The University of Alberta

An Execution Model for some Parallel Logic Programming Languages

by

LAW, Chung Sea

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1987

ISBN 0-315-37769-0

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: LAW, Chung Sea

TITLE OF THESIS: An Execution Model for some Parallel Logic Programming Languages

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1987

(Signed) [illegible]

Permanent Address:
18, 4th Floor, Block B,
237A To Kwa Wan Road,
Kowloon,
Hong Kong.

Dated Jan 12, 1987

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **An Execution Model for some Parallel Logic Programming Languages** submitted by **LAW, Chung Sea** in partial fulfillment of the requirements for the degree of Master of Science.

T. A. Marsland

Supervisor

Jia-huai You

[illegible]

[illegible]

Date Jan 12, 1987

To My Parents

ABSTRACT

The exploitation of parallelism in logic programs has recently been an active research topic in logic programming, especially after the announcement of the Japanese Fifth Generation Project. The research includes two areas: the development of parallel logic programming languages, and the design and realization of parallel execution models for logic programs. This thesis studies the parallel logic programming languages Concurrent Prolog, PARLOG and Guarded Horn Clauses (GHC), and some of the most recently proposed parallel execution models for logic programs. An execution model for those languages, concentrated on the control of parallel activities, is proposed. The model is targeted for implementation on a shared memory architecture, where moderate parallelism is exploited. The emphasis of the design is on ease of realization and low overheads.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my supervisor, Prof. T. A. Marsland, for his guidance, assistance, criticisms and encouragement throughout the research and preparation of this thesis.

I am grateful for the careful reading and useful comments provided by the members of my examination committee: Dr. J.-H. You, Dr. P. Rudnicki and Mr. W. Joerg.

Financial support from NSERC and the Department of Computing Science in the form of teaching and research assistantships are gratefully acknowledged.

Special thanks to all my friends, especially C. A. Wang and Larry W[illegible] for their encouragements during my stay here in Edmonton.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter is primarily a description of the motivation and objectives of the research, and an overview of the basic concepts in *logic programming* and the programming language *Prolog*. Since there are many good references to the above topics but they contain a slight variance in terminologies, the overview is only briefly provided to allow the thesis to be self-contained. Interested readers are referred to [CIT82, Kow74, Kow79a, Llo84] for details in logic programming, and [Che84, ClM81, Fer81, War77] for details in Prolog. Further references will be quoted when appropriate.

1.1. Motivation and Objectives

1.1.1. Motivation

Being based on symbol logic which was originally devised to clarify human thoughts, logic programming languages are claimed to be particularly human-oriented. Their once un-popularity may be attributed to their low performance on conventional computers, and the heavy investment in and established programming style of existing programming languages. However, this situation has drastically been changed by the *Japanese Fifth Generation Project* where one of the objectives is the building of an innovative high-speed inference machine based on logic programming [MoK85], and the various responses from North America and Europe ([Bis86], Chapter 3).

A most appealing characteristic of logic programs is their admissibility of parallelism. However, improving logic programs' execution speeds by incorporating parallelism is not an easy task. A proposed *parallel execution model* should at least be justified by the following factors: target architecture(s), forms of parallelism supported, potential of parallelism, incurred overheads and ease of realization.

Many parallel execution models have recently been proposed. Most of them are parts of some ambitious long-range projects aiming at exploitation of massive parallelism in logic programs (see Chapter 3). These projects encounter various problems, such as the design of the proper supporting architectures, the effective control of overheads, etc, that must be satisfactorily solved before actual implementations can be realized. In view of this, it seems more sensible for a short-term project to start small. My idea is that a good starting point is to support some simple languages on a moderately-parallel, readily-available architecture. There have recently been some proposals of parallel logic programming language, most notably *Concurrent Prolog*, *PARLOG* and *Guarded Horn Clauses* (GHC), which are intended to support concurrent programming and parallel execution. The efforts spent to design an execution model for them that aims at exploiting moderate parallelism, incurs low overheads, and is relatively easy to realize are highly justifiable.

1.1.2. Objectives

The research in exploiting parallelism in logic programs involves two non-complementary areas: the development of parallel logic programming languages, and the design and realization of parallel execution models for logic programs. The latter area can be further divided into two major problems: (1) the control of parallel activities, and (2) the management of *binding environments* (i.e. the different variable bindings generated in the computations).

The work attempted in this thesis can be divided into two parts. The first part (Chapters 2 and 3) is the study and analysis of some parallel logic programming languages and parallel execution models for logic programs. The second part (Chapter 4 and 5) is the proposal of an execution model for the parallel logic programming languages Concurrent Prolog, PARLOG, and GHC. The proposed model is strongly influenced by Levy's work [Lev86], but incorporates some extensions and modifications (see Chapter 4). The proposal concentrates on controlling parallel activities, while a brief discussion of the problems in managing the binding environments is provided in Chapter 6. The target architecture of the proposed model is a shared memory multiprocessor system. The emphasis is on ease of realization and low overheads. The objectives are:

(1) To investigate some of the data structures and algorithms needed to support the model.

(2) To investigate the effects of the target architecture over the design decisions of the model.

(3) To justify the model on the abovementioned factors, especially the forms of parallelism supported, incurred overheads and ease of realization.

(4) To provide a basis for further research and implementation.

1.2. Logic Programming

Logic programming is based on the ideas of the statements: "logical inference is computation" and "logic can be a programming language" [Kow74]. It is the outgrowth of research in automatic inference, in particular *resolution* [KoK71, Rob65] which is a generalization of modus ponens coupled with *unification* [Nil80], a powerful pattern matching operation. Special features such as *non-determinism*, the flexibility of *logical variables*, and the close intimacy of operational and declarative semantics make logic programming a promising tool for problem solving.

1.2.1. Syntax

The building boxes of a logic program are *terms*. A term is either a constant (number, name, character, etc), a variable, or a composite term.[1] A composite term acts as a complex data structure or a function in logic programs. It takes the form:

[1] In this thesis, the syntactic conventions of DEC-10 Prolog are followed: constants begin with a lower-case letter, variables begin with an upper-case letter. The term [X|Y] denotes a list whose head (CAR) is X, and tail (CDR) is Y. The constant [] denotes an empty list.

$$f(t_1,t_2,...,t_m) \qquad m \geq 1.$$

where f is a name called the *functor* and each argument t_i is a term.

An *atom* is either a name or a composite term. The name or functor (of the composite term) is called the *predicate symbol*. Predicate symbols must occur at the top level, i.e. they cannot appear inside any composite term.

A *Horn clause* is either a *program clause* or a *goal clause*. All the variables in a Horn clause are universally quantified. The scope of a variable is confined to the clause containing it, i.e. two variables with the same name but in different clauses are independent. A program clause has the form:

$$H <\!- B_1,B_2,...,B_n. \qquad n \geq 0. \qquad [PC]$$

where the H and B_i's are atoms. The H is called the *head* of the clause and the B_i's collectively are called its *body*. The symbol "<-" can be omitted if the body is empty, i.e. when n=0. The set of program clauses whose heads have the same predicate name and *arity* (i.e. number of arguments) is sometimes known as the procedure for that predicate symbol. A goal clause has the form:

$$<\!- G_1,G_2,...,G_m. \qquad m \geq 0. \qquad [GC]$$

where the G_i's are atoms called *goals*. It is called an *empty clause* if m=0. In this thesis, the term "program clause" is sometimes simplified to just "clause"; a clause is sometimes known as an "assertion" if its body is empty, or a "rule" otherwise.

A *logic program* is a finite set of program clauses. Figure 1.1 is a sample logic program.

```
[C1] father(john,tom).
[C2] father(tom,sam).
[C3] grandfather(X,Y) <- father(X,Z),father(Z,Y).
```

Figure 1.1 A Sample Logic Program

1.2.2. Semantics

Informally, the program clause PC in Section 1.2.1 has the declarative reading: H if B_1 and B_2 and ... and B_n. The goal clause GC has the reading: the conjunction of the G_i's is false. If GC is an empty clause, it denotes a contradiction.[2]

The formal semantics of logic programs were investigated by van Emden and Kowalski [vaK76]. They considered operational semantics and two kinds of declarative semantics: *model-theoretic* and *fixpoint*. In the operational semantics, the denotation of a n-ary predicate symbol p is the set of n-tuples $<t_1,...,t_n>$ such that $p(t_1,...,t_n)$ is provable given the program as the set of axioms. In the model-theoretic semantics, the denotation of p is the set of n-tuples $<t_1,...,t_n>$ such that $p(t_1,...,t_n)$ is logically implied by the program. In the fixpoint semantics, the denotation of p is derived from the program through a transformation that maps clauses into ground clauses, from which tuples of ground terms are formed.[3] These three kinds of semantics were shown to be equivalent to each other.

In practical programming we ignore formal semantics, contend with the intuitive idea of truth and write programs which are true in the intended meanings. For example, the intended meanings of the program in Figure 1.1 are:

[2] The proof procedure in logic programming is based on resolution which is a *refutation* system. This means that the procedure proves a formula by including its negation into the set of axioms (i.e. program clauses) and tries to derive a contradiction. The goal clause is in fact the negation of the formula we want to prove, i.e. "for some variable bindings, the conjunction of G_i's is true" (see Section 1.2.4).

[3] A clause or term is ground if and only if it contains no variable.

(1) john is tom's father.

(2) tom is sam's father.

(3) for all X and Y, if X is the father of some Z and Z is the father of Y, then X is the grandfather of Y.

1.2.3. Unification

Two atoms G and H are unifiable if there exists a substitution θ such that $G.\theta = H.\theta$. The *unifier* (i.e. unifying substitution) θ should be the *most general unifier* [Rob65] of G and H. A substitution is a set of variable bindings. $G.\theta$ means applying the substitution θ to the term G. For example:

```
G:      father(X,Y)
θ:      {X=john,Y=tom}
G.θ:    father(john,tom)
```

In unifying a goal G with a clause head H (i.e. *head unification*), the unifying substitution is often classified into two disjoint sets. The *input substitution* is the set of bindings to variables in H, and the *output substitution* is the set of bindings to variables in G. For example:

```
goal:                  father(X,tom)
clause head:           father(john,Y)
input substitution:    {Y=tom}
output substitution:   {X=john}
```

1.2.4. Computation Model

A logic program P is invoked by a goal clause G which has the form:

$$<\!-\ G_1, G_2, \ldots, G_m. \qquad m \geq 0.$$

The computations are to derive an empty clause using G and P, i.e. to prove that an instance of ~G is implied by P. It is a sequence of *goal reduction* steps:

(1) Arbitrarily selecting a goal G_i, $1 \leq i \leq m$, in the goal clause G.

