metavariables (by the call to *numbervars*) and tacks on the #N: prefixes to each literal.

```
%c modified(H,B,B2,NH,NB,N) -- B2 is a copy of body B with
%c literal numbers added, and NH and NB are copies of head H
%c and body B with all vars "numbered" (via numbervars).

modified(H,B,B2,NH,NB,N):-
        number(B,1,B2),
        copy([H,B2],[NH,NB]),
        numbervars([NH,NB],0,N).
```

For each literal in *G*, see if there is a mode declaration; if so, use it to build a piece of the static structure:

```
% use_modes(N,L,Gi,Go) -- for every literal in list L that has a
% mode declaration, add more info to generator list; Gi is
% input list of generators, Go is output, N is current lit #.

use_modes([],G,G).
use_modes([#N:L1|Ln],Gi,Go):-
        L1=..[F|A],             % if there is a mode
        recorded(modes,m(F,M),K),% declaration for F, use it
        infer(N,A,M,Gi,Gt),
        use_modes(Ln,Gt,Go),        !.
use_modes([L1|Ln],Gi,Go):-use_modes(Ln,Gi,Go),        !.

% infer(N,A,M,Gi,Go) -- use mode info M for arg list A to
% update generator list Gi, creating Go, for literal number N

infer(N,[],[],G,G).
infer(N,[A1|An],[?|Mn],Gi,Go):-infer(N,An,Mn,Gi,Go), !.
infer(N,[A1|An],[+|Mn],Gi,Go):-known_not(N,A1,Gi,Gt),
        infer(N,An,Mn,Gt,Go),       !.
infer(N,[A1|An],[-|Mn],Gi,Go):-known(N,A1,Gi,Gt),
        infer(N,An,Mn,Gt,Go),       !.

% known_not(N,A,Gi,Go) -- it is now known that literal N cannot
% be the generator of any var in term A; add N to the rhs of
% every var in Gi, making Go

known_not(N,A,Gi,Go):-vars_in(A,V),
        concat_all(V,N,Gi,Go).

% known(N,A,Gi,Go) -- it is known that literal N is the generator
% for all vars in term A; replace rhs of each var in Gi with N

known(N,A,Gi,Go):-vars_in(A,V),
        replace_all(V,N,Gi,Go).

replace_all([],N,G,G).
replace_all([V1|Vn],N,Gi,Go):-replace(V1,N,Gi,Gt),
        replace_all(Vn,N,Gt,Go),    !.
```

% if N is in the list of known non-generators, it can't be used;
% fail, and hope 'connect' (or whatever) creates another one.

```
replace(V,N,[V:X|G],[V:X|G]):-member(N,X),!,fail.
replace(V,N,[V:X|G],[V:gen(N)|G]).
replace(V,N,[X:Y|Gi],[X:Y|Go]):-replace(V,N,Gi,Go),   !.
```

Next is the code for the "dynamic rule" that is applied when an AND process is first created. It uses *SS*, the static structure for the clause, and information about which variables in the head are bound. The overall goal here is to find generators for the remaining variables by calling on the connection rule or the leftmost rule:

```
dynamic(Head,Body,NHead,NBody,Ss,[gen(G2),pred(Pred),lin(Linear)]):-
        head_gen(Head,NHead,Ss,G1,Used),
        others(Ss,Used,UnUsed),
        connect(NBody,RemBody,G1,G2,Used,Uo,UnUsed,UUo),
        final(G2,NHead,NBody,Linear,Pred).
```

Call is the literal actually used in the procedure call, and *Head* is what was written by the programmer; by comparing the two, we find out which variables are generated by the head; return them in the set *U* (for 'used'):

```
head_gen(Call,Head,Gi,Go,U):-Call=..[F|CArgs],
        Head=..[F|HArgs],
        extract(CArgs,HArgs,U),
        replace_all(U,head,Gi,Go),  !.
```

Now, from the static structure and the set *U* computed above, we can determine *UU*, the set of 'unused' variables that need generators:

```
others([],_,[]):-!.
others([V:gen(L)|R],U,X):-others(R,U,X).
others([V:G|R],U,X):-others(R,U,X1),app(V,U,X1,X),   !.
```