(2) Nondeterministically choosing from P a program clause C of the form H <- $B_1,...,B_k$, $k \geq 0$, where H and G_i are unifiable via a substitution θ.

(3) Using C to *reduce* G. The reduced G is G with G_i replaced by the C's body (which may be empty) and the substitution θ applied throughoutly.

(4) Repeating the goal reduction step (i.e. steps 1 to 3) until either G is empty (i.e. the proof succeeds) or G is not empty but no clause C can be found in step 2 (i.e. the proof fails).[4]

For example, given G as <- grandfather(Who,sam) and P as the program in Figure 1.1, Figure 1.2 shows the computations invoked by G. Note that the proof procedure is *constructive*, i.e. not only can it prove the existence of an instance of ~G implied by P, but that particular instance can also be found by constructing the bindings to the variables in G using the substitutions generated in the sequence of goal reduction steps. In this example, since Who=X=john, the instance of ~G implied by P is grandfather(john,sam).

$$\begin{tikzcd}
\text{<- grandfather(Who,sam)} \arrow[d, "\text{apply C3: Who=X,Y=sam}"] \\
\text{<- father(X,Z), father(Z,sam)} \arrow[d, "\text{apply C1: X=john,Z=tom}"] \\
\text{<- father(tom,sam)} \arrow[d, "\text{apply C2}"] \\
\text{<-}
\end{tikzcd}$$

Figure 1.2 A Resolution Proof

[4] Since first-order logic is *semidecidable* ([Nil80], pages 144-145), nonterminating computation may happen in some cases.

1.2.5. Non-determinism

A distinguished feature of logic programming is its non-determinism. Two sources of non-determinism are apparent in the computation model in 1.2.4: any goal can be selected in step 1, and any applicable clause can be chosen in step 2. The latter determines which solution is computed, the former only affects the system's behavior.

A system may have different attitudes in dealing with non-determinism. It features *don't-know nondeterminism* if it doesn't know how to obtain a solution. An example is Prolog (see Section 1.3) which relies on its *backtracking* facility to explore the search space. It features *don't-care non-determinism* if it doesn't care how solution is obtained, hence no or a fewer amount of search is needed. Examples are the languages discussed in Section 2.2 whose *commitment* mechanism non-deterministically chooses a path to a single solution.

1.2.6. The Logical Variable

Another important feature of logic programming is the logical variable. Because of the flexibility of unification, a variable can be partially instantiated by an atom, and subsequently further instantiated by other atoms. This nature of the logical variable is heavily exploited in logic programs, and it allows logic programming to be an effective tool in many practical applications (e.g. the examples in [Sha83, War80]).

1.2.7. AND-OR Tree

The execution of a logic program can be conceptually viewed as a search in an *AND-OR tree* [Nil80] based on resolution proof procedures. The OR-branches of an AND-OR tree lead from a goal to its applicable clauses, the AND-branches tie the atoms of a clause together. Figure 1.3 is the AND-OR tree solving grandfather(Who,sam) using the program in Figure 1.1. Note that the bindings to the shared variable Z must be consistent, Who=john is the solution.

A logic program is *non-deterministic* if a goal clause can have more than one solution (i.e. the AND-OR tree has many active OR-branches). It is *deterministic* if a goal clause has at most one solution.

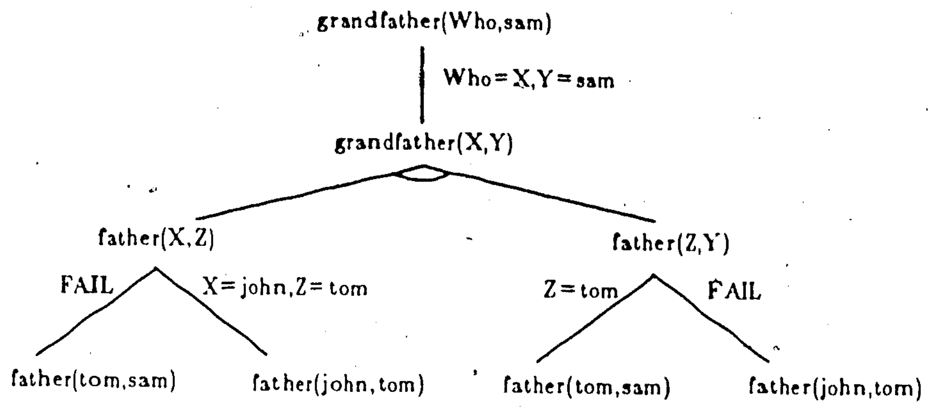


Figure 1.3 A Sample AND-OR Tree

1.3. Prolog

Prolog was the first [BaM73], and is still the most widely used, logic programming language. Standard Prolog implementations use the order of goals in the goal clause and the order of clauses in the program to control the search in the AND-OR tree. The non-determinism is simulated by sequential search and backtracking. In each cycle, the system chooses the left-most goal as the current goal and sequentially tries the applicable clauses in their textual order. If a goal cannot be reduced, the system backtracks to the most recently reduced goal with untried clauses, makes it the current goal and tries its next untried clause. This results in a left-right depth-first search which allows the AND-OR tree to be collapsed into a stack of records[5] (each represents one head unification) that can be space-efficiently implemented [Bru82].

[5] In fact, an actual Prolog implementation uses three stacks: the *environment stack*, the *global/copy stack* and the *trail stack*. The trail stack is used to undo global effects in backtracking. The environment stack holds control and binding information. Some binding information which

The growth and shrinkage of the tree are accomplished by the push and pop operations of the stack. However, Prolog's search strategy is unfair and incomplete; solutions to the right of an infinite branch in the AND-OR tree will never be discovered.

Pure Prolog is non-practical and cannot exploit the knowledge built into many programs. Because of this, most Prolog implementations incorporate the following extensions.

1.3.1. Cut

Cut (!) is a widely used and controversial control facility in Prolog. It stops the system from choosing untried clauses for an atom in backtracking and hence allows the programmer to control the search by pruning unwanted OR-branches from the AND-OR tree. For example, consider the following:

```
<- b.           [goal clause]
b <- c,!,d.     [C1]
b <- e,f.       [C2]
c <- ...
   .
   .
   .
```

C1 is first applied to the goal b. Suppose c is provable, the effect of executing the cut will commit the system to C1, the system will not try C2 (since it is pruned) even if the proof of d fails. Cut is dangerous since undisciplined usage of it will introduce incompleteness into the system; as a search space containing solutions may be accidentally pruned.

has a longer life span is placed in the global/copy stack so that space in the environment stack can be reclaimed more frequently. See [War77].

1.3.2. Other Built-in Predicates

Prolog, like many other declarative programming languages, provides a metalevel programming facility by which a program can construct and evaluate other programs. The system built-in predicate "call" accepts a goal as argument and evaluates it when invoked (see Section 1.3.3 for an example).

Side effects such as the ability to perform input/output are required in practical Prolog programs. The system built-in predicates such as "read", "write", "add_axiom" and "delete_axiom" are introduced for these purposes. The latter two predicates allow Prolog to change the *clause database* (i.e. the logic program) at run-time, thus endowing the program with self-modifying capability.

Other evaluable predicates are provided to perform useful functions such as arithmetic operations. The built-in predicate "fail" is a control facility used to simulate a failure in the proof.

1.3.3. Negation As Failure

The needs to specify negative information in the clause body naturally arise in practical Prolog programs. Prolog supports this by implementing the *negation as failure* (NAF) rule [Cla78]. Basically the NAF rule gives a procedural interpretation to negation: if a system fails to prove an atom X, then it infers not(X). The rule can be implemented as the following Prolog program:

```
[C1] not(X) <- call(X),!,fail.
[C2] not(X).
```

The correctness of the above program depends on Prolog's backtracking facility and the textual order of the clauses. C1 tries to prove X (instantiated to a goal) using the metalevel facility "call" and reports failure if X is provable (i.e. then not(X) is false). C2 is a catch-all clause. It is invoked when call(X) fails (i.e. X is not provable, so not(X) is true).

1.4. Summary

We have described the basic concepts in the field of logic programming and introduced its first practical outcome, the programming language Prolog. Prolog's rigid search strategy, while resulting in a space-efficient implementation on [illegible] von Neumann machine, usually forces the programmers to use odd tricks to improve execution speed. There are some attempts to improve the performance of Prolog programs, such as the provision of various control strategies in *IC-Prolog* [CMG82] and the research in *intelligent backtracking* [BrP84, Cox84, KuL86]. While these attempts (especially the latter) have obtained some achievements, performance improvement within the constraints of sequential processing cannot be expected to be large.

Besides using sequential search and backtracking, the non-determinism in logic programs can also be simulated by parallel search in the AND-OR tree. The approach of improving performance by incorporating parallelism has a high potential, and is reinforced by the advances in hardware technologies and VLSI techniques. The main questions are: What forms of parallelism can be exploited in logic programs? What are the desired features of a parallel language within the formalism of logic programming? These are the topics to be covered in Chapter 2.

Chapter 2

Parallel Logic Programming Languages

In this chapter, we describe the forms of parallelism in logic programs which may be usefully exploited. Then we focus on some recently proposed parallel logic programming languages. These languages incorporate special features that are designed particularly to uncover the expressive power of parallel programming languages embodied in the logic programming formalism.

2.1. Parallelism in Logic Programs

A pure logic program has no restriction on execution order, this separation of control from logic [Kow79b] admits flexibility in execution which allows the exploitation of parallelism. Conery and Kibler classified four forms of parallelism[6] in logic programs: *AND-parallelism*, *OR-parallelism*, *stream parallelism*,[7] and *search parallelism* [CoK81]. Their natures and some proposals of their exploitation are described in this section.

2.1.1. AND-Parallelism

AND-parallelism refers to the concurrent execution of more than one atom in a conjunction. The major difficulty of implementing AND-parallelism is the problem of shared variables; not only must the constituent atoms in the conjunction be *solved* (i.e. reduced to an empty clause), but their generated bindings for a shared variable must also be consistent.

If the constituent atoms share no variable (or all shared variables are bound to ground terms before invocation), so that they are independent, then there is no

[6] Parallelism in unification, which is in a lower level than the forms of parallelism mentioned here, is not discussed in this thesis.

[7] A *stream* is a sequence of partial approximations to a data structure, each approximation is a further extension of the previous one. For example, the following is a stream: [2|Z], [2,3|Z'], [2,3,4,|Z''], [2,3,4].

difficulty in evaluating them in parallel. The concurrent evaluation of such independent atoms is sometimes known as *independent (or restricted) AND-parallelism* [DeG84].

For non-deterministic programs, there are several effective ways to exploit AND-parallelism. The first way is to compute the solutions of each constituent atom independently, and then perform *JOIN* operations [Mai83] on the shared variables to eliminate solutions containing inconsistent bindings. This scheme has a high potential for parallelism, but is not practical in conventional computers because of the expensive JOIN operations and the wasted efforts in computing the solutions containing inconsistent bindings. However, it is proposed for a Prolog implementation on a high-speed, VLSI-based, tree-structure architecture [TLM84] and seems to be used in a dataflow implementation of the OR-parallel subset of ICOT's KL1 (*Kernel Language 1*) [KKI85].

A second way is to have the constituent atoms working concurrently on different solutions. One variant is an extension of the Prolog control strategy. To understand this scheme, consider the conjunction:

p1(X), p2(X).

To introduce parallelism, after p1 passes the first solution to p2, it immediately proceeds to search for other solutions, even before they are required by p2. For the pre-searched solutions, either they are buffered in p1 until they are required, or a new invocation of p2 is created and evaluated for each of them. The latter scheme apparently has a greater parallelism potential but also higher overheads [TaK84].

Another variant is not to use Prolog's control strategy, but to impose a partial order in the evaluation of the constituent atoms. For example, in the *AND-OR process model* [Con83], among the atoms sharing a variable, one of them is designated the producer and the others are consumers. Bindings for shared variables are sent from producers to their consumers. An atom can start evaluation if it has no producer, or it has received all the bindings from its producers. If the evaluation fails, the system, based on some heuristic rules, selects one of the atom's producer and requests the transmission of a new binding. The system reports failure if the failed atom has no producer, or all its producers are unable to generate new bindings. AND-parallelism in this scheme is introduced by the concurrent evaluation of independent atoms and the pre-search for solutions by the producers.

The above schemes are not effective in exploiting parallelism for deterministic programs. For these programs, the exploitation of stream parallelism, described in Section 2.1.3, should be considered.

2.1.2. OR-Parallelism

OR-parallelism refers to the concurrent invocation of applicable clauses in solving an atom. It is related to breadth-first search and tends to be more complete than Prolog's control strategy. Since OR-branches of the AND-OR tree are traversed concurrently, even solutions to the right of an infinite branch can be computed.

Because of the possible exponential fanout of OR-branches in an AND-OR tree, OR-parallelism may exhaust system resources rapidly and hence must be controlled. This involves employing an appropriate scheduling algorithm to control the traversal of the search space, and adopting a richer control language to prune the unwanted subspaces [CiH84].

For systems featuring committed (don't care) non-determinism, a limited form of OR-parallelism is still possible. In those systems, the *guards* of all the alternative clauses are evaluated in parallel, if one of them succeeds, the system commits to that clause and eliminates all its siblings (see Section 2.2). This *committed OR-parallelism* is an OR-parallel search for a single solution, where full OR-parallelism is usually used to compute all solutions concurrently.

2.1.3. Stream Parallelism

Stream parallelism allows the concurrent evaluation of atoms in a conjunction that share a variable *while working on a single solution*. The concurrency is enabled by the stream, i.e. the partial results that are incrementally communicated through bindings to the shared variable. For example, in the conjunction:

```
create(List),member(X,List)
```

the atom "member" can start testing membership of X in List while "create" is constructing and passing to it the stream of partial bindings for List.

The implementation of stream parallelism depends heavily on how variable bindings are maintained and how execution of atoms are synchronized. Usually some control constructs should be provided in the language for synchronization and stream direction determination (see Section 2.2). The exploitation of stream parallelism is only practical for deterministic programs or programs with don't-care nondeterminism, where a partial result is not subsequently retracted, and thus can be immediately conveyed to other atoms.

2.1.4. Search Parallelism

Search parallelism refers to the ability to search the clause database (i.e. the logic program) concurrently. There are two variants: concurrent search for clauses whose heads have the same predicate name, and those whose heads have different predicate names. The former supports OR-parallelism, the latter supports AND-parallelism and stream parallelism.

Most proposals for exploiting search parallelism is the division of the clause database into separate components locating on different memory modules, and initiating searches simultaneously in these components. Examples of these proposals are the systems described by Warren, Taylor et al. [TLM84,WAD84]. The problems they encountered usually fall into the realm of database theory, such as finding the ideal way to divide the clause database, and controlling communication and computation costs (in a distributed database).

2.2. Some Parallel Logic Programming Languages

Because of the separation of logic from control, a logic program basically needs no special language feature for parallel execution. For example, a parallel language can be defined by replacing Prolog's left-right depth-first search strategy with a parallel control strategy without altering Prolog's syntax. However, in the light of the principle stated by Warren:

"Logic programming is programming"

and a special case of the dictum mentioned in Gregory's thesis ([Gre85], page 21):

logic programming language = logic programming formalism
+ control strategy

control strategy = evaluator + control language

it is desirable to provide some control constructs in a language so that the programmer can have a greater control over program execution, and the evaluator can be simpler and thus behaves more predictably.

Also, parallel programming is more than attempting to parallelize program execution; it must have the ability to respond in real-time to multiple events that occur concurrently. An ideal parallel programming language should be capable of expressing important concepts such as the forms of *concurrency*, the type of *communication, synchronization* of concurrent actions, and *indeterminacy* for time-dependent behavior. Shapiro claimed that the logic programming formalism embodies all these mechanisms, the problem is how to uncover them ([Sha83], page 9).

Recent researches have led to the development of three parallel logic programming languages: Concurrent Prolog, PARLOG and GHC. All of them are endowed with the abovementioned expressive power but in different ways. As a prologue, Figure 2.1 lists the important parallel computational concepts and their counterparts in these languages.

Concepts	**Counterparts in Parallel Logic Languages**
Process	Goal
System of processes	Conjunction of goals
Process state	Value of arguments in goal
Process computation	Goal reduction
Concurrency	AND-parallelism
Indeterminacy	OR-parallelism
Communication	Shared variables
Synchronization	Read-only variable in Concurrent Prolog, Mode declaration in PARLOG, Semantics of guards in GHC

Figure 2.1 Parallel Computational Concepts and Their Counterparts

2.2.1. Concurrent Prolog

Concurrent Prolog was proposed by Shapiro [Sha83]. It and its predecessors, the *Relational Language* [ClG81] and the language developed by van Emden and de Lucena [vad79], in addition to Kowalski's procedural reading [Kow74], give a ***behavior reading*** to logic programs.[8]

[8] For a program clause:

$$H <- B_1, B_2, \ldots, B_n. \qquad n \geq 0.$$

In addition to its declarative reading: H is implied by B_1 and B_2 and ... and B_n, Kowalski suggested the procedural reading: to execute the procedure H, perform the procedure calls $B_1, B_2, \ldots, B_n$.

2.2.1.1. Syntax

A Concurrent Prolog program is a finite set of *guarded clauses* each of which has the form:

$$H <\text{-}\ G_1,\ldots,G_m \mid B_1,\ldots,B_n. \qquad m,n \geq 0.$$

where the H (clause head), G_i's (guard goal) and B_i's (body goal) are atoms. The G_i's collectively are called the *guard.* The B_i's collectively are called the *clause body.* When the guard is empty the *commit operator* "|" can be omitted. A clause may contain variables trailed with "?", such as X?; these variables are called the *read-only variables.*

2.2.1.2. Semantics

In the declarative reading, the commit operator is treated as an "AND" operator, i.e. the program clause reads: H if G_1 and ... and G_m and B_1 and ... and B_n. Operationally, a goal reduction step is depicted in Figure 2.2. It has the following meanings:

(1) Unify: to reduce a goal, the head unifications of all the applicable clauses are tried in parallel. Each of which may result in success, suspension (see below), or failure.

(2) Guard spawn: the guards of the clauses with successful head unification are created and evaluated.

(3) Commit: the first clause whose guard is successfully solved is the *candidate (i.e. committed) clause*, the goal commits to it and eliminates all its siblings.

(4) Body spawn: finally, the goal is replaced by the committed clause body.

Under this reading, a program clause is a procedure definition, a goal is a procedure call, and unification provides a unique mechanism for parameter passing, variable assignment, and data access and construction.

In the behavior reading, the program clause reads: a process H can replace itself by the system of processes containing B_1, B_2, ..., B_n. A process terminates by replacing itself with an empty system. Under this reading, a goal is analogous to a process, a conjunction is analogous to a system of processes, shared variables act as communication channels among processes, and unification, in addition to the abovementioned functions, also provides a mechanism for message sending and receiving.

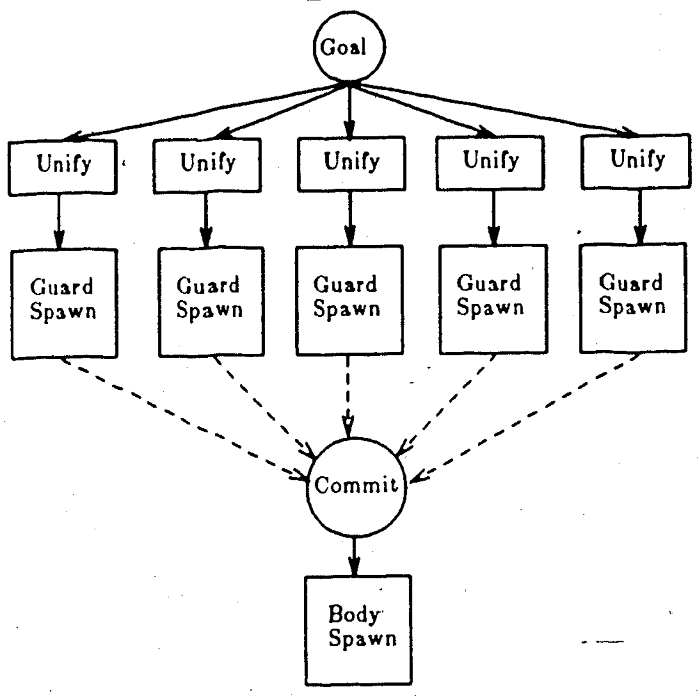


Figure 2.2 A Typical Goal Reduction Step

A unification is suspended if it requires the instantiation of a read-only variable. It is re-activated until the variable causing the suspension is instantiated by other computations. In most implementations, this suspension mechanism is realized by associating a *suspension list* to each variable. A goal suspended on a variable is added into its suspension list. When a variable is instantiated, the goal(s) in its suspension list are de-linked and re-activated. For example:

```
<- p(X), q(X?).

[C1] p(1).
[C2] q(1).
```

When the goal clause is activated, both the goals p and q try their head unifications in parallel. However, q cannot proceed since its head unification tries to instantiate the read-only variable X? to 1. As a result, q is added into X's suspension list and is re-activated until p has committed to C1 and instantiated X to 1.