This next procedure is the heart of the dynamic rule. It tries to remove a literal from current body Bi to make a new body Bo. The selected literal is a candidate generator; if *update* accepts it (meaning it doesn't violate any mode declarations in the static structure) then the dynamic strucure is modified to include it; otherwise we backtrack to *choose* to get another literal. We are

guaranteed of finding an acceptable literal eventually, unless the programmer has made a circular mode declaration.

```
% connect(Bi,Bo,Gi,Go,Si,So,UUi,UUo) -- using body Bi and existing
% structure Gi and info about vars unused (UUi) so far, make a new
% structure Go.  Si is the current set of vars we have to connect
% to, UUo will be list of remaining vars, So the vars to connect
% on the next iteration, Bo the remaining literals.

connect([],[],G,G,S,S,U,U):-!.          % all lits used
connect(B,B,G,G,S,S,[],[]):-!.          % no unused vars
connect(Bi,Bo,Gi,Go,Si,So,UUi,UUo):-
        choose(Bi,Bt,Si,UUi,UUt,X),     % select a lit, can use it
        update(X,Gi,Gt,St),  % if no mode violation
        connect(Bt,Bo,Gt,Go,St,So,UUt,UUo).
```

Si is the set of variables we want to connect to. If it is empty, or if the selection of X by the connection rule does not provide any information, then fail, and let the leftmost rule select X:

```
choose(Bi,Bo,Si,UUi,UUo,X):-
        neq(Si,[]),
        connect_all(Bi,Bo,Si,UUi,UUo,X),
        neq(Bo,Bi).
choose(Bi,Bo,Si,UUi,UUo,X):-
        lr(Bi,Bo,UUi,UUo,X).% call leftmost rule . . .
```

Add the final touches to the stored data structure for an AND process: we know generators for all vars in body (this information is in GS), so now figure out linear ordering L and list of predecessors P.

```
final(GS,Head,Body,L,[#head:HP|P]):-
        level(head,down,L,Body,GS,[head]),      % do level order traverse
        predecessors(Body,Body,GS,L,P),  % starting at head
        head_pred(Head,Body,GS,L,HP).
```

```
% a level order traverse in the "down" direction means looking for
% literals that consume vars "already generated"

level(N,down,L,Body,GS,XL):-
        setof(X,desc(N,Body,GS,XL,X),S), % S is set of immed desc
        concat(XL,S,XL2),   % is part of solved list
        level_all(S,down,Ln,Body,GS,XL2),
        conc(S,Ln,L),!.

% level order traverse in "up" direction means closure of
% predecessor relation

level(N,up,L,Body,GS,XL):-
        setof(X,pred(N,Body,GS,X),S),      % set of immediate pred
        rev(S,[],S2),
        level_all(S2,up,Ln,Body,GS,X),
        conc(S2,Ln,L),          !.
```

APPENDIX II

Parallel AND Process Examples

A number of examples of conceivable situations that arise in the backward execution phase in parallel AND processes will be discussed in this Appendix. Each situation, described in terms of the map coloring problem first mentioned in Chapter 5, illustrates a different aspect of the rules for handling nondeterminism in AND processes. The dataflow graph for the example is reproduced here as Figure 22. The discussion continues the notation #N for "literal number N" and ↑N for "the OR process created to solve literal #N".

A. Complete Solution of the Map Coloring Problem

The first example is the complete solution of the map coloring problem, involving the processing of multiple failures.

When the AND process was first created, and after the literals were ordered, the only literal for which an OR process could be started was #1:*next*(*A*,*B*). Eventually, this sent back *success*(*next*(*red*,*blue*)), binding *A* to *red* and *B* to *blue*. Next, processes for the three generators in the middle row of the graph were started. All three succeeded, and as the success messages arriveed, the following occurred:

-- ↑3 sent *success*(*next*(*red*,*blue*)), setting *C* to *blue*. All of the predecessors of literal #5 were then solved, so a process for this literal (at that time *next*(*blue*,*blue*)) was created.