A clause is not allowed to bind any goal variable before commitment. For example:

```
<- p(X), q(X?).

[C1] p(1) :- guard1 | body1.
[C2] q(1) :- guard2 | body2.
```

As before, the head unification of q will cause q to be suspended on X. The head unification of p binds X to 1. However, since C1 has not yet committed, this binding is locally recorded in C1's environment.[9] When guard1 is solved, C1's binding for the goal variable X is unified with that in the goal clause's environment, only if this *environment unification* succeeds will p commit to C1. After C1's commitment, the binding X=1 is visible in the goal clause, only then can the suspended goal q be re-activated.

To support Concurrent Prolog, the unification algorithm is extended to handle read-only variables [Lev84,Sha83], and multiple OR-parallel environments are maintained for the applicable clauses to record bindings made locally to a goal variable. The commit operation hence involves three steps: (1) environment unification for goal variables; (2) if step 1 succeeds, eliminate the committed clause's siblings; (3) if the commitment in step 2 instantiates variables with pending goals, re-activate those goals.

[9] A (binding) environment is allocated to each clause invocation. It is a set of memory cells containing variable bindings.

2.2.2. PARLOG

PARLOG was proposed by Gregory [Gre85]. Its most notable feature is the *mode declaration* which not only enables synchronization, but also avoids the needs of full unification and multiple environments for goal variables (c.f. Concurrent Prolog). The language has two components: the single-solution subset and the all-solution subset. These two components are interfaced by PARLOG's set constructors. The all-solution subset is merely included to allow pure Horn clause programs to be written in PARLOG to solve all-solution problems; it is excluded from discussion in this thesis. The single-solution subset (hereafter known as PARLOG) is described below.

2.2.2.1. Syntax

A program in PARLOG is a finite set of procedures. Each procedure is a sequence of guarded clauses with the same predicate name and arity and separated by the symbol "." or ";". A guarded clause in PARLOG has the form:

$$H <\text{-}\ G_1 \text{ and } \ldots \text{ and } G_m : B_1 \text{ and } \ldots \text{ and } B_n \qquad m,n \geq 0.$$

where "and" is either the symbol "&" or ",". The H (clause head), G_i's (guard goal) and B_i's (body goal) are atoms. The G_i's collectively are called the guard. The B_i's collectively are called the clause body. The symbol ":" is the commit operator.

A mode declaration for a predicate symbol P has the form:

$$\text{mode } P(m_1,\ldots,m_n)$$

where each m_i is either the symbol "?" (input) or "^" (output). Each procedure has one and only one mode declaration.

2.2.2.2. Semantics

Declaratively, each of the commit operator ":" and the symbols "&" and "," is treated as an "AND" operator. Hence the program clause reads: H if G_1 and ... and G_m and B_1 and ..., and B_n. Each of the symbols ";" and "." is treated as an "OR" operator, so the clauses in a procedure are disjunctive.

Operationally, PARLOG provides a sequencing facility in goal evaluation and clause selection. The symbols "&" and "," denote sequential AND and parallel AND, respectively. They have no syntactic precedence, parentheses must be used to resolve ambiguity if they are mixed. For example, to evaluate the conjunction (A1,A2)&(A3,A4), A1 and A2 are first evaluated in parallel, if both succeed then A3 and A4 are evaluated in parallel.

The symbols ";" and "." denote sequential OR and parallel OR, respectively. "." has a higher precedence than ";". For example, if the following procedure is invoked:

```
Clause1;
Clause2.
Clause3;
Clause4.
```

Clause1 is attempted first. If it fails, Clause2 and Clause3 are attempted in parallel. If both fail, then Clause4 is attempted.

The mode declaration for the ~~predicate~~ symbol of a procedure constrains the head unification in two ways:

(1) The input constraint: the unification substitution for each *input argument* (trailed by "?") must be only an input substitution (see Section 1.2.3).

(2) The output constraint: each output argument (trailed by "^") in the goal must be an unbound variable at head unification time.

A unification violating an input constraint is suspended, that violating an output constraint causes a run-time error.

Unlike Concurrent Prolog, PARLOG's means of synchronization is attached to procedures, instead of data. When a procedure is invoked by a goal, its clauses can be classified into three categories:

(1) Candidate clause: if both the head unification's *input matching*[10] and guard evaluation succeed.

(2) Non-candidate clause: if either the input matching or the guard evaluation fails.

(3) Suspended clause: if either the input matching or the guard evaluation is suspended, but neither of them fails.

The goal fails if all its clauses are non-candidate, is suspended if there is no candidate clause but some suspended clauses, and succeeds if there is at least one candidate clause. In the last case, the goal commits to the first candidate clause (i.e. the committed clause) by replacing itself with the committed clause body, and aborts the committed clause's siblings. The head unification's output matching (i.e. unifications of the output arguments) and clause body are then evaluated. This may instantiate some variables and cause the re-activation of some suspended clauses.

Since the input/output arguments are fixed by the unique mode declaration of each procedure, there is no need to support full unification in PARLOG. In fact, each PARLOG program P will be translated into a *Kernel PARLOG* program KP before execution. In KP, there is no mode declaration. All the head arguments of a clause are replaced by distinct variables, and the input and output matchings are expressed in unification primitives and added to the original guard and clause body, respectively. Hence after the translation, the input matching in P is evaluated in parallel with the guard before commitment, and the output matching is evaluated in parallel with the clause body after commitment.

[10] A head unification's input matching is the unification of the input arguments. It succeeds if the unification of those arguments succeeds without violating the input constraint. It is suspended if the unification succeeds only when the input constraint is violated. It fails if the unification fails.

As in Concurrent Prolog, a clause must not instantiate a goal variable before commitment. Since in KP the output matching is evaluated after commitment, the only risk of binding a goal variable is in the guard evaluation. However, a compile-time algorithm can be devised to verify that a guard is *safe* (i.e. it does not bind any goal variable). Hence if a PARLOG program P passes the safe-guard check, its corresponding kernel PARLOG program KP will not bind any goal variable before commitment; this avoids the need to support multiple environments for a goal variable. The interested reader is referred to Gregory's thesis [Gre85] for more details in PARLOG's unification primitives, the features of Kernel PARLOG, and the algorithm for guard safety check.

2.2.3. GHC

GHC (i.e. Guarded Horn Clauses) was proposed by Ueda [Ued85]. Its most notable feature is the idea that only the guard is sufficient for describing concurrent computations.

2.2.3.1. Syntax

A GHC program is a finite set of guarded Horn clauses each has the form:

$$H <- G_1,...,G_m \mid B_1,...,B_n. \qquad m,n \geq 0.$$

The H (head), G_i's (guard goals), and B_i's (body goals) are atoms. The symbol "|" is the *trust operator*. The part of the clause before "|" (i.e. the H and G_i's collectively) is called the guard, and the part after "|" is called the clause body.

2.2.3.2. Semantics

GHC's semantics are defined by the *suspension rule* and the *trust rule*. The suspension rule is stated as follows:

(1) [Synchronization] Any piece of unification invoked directly or indirectly in the guard cannot bind a variable appearing in the invoking goal with (i) a non-variable term or (ii) another variable appearing in the invoking goal.

(2) [Sequencing] Any piece of unification invoked directly or indirectly in the clause body cannot bind a variable appearing in the guard with (i) a non-variable term or (ii) another variable appearing in the guard until the clause is trusted (see below).

A unification violating the suspension rule will be suspended. As a result, no goal variable can be bound before the clause is trusted. Hence GHC, like PARLOG, also does not need to maintain multiple environments for a goal variable.

The trust rule states that: a clause whose guard is successfully solved can try to be trusted. To be trusted, it must first confirm that no other clause in the same procedure has been trusted for the same goal. If confirmed, it is trusted indivisibly. The evaluation of a trusted clause body may bind some variables in the guard, and thus re-activate some suspended unification.

A set of goals is solved if it is reduced to an empty clause using only trusted clauses; hence after a clause is trusted all its siblings are useless and can be eliminated. The original semantics of GHC do not have the concept of failure. However, failure of unification can be readily introduced into the language: a set of goals fails if it contains or derives a *unification goal* (e.g. "=", see below) with non-unifiable arguments. Also, a goal proved to have no trustable clause can be detected as fail (under the *closed world assumption*, i.e. the assumption that the program contains all the relevant information about the problem).

As an example, consider the following program:

```
<- p(X), q(X).
```

```
[C1] p(ok) <- true | ... .
[C2] q(Z)  <- true | Z=ok.
```

The predicate "true" is used in GHC to denote empty set of guard or body goals. The predicate "=" is a predefined predicate which unifies its two arguments. C1 cannot instantiate the goal argument X since this will violate the suspension rule. After binding X to Z, C2 tries to be trusted and succeeds. It can bind X to "ok" after it is trusted.

Finally, it is worth mentioning that in GHC any kind of evaluation pattern is possible as long as the two rules are not violated. For example, the unification of the head arguments may be executed in any order or even parallel. The head unification and guard evaluation can be done in any order, e.g. in parallel. Even the clause body evaluation can start before that clause is trusted.

2.2.4. Comparisons

2.2.4.1. Comparison among the Three Languages

All of the three languages share the features of guarded clauses and committed don't-care non-determinism to implement stream parallelism, and as a result can only solve single-solution problems. However, there are several major differences among them. The first is the means of suspension in unification: in Concurrent Prolog, it is attached to data; in PARLOG, it is attached to procedures; in GHC, it is determined by the suspension rule.

Secondly, they use different unification algorithms: in Concurrent Prolog, unification is extended to handle read-only variables; in PARLOG, unification is dissolved into several primitives; in GHC the full unification is employed.

Lastly, they have different attitudes to guard safety. Concurrent Prolog does not detect unsafe guard, but provides multiple environments and performs environment unification at commitment. In PARLOG, unsafe guard is detected at compile time. A program with unsafe guards is rejected by the detection algorithm. In GHC, unsafe guard is detected at run time. Before the clause is trusted, any unification trying to bind a goal variable is suspended. However, as pointed out by Gregory [Gre85], all these methods are not satisfactory. Concurrent Prolog's multiple environment mechanism incurs considerable run-time overheads, and is claimed to have semantics problems: PARLOG's compile-time detection algorithm is insufficiently discriminating; it may sometimes reject a program that actually has safe guards. Also, even a perfect compile-time algorithm requires the check to be done in a global way; thus making separate compilation difficult. GHC's run-time check may be computationally expensive unless special-purpose architectural support is provided. Finding an efficient run-time safety check algorithm is therefore still an open question in the area of language design.

2.2.4.2. Comparison with Parallel Prolog

While parallel Prolog (i.e. Prolog using a parallel control strategy) is inferior in describing concurrent computations, it has two advantages over the three languages. Firstly, because of its don't-know non-determinism, it can solve all-solution problems. Also, it can find a solution provided one exists, while such guarantee is not provided in the three languages. As an example, consider the following Concurrent Prolog program:

```
<- p(X), q(X?).

[C1] p(1).
[C2] p(2).
[C3] q(1).
```

Whether the execution of this program succeeds or fails depends on which clause, C1 or C2, commits first. The implication of the commitment in the three languages is that: either the committed path leads to a solution, or, if it does not, there is no solution elsewhere in the AND-OR tree. It is the responsibility of the programmer to enforce this property. Despite these restrictions, the three languages are shown to be simple but powerful, and enjoy a large variety of application areas [ClG84,ShT83,Ued85].

2.3. Summary

We have described four forms of parallelism in logic programs and with each form, identified their natures and the problems of exploiting them. The desired features of a parallel logic programming language, namely concurrency, communication, synchronization and indeterminacy, are mentioned. Three of the recently proposed parallel logic programming languages, namely Concurrent Prolog, PARLOG and GHC, are introduced and compared with each other and with parallel Prolog. Appendix A1 contains some example programs written in the three languages.

As the sources of parallelism in logic programs have been identified and some parallel logic programming languages capable of describing concurrent computations have been devised, the next important questions should be: How to implement those forms of parallelism on parallel architectures? How to support the parallel execution of programs written in Prolog and those parallel logic programming languages? Chapter 3 is a survey of the research attempting to find the answers.

Chapter 3

Parallel Execution Models of Logic Programs

Parallel execution of logic programs is an active research topic in logic programming; it involves the development of parallel execution models and their realization on parallel architectures. Many parallel execution models, for both Prolog, its variants and the three parallel logic programming languages mentioned in Chapter 2, have recently been proposed and they have various scores on the justification factors stated in Section 1.1.1. In this chapter, we will study and justify the typical ones among them. These models can be broadly classified into five classes: the *communicating-processes approach*, the *work-pool approach*, the *reconciliation approach*, the *dataflow-based approach* and the *reduction-based approach*.

3.1. The Communicating-Processes Approach

A natural approach to represent the parallel computation in a logic program is to organize the concurrent components of computations as communicating processes, each notionally allocates a processor of its own. The processes share no common memory, but communicate with one another through message passing. As the AND/OR tree grows and shrinks during program execution, the process structure (i.e. the processes in the system and their interactions) changes accordingly. The execution models in this approach can be further divided into two subclasses: those employing *homogeneous processes*, and those employing *heterogeneous processes*.

3.1.1. Execution Models Employing Homogeneous Processes

A typical example in this subclass is *Bowen's model* for interpreting Prolog programs [Bow82]. Figure 3.1 gives the process structure of the model. The USER AGENT interacts with human users and generates problems for the rest of the system. The database machine DBM manages the clause database. The SCHEDULER allocates EXPLORERs to problems and coordinates the EXPLORERs and the USER AGENT. The EXPLORERs are the homogeneous processes searching the AND/OR tree, each of them executes a version of van Emden's algorithm [van81]. Basically, an EXPLORER tries to verify that its current tree node is an empty clause. If the verification succeeds, it informs its parent of the success and awaits further instructions. Otherwise, it expands the current node by generating the node's sons, and requests the SCHEDULER to allocate an EXPLORER to each son. It may recurse to explore a subtree rooted by a son, if no EXPLORER is available. When no son of the current node can be generated, the EXPLORER fails and informs its parent of the failure.

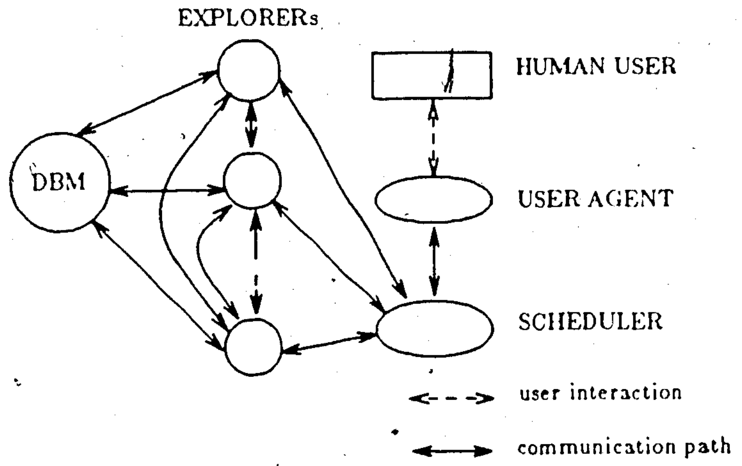


Figure 3.1 Process Structure of Bowen's Model

Three kinds of messages are passed among the EXPLORERs: the *stop* messages from parents to sons, and the *failure* and *success* messages from sons to parents. The model can exploit both AND-parallelism and OR-parallelism; for the former it depends on the explicit notations of the language (like those in IC-Prolog) to tell the interactions of the different AND-branches. Search parallelism is also possible if the DBM comprises several components.

Another example is the *PRISM model* [KKM83] which has been developed at the University of Maryland and has been implemented on ZMOB, a parallel machine consisting of 256 Z80A microprocessors. The PRISM model aims at the development of distributed AI problem solvers, and is based on Prolog enriched with various control annotations. It has three components: the *problem solver* (PS), the *extensional database* (EDB) that consists of all the atomic ground assertions, and the *intentional database* (IDB) that consists of all the rules and non-EDB assertions (i.e. those having complex terms and variables). The PS component administers the search space defined by the AND/OR tree; while the EDB/IDB components store the clauses and perform unifications. The separation of the clause database into EDB and IDB enables the application of different unification algorithms for efficiency purposes.

At system initialization time, the pool of processors in ZMOB is divided into three groups: the PS machines (PSMs), the EDB and IDB machines (collectively known as the database machines or DBMs). Each of the homogeneous processes searching the AND/OR tree is executed on a PSM. In each cycle, a PSM process chooses an expandable goal clause in the tree, selects one or more atoms from it, and sends them to the DBMs for unifications. The DBMs return all the necessary information needed to generate the successors of the goal clause. While waiting for the results from the DBMs, a PSM process can start expanding other goal clauses. During the expansion of the AND/OR tree, a new PSM process can be started every time an OR-branch or an independent AND-branch (i.e. that shares no variable with others) takes place. This results in a PSM process tree whose nodes are autonomous except for the parent-child relations; hence interprocessor communication is necessary only when answers are available. The model can exploit OR-parallelism, independent AND-parallelism, and search parallelism.

3.1.2. Execution Models Employing Heterogeneous Processes

Unlike the models in Section 3.1.1 which merge the controls of AND/OR parallelism into a single process, the execution models in this subclass employ different kinds of processes to search the AND/OR tree, so that the algorithm handling each form of parallelism can more easily be identified.

A typical example is Conery's *AND/OR Process Model* for interpreting Prolog programs [Con83]. It has two kinds of processes: the *AND-processes* and the *OR-processes*. The former handle AND-parallelism, while the latter handle OR-parallelism. Figure 3.2 depicts the alternating levels of AND-processes and OR-processes in a sample AND/OR process tree.

An OR-process controls the evaluation [illegible]l the applicable clauses for an atom, and relays the results to its parent AND-process. An AND-process controls the evaluation of a clause body. It has two modes of execution: the *forward execution* mode spawns descendant OR-processes to solve the atoms in the clause body and schedules their execution, while the *backward execution* mode performs backtracking in a parallel context, but needs quite expensive bookkeepings. Besides the success and fail messages, there are other kinds of message passing among the processes: a *start* message is used to invoke a descendent, a *redo* message is used to ask for a new answer from a descendent, and the *reset* and *cancel* messages are used by an AND-process in backtracking to reset and abort the executions of the affected descendant OR-processes, respectively. The model can exploit both OR-parallelism and AND-parallelism.

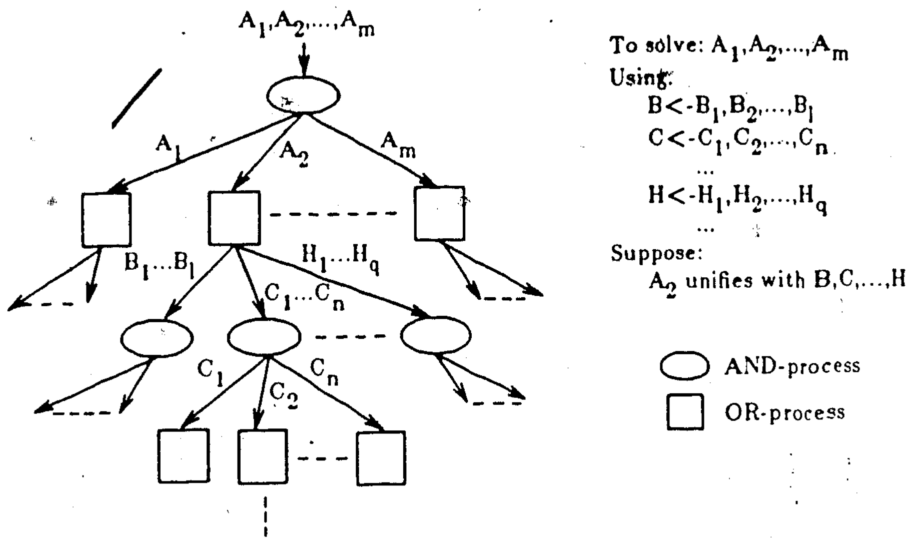

$A_1,A_2,\ldots,A_m$
A_1
A_2
A_m
$B_1 \ldots B_l$
$H_1 \ldots H_q$
$C_1 \ldots C_n$
C_1
C_2
C_n
To solve: $A_1,A_2,\ldots,A_m$
Using:
$B \text{<-} B_1,B_2,\ldots,B_l$
$C \text{<-} C_1,C_2,\ldots,C_n$
...
$H \text{<-} H_1,H_2,\ldots,H_q$
...
Suppose:
A_2 unifies with B,C,...,H
AND-process
OR-process

Figure 3.2 A Sample AND/OR Process Tree

Another example is the *distributed Concurrent Prolog interpreter* sketched in Shapiro's paper ([Sha83], pages 12-14). The *Port Prolog* execution model [Lee84] realizes those ideas on a quasi-parallel environment. The model uses three kinds of processes: the *conjunction-processes*, the *goal-processes* and the *clause-processes*. Figure 3.3 depicts a sample process structure in Port Prolog.

Given a conjunction $A_1,A_2,\ldots,A_m$, the conjunction-process creates a goal process for every atom A_i, $1 \le i \le m$. A goal process finds the solution of its atom A_i by creating a clause-process for each clause whose head has the same predicate name and arity as A_i. Given a clause C and an atom A_i, a clause process tries the head unification between C and A_i, and if successful, spawns a conjunction-process to solve C's guard. The first clause process to inform its parent goal-process of successful guard evaluation becomes the *candidate clause-process*, all the other clause-processes are then destroyed. The parent goal-process sets up the connection between its father conjunction-process and the candidate clause-process and then destroys itself.