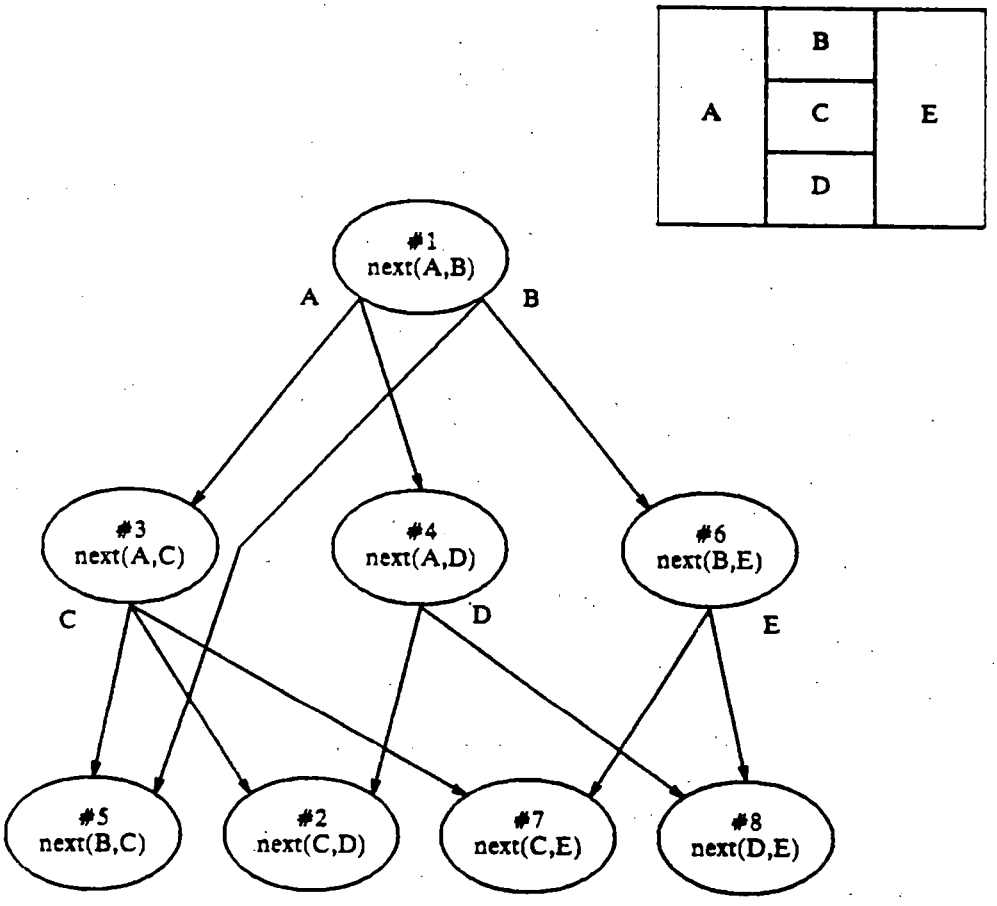
-- ↑4 sent *success*(*next*(*red*,*blue*)), setting *D* to *blue*, enabling a process for #2, *next*(*blue*,*blue*).

-- ↑6 sent *success*(*next*(*blue*,*red*)), binding *E* to *red*. Processes for the remaining two literals, #7 and #8, both *next*(*blue*,*red*), were started.

At this point, the status of the AND process was: literals #1, #3, #4, and #6 solved; literals #2, #5, #7, and #8 pending; failure context empty. The

```
color (A,B,C,D,E) ←
        next (A,B) & next (C,D) & next (A,C) &
        next (A,D) & next (B,C) & next (B,E) &
        next (C,E) & next (D,E).

next(red,blue) ← .      next(red,yellow) ← .      next(red,green) ← .
next(blue,red) ← .      next(blue,yellow) ← .     next(blue,green) ← .
next(yellow,red) ← .    next(yellow,blue) ← .     next(yellow,green) ← .
next(green,red) ← .     next(green,blue) ← .      next(green,yellow) ← .
```



A map with five regions, the coloring problem as a list of eight borders, and the dataflow graph created by the ordering algorithm. This method of solving a map coloring problem in logic was originally used by Pereira and Porto [46] to illustrate intelligent backtracking.