Meanwhile, the candidate clause-process spawns a conjunction-process to solve the clause body and sends the result to its grandparent conjunction-process. A conjunction-process succeeds if all its candidate clause-processes succeed, and fails if any of them fails; the result is, in turn, communicated to its parent clause-process (if any). The messages transmitted among the processes include both control (e.g. success, fail, or *commit*) and variable binding (e.g. *new binding, need binding*) information. The model, if implemented on a parallel architecture, can exploit both AND-parallelism, committed OR-parallelism and stream parallelism.

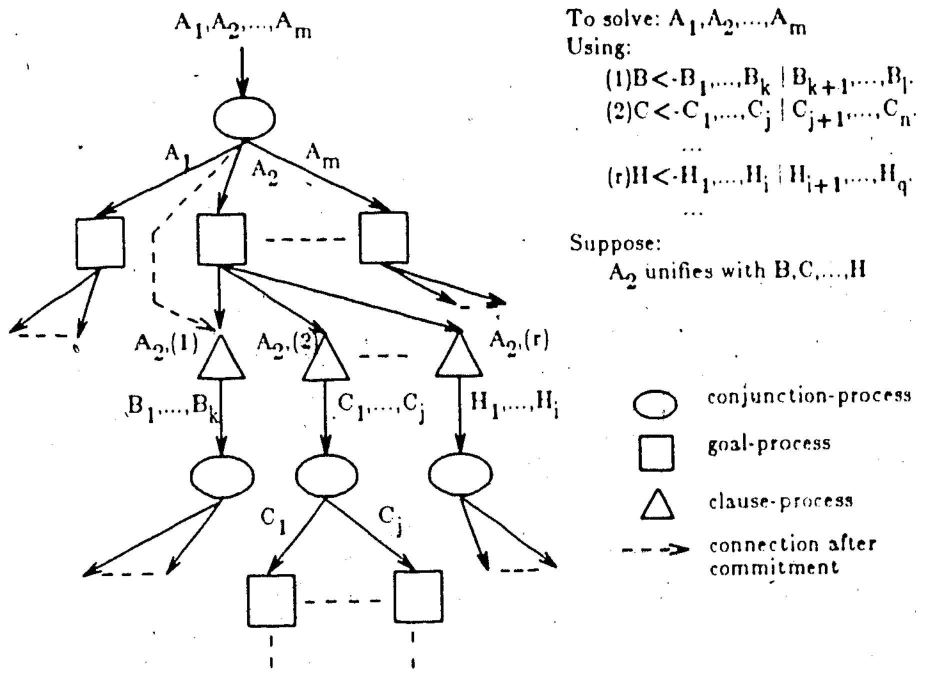

$A_1,A_2,\ldots,A_m$
A_1
A_2
A_m
$A_2,(1)$
$A_2,(2)$
$A_2,(r)$
$B_1,\ldots,B_k$
$C_1,\ldots,C_j$
$H_1,\ldots,H_i$
C_1
C_j
To solve: $A_1,A_2,\ldots,A_m$
Using:
$(1) B <- B_1,\ldots,B_k \mid B_{k+1},\ldots,B_l.$
$(2) C <- C_1,\ldots,C_j \mid C_{j+1},\ldots,C_n.$
...
$(r) H <- H_1,\ldots,H_i \mid H_{i+1},\ldots,H_q.$
...
Suppose:
A_2 unifies with B,C,...,H
conjunction-process
goal-process
clause-process
connection after commitment

Figure 3.3 A Sample Process Structure in Port Prolog

The *AND-OR Tree Model* for PARLOG ([Gre85], Chapter 6) can also be classified into this subclass. In this model, the process structure is not a hierarchy of alternating levels of AND/OR processes, but is controlled by the sequential and parallel AND/OR operators (i.e. "&", ",", ";" and "."). The model can exploit both AND-parallelism, committed OR-parallelism and stream parallelism.

3.1.3. Evaluation of the Approach

While the representation of an AND/OR tree as communicating processes is natural and has a high parallelism potential, execution models in this approach usually encounter the following difficulties. Firstly, many practical problems must be satisfactorily solved before the models can be effectively realized on their target architectures; usually some conventional distributed systems. These problems include the optimal allocation of processors to processes, the handling of unexpected communication patterns, etc. Also, the overheads of the models, in terms of the management of complex data structures, the message costs and the process synchronization delay (for some models), are quite high.

3.2. The Work-Pool Approach

The execution models in this approach partition the AND/OR tree into computation units and store them in a work pool which is commonly accessed by a set of processes. The processes advance the concurrent computations independently (vs. message passing); each process repeatedly selects a unit from the pool, performs some computations, and adds the generated results (if any) back into the pool. The concurrent computations complete successfully only if the work pool is empty.

3.2.1. The OR-Parallel Token Machine Model

A typical example is the *OR-Parallel Token Machine model* which exploits OR-parallelism in Prolog program [CiH84]. The model, as depicted in Figure 3.4, consists of a token pool, a set of processors and a storage.

The storage is divided into a static memory for user program and a dynamic memory for binding environments and other management information. The processors execute computation units prescribed by the tokens and generate new tokens. The derivational relations among tokens are implicit in the token naming scheme: A token contains references to an atom, a context, a binding environment and a *continuation frame*. The atom reference refers to an instruction stored in the static memory. The context and binding environment references collectively identify the set of variables used by the prescribed computation unit. The continuation-frame reference refers to a continuation frame representing the untried siblings of the atom. The sequential nature implied by the continuation frame disallows the exploitation of AND-parallelism and stream parallelism. The target architecture of the model is a system containing a set of processing elements that communicate via an order preserving interconnection network. A sketch of implementing the model on a ring network is provided by Ciepielewski and Haridi [CiH84]. While featuring demand load balancing and abortion propagation, the sketch seems to be realizable with reasonable efforts.

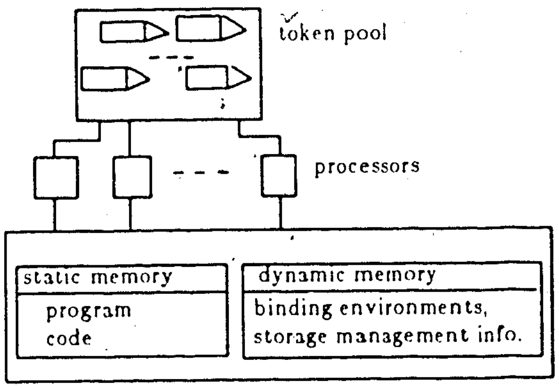


Figure 3.4 The OR-Parallel Token Machine Model

3.2.2. The Goal Rewriting Model

Another example is the *goal rewriting model* on which the PIE (Parallel Inference Engine), a highly parallel architecture containing 100s ~ 1000s inference units (IUs), is based [GTM84]. The computation units in the goal rewriting model are *goalframes*. A goalframe is a partial result in the derivation of an empty clause; it is the combination of an intermediate goal clause and the substitutions produced by the previous unifications. For example, Figure 3.5(a) depicts a goalframe resulting from two contiguous reductions of the initial goal clause G using the clauses C3 and C1. The size of a goalframe increases as the computation proceeds, however, it can be reduced by the removal of information unnecessary for later unifications. For example, the goalframe in Figure 3.5(a) can be reduced to that in Figure 3.5(b) by removing successful unifications (represented by full circles) from it.

```
[G]  <- subl([a|X],[Y]).
[C1] app([],X,X).
[C2] app([U|X],y,[U|Z]) <- app(X,Y,Z).
[C3] subl(X,Y) <- app(U,X,V),app(V,W,Y).
```

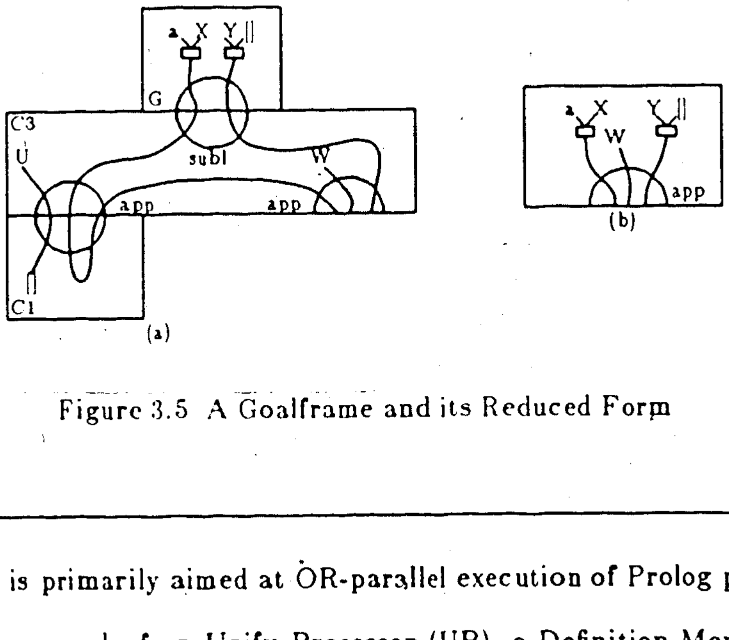


Figure 3.5 A Goalframe and its Reduced Form

The model is primarily aimed at OR-parallel execution of Prolog program. An IU in the PIE is composed of an Unify Processor (UP), a Definition Memory (DM) that stores all the clauses, a Memory Module (MM) that stores the goalframes, and an

Activity Controller (AC) that creates and maintains part of the *inference tree* (see below). The basic operation of an UP is the rewriting of goalframes: each UP repeatedly inputs a goalframe from the MM and selects an atom from it, then unifies the atom with the heads of the applicable clauses read from the DM, and finally generates the new goalframes (if any), reduces them, and adds them into the MM. The goalframes are independent except for derivational relations which are recorded in the inference tree (vs. the token naming scheme used in the OR-parallel token machine). An internal node of the inference tree records a relation among goalframes; while a leaf node either refers to a goalframe (goal-node), indicates failure (fail-node), or has special function (e.g. NOT). Besides OR relationship, the inference tree is claimed to be capable of supporting independent-AND relationship, negation as failure, and the guard concept (hence can be adapted for implementing Concurrent Prolog, etc). The major practical problems are the efficient activity control involving the distributed inference tree in the ACs, and load balancing involving the goal pool distributed among the MMs. Figure 3.6 shows an abstract view of the model. Results of simulations and pilot implementations of the model are provided by Moto-oka, et al. [MTA84].

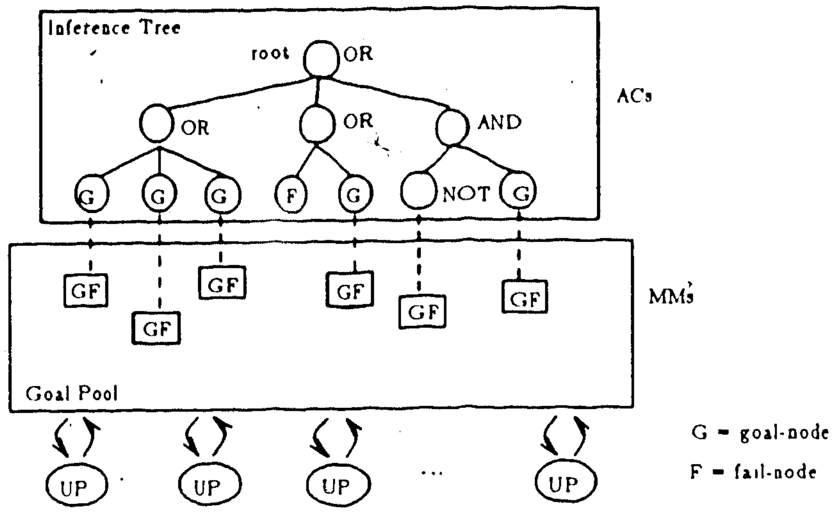


Figure 3.6 The Goal Rewriting Model on PIE

3.2.3. Evaluation of the Approach

The benefit of this approach is that OR-parallelism is easily controlled through the management of the work pool; where different scheduling strategies, such as depth-first, breadth-first, or some mixed schemes, can be employed to direct the growth of the AND/OR tree. However, unrestricted AND-parallelism, which requires communication and coördination among the processes, is quite hard to exploit. A simpler task may be to support the parallel logic programming languages mentioned in Chapter 2 in this approach, where the AND-parallelism and stream parallelism are severely constrainted on order and pace by the synchronization mechanisms (e.g. the read-only variable).

3.3. The Reconciliation Approach

The example execution model in this approach is the *AND-OR Proof Procedure* proposed by Pollard for parallel execution of Horn clause programs ([Pol81], Chapter 6). The model performs a completely unrestrained parallel traversal of the AND/OR tree; searching for compatible sets of branches. In parallel with the AND/OR tree traversal performed by a set of processes, there is another set of administrative processes performing the tasks of *reconciliation* and pruning. Basically, the unification substitutions generated in different branches of the AND/OR tree are reconciled to produce *filters* that identify sets of mutually incompatible branches. Based on the AND/OR tree structure, the filters are *promoted* and *reduced* to derive information leading to the pruning of unfruitful branches. Figure 3.7 is a simple example to clarify the ideas.

Referring to the scopes of the various bindings for the variable Z, {h} and {i} are *disjoint* (i.e. most recent common ancestor is an OR-node) whereas {h} and {j}, {i} and {j} are *conjoint* (i.e. most recent common ancestor is an AND-node). Bindings in conjoint scopes must be compatible, hence two reconciliations are set to progress for the scopes {h, j} and {i, j}, trying to reconcile the components {Z=1, Z=3} and {Z=2, Z=3}, respectively. Obviously both the reconciliations will fail, so two filters {h, j} and {i, j} are developed. Since h and i are the only ways to solve the OR-node d, the filters can be promoted to {d, j}. However, since any solution including the AND-node b must also include its descendants (where one of them is d), the filter can be further promoted to {b, j}. As b is an ancestor of j, it can be removed from the filter, reducing the filter to a singleton set {j} which indicates that the branch leading to node j can be safely pruned.

```
goal clause and program
<- goal(Y,Z).
goal(Y,Z) <- p(U),q(V),R(U,V).
p(1).
p(2).
q(1).
q(2).
r(W,3) <- ... .
r(X,X).
...
```

variable	binding {scope}
U	Y {b}
V	Z {b}
W	Y {j}
X	Y {k}
Y	1 {f}, 2 {g}, Z {k}
Z	1 {h}, 2 {i}, 3 {j}

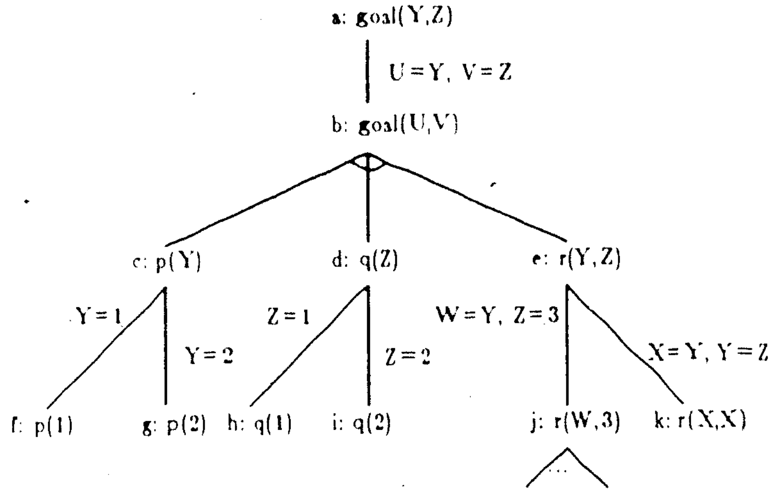


Figure 3.7 The Generated Bindings in an AND/OR Tree

3.3.1. Evaluation of the Approach

While the AND-OR proof procedure can achieve the maximum possible (AND-, OR-, and stream) parallelism from any pure Horn clause program, it has the risk of an explosion in parallel activities, and its computational overhead (e.g. a set of administrative processes is associated to every generated variable) is so high that no efficient realization on a present-day architecture is feasible.

3.4. The Dataflow-Based Approach

The dataflow model is perhaps one of the most promising models for parallel computation. The basic idea is the representation of computation as a ***dataflow graph***: a directed graph whose nodes represent the operators while the data are represented as tokens flowing on the directed arcs connecting the nodes. The lack of global memory and the data-driven characteristic (i.e. an operator is enabled once all its input tokens have arrived) of the model allows a high potential for parallelism [TBH82]. A simple computation step is illustrated in Figure 3.8 that shows two dataflow graph segments for the execution of (3+5)*(7-12). The dataflow model was originally designed for *functional programming*, but also has similarity to the goal driven nature of logic programming; this leads to the proposals of various dataflow-based parallel execution models for logic programs.

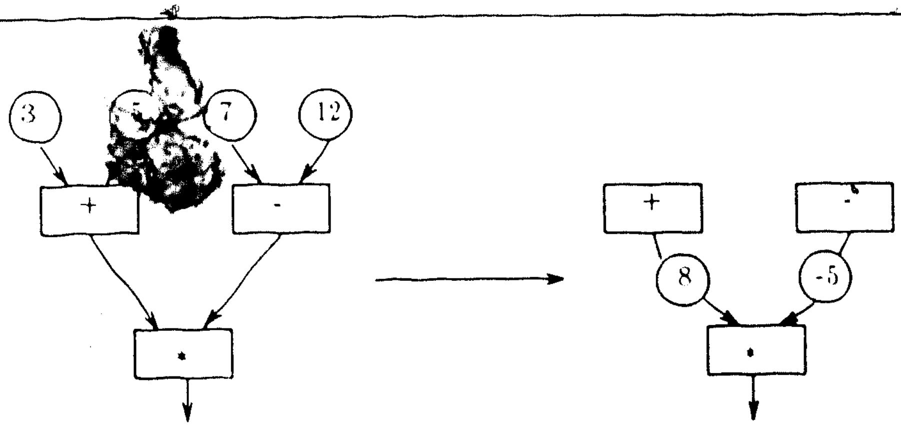


Figure 3.8 A Simple Computation Step in the Dataflow Model

A typical example is Halim's data-driven model which exploits OR-parallelism in pure Horn clause programs [Hal86]. Figure 3.9 gives an abstract view of the model where clauses are compiled into dataflow graphs and the data flowing on the directed arcs are binding environments (BEs). A *Solve node* is associated with the pure code of a goal, and is viewed as a function which maps an input BE into a set (organized as a stream) of BEs, one for each alternative solution. Figure 3.9(a) is an example of a Solve node associated with the pure code p(X,Y). When invoked by an input BE E1, the node will generate the set of BEs E2 and E3 as its output. Using the solve node as a building block, the model can represent goal clauses, rules and assertions as shown in Figure 3.9(b). The representation does not support dynamic goal selection, thus disallowing the exploitation of AND-parallelism and stream parallelism. The *Unify node* is a primitive operator; it has the clause head as its constant argument and is activated by the arrival of a goal token (constructed by an *Activate node*, see below). Its function is to construct a BE containing the bindings unifying the clause head with the incoming goal. The Solve node is not a primitive operator; its structure is shown in Figure 3.9(c). The pure code of a goal G, associated with the solve node, is a constant argument of the Activate node. The function of this primitive operator, when enabled by the *activating BE* E, is to construct the goal G.E (i.e. G with E applied to it) and send it to each of the applicable clauses. A clause in 3.9(c) is represented by a procedure call Proc-j that corresponds to either (ii) or (iii) in 3.9(b). All the clauses are invoked by the goal token G.E concurrently, thus exploiting OR-parallelism. Each invoked clause Proc-j evaluates a stream of *returning BEs* $\{E_i\}_j$ which are fed into the *Extract-output* node whose function is to extract output bindings from the returning BEs and assimilate them into the activating BE E to produce the *solution BEs*.