Figure 22. The Map Coloring Problem

processes for literals #2 and #5 are about to send fail messages, while the other two are about to succeed. The transitions explained next describe what happened when the fail message from ↑5 was read first; then the transitions that would have occured if the fail from ↑2 arrived first will be explained. In either case, the AND process will go into the same state eventually. The only difference in the sequences of state transitions is that if the message from ↑2 arrives first, the number of transitions required to reach the state in which the first success is sent is longer.

Case 1: #5 fails first.

The failure context was set to [#5], which is the prefix of the redo list [#5,#3,#1]. The suffix is [#3,#1], and a redo message was sent to ↑3. The literals to the right of #3 in the linear ordering were

-- #4: Reset.

-- #6: Reset.

-- #2: Canceled (will be replaced when new C arrives from ↑3).

-- #5: Already terminated.

-- #7: Canceled (will be replaced when new C arrives).

-- #8: Replaced with new process (since D, E modified by resets).

Again, as an implementation detail, the reset of a generator that has sent only one value really has no effect, and the replacement of a process such as that for literal #8 can be avoided when its variables do not change values. The state of the AND process after this transition: literals #1, #4, and #6 solved; literals #3 and #8 pending; literals #2, #5, and #7 blocked; failure context [#5].

The fail message from the original process for literal #2 then arrived. Since that process was canceled, this message was ignored. The successes from the original processes for #7 and #8 arrived, and they also were ignored. Note that even though there is a process for #8 at this time, it has a different ID than the original process. The AND process always ignores messages from processes it has

canceled.

The success from #3 arrived, with *next(red,yellow)*, binding C to *yellow*. New processes for #2, now *next(yellow,blue)*, and #5, now *next(blue,yellow)*, and #7, now *next(yellow,red)*, were created. Since there is a new process for #5, it was removed from the failure context. The state of the AND process: literals #1,3, #4, and #6 solved; literals #2, #5, #7, and #8 pending; failure context [].

All of the pending processes sent success messages; the order is irrelevant. In particular, note that ↑8 could have sent its success message before the success from ↑3 in the previous paragraph. After the last was received, the AND process sent its parent the message

success(color(red,blue,yellow,blue,red))

Case 2: #2 fails first.

When the state of the AND process had literals #1, #3, #4, and #6 solved, with the remaining literals pending and an empty failure context, two fail messages were on the way. The next sequence of transitions shows how the failure from ↑2 would be handled. This sequence involves multiple failures.

The fail message from ↑2 arrives, the failure context is set to [#2], the prefix of the redo list [#2,#4,#3,#1]. ↑4 is sent a redo message, and then the remaining literals in the linear ordering are

-- #6: Reset.

-- #2: Already failed.

-- #5: Not affected.

-- #7: Canceled (since E was reset), replaced with literal that has same values for variables.

-- #8: Canceled, will be replaced when new D arrives from ↑4.

The failure context is [#2]; solved literals are #1, #3, #6; pending literals are #4, #5, #7; and #2 and #8 are blocked.

The fail message from ↑5 arrives. #5 is appended to the failure context, making [#2,#5]. This does not match any redo list. The processing of this failure is postponed until the failure context is reset to the empty list. The state remains the same, except #5 is now blocked and not pending as before.

A success from the process for #7 will arrive (either now or after the success from #4; either way, it has no effect on what follows). *success*(*next*(*blue*,*yellow*)) arrives from ↑4. Start processes for #2, *next*(*blue*,*yellow*), and #8, *next*(*yellow*,*red*). Remove #2 from the failure context, which becomes the empty list. There is one postponed failure, from the original process for literal #5. This process was never canceled during the backward execution on behalf of #2, so a failure context is created for it now. From this point, the AND process behaves as if it had just received the fail from ↑5: the failure context is [#5], the matching redo list is [#5,#3,#1], ↑3 is sent a redo message, and the literals to the right of #3 are:

-- #4: Reset (D is once again *blue*).

-- #6: Reset (E is still *red*).

-- #2: Canceled, will be replaced when new C arrives from ↑3.

-- #5: Already canceled.

-- #7: Canceled, will be replaced when new C arrives.

-- #8: Canceled (since D, E reset), replaced by new process for with original D and E.

The state of the AND process is now: literals #1, #4, and #6 solved; literals #3 and #8 pending; literals #2, #5, and #7 blocked; failure context [#5]. The current values of the variables are A=*red*, B=*blue*, C unbound, D=*blue*, and E=*red*. Note that this is the same state as earlier (in case 1), when #5 and #2 were failures and the fail message from #5 arrived first.

B. Parallel Processing of Failure Contexts

The previous example showed how an AND process resolves conflicts in the handling of fail messages, by postponing the handling of a fail message if the

corresponding literal is not in the current redo list. This strategy can lead to extra work, as in the second case: while the AND process was in the sequence used to handle the failure of #2, it was doing work that would later be undone when the fail message from #5 was processed.

In that example, there is enough information to decide immediately that the failure of #5 should take precedence. The redo sequence for #5 is [#5,#3,#1] and the redo sequence for #2 is [#2,#4,#3,#1]. Comparing the second elements (those literals that will be sent the redo messages), one can see that the failure of #5 causes #3 to be redone, while the failure of #2 causes #4 to be redone. Since, according to the linear ordering being used, #4 is reset when #3 is redone, it makes sense to give precedence to the redo sequence involving #5. In other words, maybe the AND process could abort the failure context for #2, and immediately start backward execution for #5 when that fail message arrives.

The reasoning used in the above example does not work for the general case, however. There are situations that arise, based on unpredictable timing sequences, that show why all fail messages have to be saved and processed eventually. The only time a fail message can safely be ignored is when the process that sent it is one that has benn explicitly canceled earlier, *i.e.* the message is from a process that sent a fail message just before it would have read a cancel from its parent.

This next example is an illustration of such a situation. This example is again based on the map coloring dataflow graph, but this time assume the literals on the "bottom" of the graph are not calls to ***next***, but calls to some other procedures *p*, *q*, *r*, and *s*. Again, assume the "top" four literals have been solved, binding variables *A* through *E* to their first values. The AND process is in a state where it is waiting for messages from processes of literals #2, #5, #7, and #8.

Suppose literal #7, *r*(*red*,*blue*), is a failure. Then literal #6 is sent a redo, and ↑8 is canceled. Next assume the the generator of *E* succeeds again, now binding *E* to *green*. New processes for literals #7 and #8 are created, and suppose both are successes. All during the backward execution just described, no

messages were received from ↑2 or ↑5.

Now, finally, suppose a fail message arrives from ↑2. As usual, the process for #4 is sent a redo, and

-- #6: Reset (E is again bound to *red*).

-- #2: Already canceled.

-- #5: Not affected.

-- #7: Canceled, replaced by *r(blue,red)*.

-- #8: Canceled, will be replaced when #4 sends new D.

The current state of the AND process is: literals #1, #3, and #6 solved; literals #4, #5, and #7 pending; literals #2 and #8 blocked, awaiting the success of #4; failure context [#2]. In addition, there is one used and one unused answer from ↑6, the generator of E. Note that literal #7 is back to its original state, namely it consumes the first values of C and E. It will fail. If that fail message arrives when the AND process is in the state just described, and the AND process attempts to process the failure contexts in parallel, the following situation arises.

The fail context is set to [#2,#7], which does not match any redo sequence. A conflict in failure contexts is created. If the conflict is resolved by seeing which redo list has a higher precedence, the sequence for #2 is selected, since the redo sequences are [#2,#4,#3,#1] and [#7,#6,#3,#1], and #4 takes precedence over #6. In the hypothetical parallel processing of failure contexts, the fail message from #7 would be ignored, which is a mistake. The state of the AND process would then be: literals #1, #3, and #6 solved; #4, #5, and #8 pending; #2 blocked, awaiting a success from #4; and #7 failed, with no rules for starting a new process for it. #4 is not a predecessor of #7, so a success from #4 does not cause a new process for #7 to be started.

The above argument is admittedly "attacking a straw man," as the scenario was carefully contrived to make a particular point. There may well be a method for processing failure contexts in parallel. But the example at least illustrates the subtle errors that may occur as a result of the relative timing of messages from descendants. For the present, at least, parallel AND processes will save all fail

messages, and start backward execution for them only after completely processing the current failure context.