(a)

program	binding environment
p(tom,john).	E1 = {X=tom, Y=UNBOUND}
p(tom,sam).	E2 = {X=tom, Y=john}
	E3 = {X=tom, Y=sam}

(b)

(i) <- P,Q,...,R.	[goal clause]
(ii) H <- $A_1,...,A_n$.	[rule]
(iii) H.	[assertion]

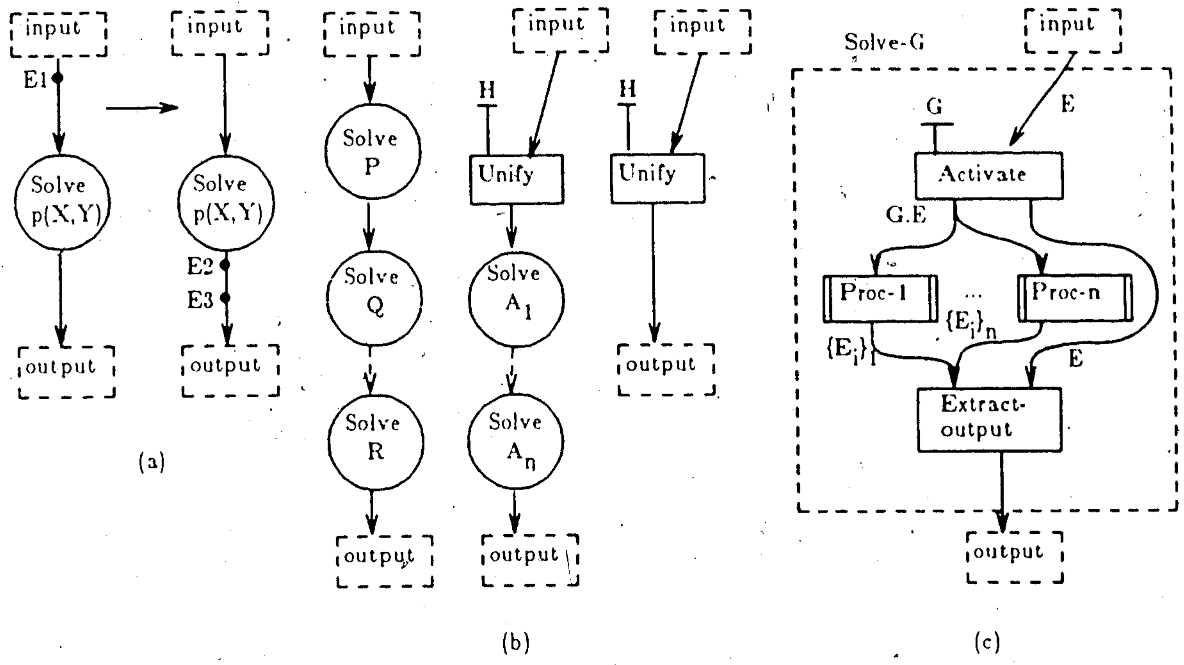


Figure 3.9 Halim's Data-Driven Model

Another example is the OR-parallel model proposed by Umeyama and Tamura [UmT83]. It is very closed to Halim's model but uses a different set of primitive operators. A comparison between the two models is provided by Halim [Hal86]. Their major similarity is the treating of alternative solutions as stream of tokens, while the major difference is the abandonment of *dynamic tagging* in Halim's model.

The dataflow architecture research group in ICOT[11] has recently proposed the

[11] Institute for New Generation Computer Technology, the organization coordinating the Japanese Fifth Generation Project.

mechanisms for executing parallel Prolog and Concurrent Prolog programs on a dataflow architecture [ISK85]. The description is made in a lower level with emphasis on architectural aspects, token format, variable representation, etc. In this proposal, the unify operator is further expanded into a dataflow graph so that parallelism in unification is also exploitable. An operand in the model may be a goal, or a finer object such as constant, variable, etc. Structure data are stored in the *structure memories* to minimize copying overhead and redundancy. A clause is, as usual, represented as a dataflow graph. OR-parallelism is exploited by concurrent invocations of applicable clauses. The *stream merge primitives* are used to merge the alternative solutions into a stream in parallel Prolog, while the *semaphore primitives* and *exporting mechanism* are used to achieve the effect of commitment in Concurrent Prolog. AN~~D-parallelism c~~an also be exploited, but syntactical notations are needed to aid the compilation of a clause into a dataflow graph.

Other examples include Bic's model where all predicates are restricted to a binary form [Bic84], and Wise's model which is based on the concept of *dframe*, a centralized data structure including all the necessary information to solve a goal [Wis82]. A sketch of a dataflow implementation for GHC is provided by Ito et al. [KKI85].

3.4.1. Evaluation of the Approach

While a dataflow-based execution model may be the most suitable scheme for exploiting massive parallelism in logic programs, a number of difficulties reveal themselves when an actual implementation is attempted. Firstly, if the model is too fine-grained, what is gained through parallelism exploitation may be lost through communication overheads unless the underlying architecture provides sufficient supports. Also, if the target applications need to manipulate a large amount of structure data (e.g. language processing), the provision of structure memories shared access by all the processing elements is necessary; this will introduce latency in accessing structure data and incur extra management costs.

3.5. The Reduction-Based Approach

The execution models in this approach are usually based on the *graph reduction* computation model where a computation is viewed as a nested expression whose execution unfolds to a graph of subexpressions. As various portions of the graph are evaluated, they are replaced by their intermediate results until the computation fails or the final result is obtained. Various steps in unfolding and reducing the graph can be carried out in parallel [TBH82].

An example is the *AND-tree model* of PARLOG where the execution of a PARLOG program, after compiled into this model, conforms to an AND-tree process-structure ([Gre85], Chapter 7). The AND-tree process structure can be represented on ALICE [DaR81], a graph reduction architecture, by a tree structure of *rewritable packets*. The format of such a rewritable packet is shown in Figure 3.10. The ID field uniquely identifies the packet, the Function field contains the function of the corresponding process (e.g. EVAL), the Arg-List field contains the identifiers of the argument packets, and the secondary fields are used for control purposes (e.g. Ref-Count counts the number of sons, Signal-List can be used for goal suspension/activation).

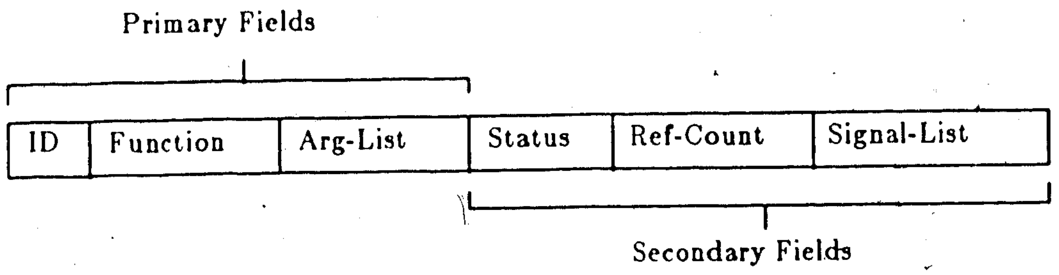


Figure 3.10 The Structure of a Rewritable Packet in ALICE

In the process tree, an internal node is represented by a rewritable packet whose argument fields reference other rewritable packets; while a leaf node (i.e. an untried goal) is represented by a rewritable packet whose argument fields reference the goal arguments represented by *constructor packets* (i.e. those that are not rewritable). The evaluation of a leaf node may rewrite the packet to a structure of rewritable packets representing the committed clause body, or the constructor packet SUCCEEDED or FAILED. The process tree is expanded in the former case, and reduced in the latter case. Computation stops when the root of the tree is rewritten to a constructor packet. AND-parallelism is exploited in this model.

The reduction architecture research group in ICOT has also proposed a highly parallel reduction-based architecture for logic programs, where OR-parallelism in Prolog and AND-parallelism in Concurrent Prolog are exploited [OAS85]. Various techniques, such as the *only reducible goal copy method,* structure memory, etc. are employed to reduce storage and copying overheads.

3.5.1. Evaluation of the Approach

A reduction-based execution model may also be a good scheme for exploiting massive parallelism. However, like a dataflow-based model, various practical problems reveal themselves when an implementation is attempted. The provision and management of structure memory are necessary, if target applications need to manipulate a large number of structure data.

3.6. Summary

In this chapter, We have classified the parallel execution models for Prolog, its variants and the three parallel logic programming languages mentioned in Chapter 2, into five classes; and within each class, studied the typical examples. These examples are briefly evaluated according to the justification factors stated in Section 1.1.1.

While the design of an elegant execution model is an important step, what is equally important is that the model should not require a highly complicated architecture (c.f. the reconciliation approach) and should be realizable on a parallel architecture with reasonable efforts. Most of the execution models studied in this chapter are parts of some ambitious long-range projects which have many objectives, such as the exploitation of massive parallelism, the search for an ideal architecture for logic programs, etc. Various practical problems, most of them concerning the realizability of the models and the effective control of overheads, must be solved before these models can actually be realized. While a long-range project has this luxury, it is reasonable for a short-term project to design an execution model that aims at exploiting moderate parallelism, and put more emphasis on its realizability and low overheads. This considerations lead us to propose the execution model mentioned in Chapter 4.

Chapter 4

Proposal of an Execution Model

In this chapter, we propose an execution model for the parallel logic programming languages Concurrent Prolog, PARLOG and GHC on a shared memory multiprocessor architecture. The model is heavily inspired by Levy's work [Lev86] (see Appendix A2), but incorporates the following extensions:

(1) Additional pointer fields are used to connect the run-time structures together so that they reveal the AND-OR tree structure which allows information of failure and abortion to be propagated promptly. Failure is propagated up the AND-OR tree and across the conjunction of goals. Abortion due to commitment is propagated down the AND-OR tree and across the disjunction of clauses (see **DO_FAILURE** and **DO_KILL** in Section 4.4.2.6).

(2) The detection of deadlocks caused by a user program (see Section 4.4.2.1).

(3) The initialization of the run-time structures for the top level goal (see Section 4.4.1).

(4) The detection of program termination (see Section 4.4.2.5).

A few modifications are also made, mainly to improve the original model (e.g. to avoid re-processing of aborted/failed goals. See **DO_WAKEUP** in Section 4.4.2.6).

4.1. Architectural Assumptions

In the following, we list the assumptions of the architecture supporting the proposed model and their implications:

(1) The architecture is a shared memory multiprocessor with a moderate number (say, from 2 to 30) of processors each of which has some local memory. The shared memory stores the run-time structures, the binding environments, the user program and the system program implementing the algorithm in Section 4.4.2; it is accessed by all the processors. Each local memory stores the local variables used in the system program; it is only accessible by its owner processor.

(2) Write access to shared memory is controlled by a *locking* mechanism. To obtain a lock is not expensive, as memory contention is assumed to be not severe since the number of processors is moderate. As a result, all the processors will use the *busy-wait* strategy to obtain a lock. Also, a processor will frequently release and re-obtain a lock to minimize the period of a *critical region*.

(3) Locking is not necessary for read access to shared memory; a consistent value will be read even if the same location is being written by another processor.

The reasons for aiming at shared memory multiprocessor architectures are:

(1) The model naturally calls for such architecture.

(2) Some shared memory multiprocessors are readily available.

(3) It is probable that in the future, a node of a loosely-coupled architecture is itself a shared memory multiprocessor, so the model developed here is still applicable to the nodes of such architecture.

4.2. Overview of the Model

The proposed model is based on the work-pool approach, where the AND-OR tree of a logic program is partitioned into computation units. The model is coarse-grained, when compared with most of the execution models discussed in Chapter 3. Instead of treating an AND-OR tree node (or an even finer object) as a computation unit, it views the computation necessary to satisfy a goal with its applicable clauses as a unit. This approach allows parallelism to be easier to control, which is significant for the target architecture. Figure 4.1 shows an sample AND-OR tree, each dashed box around a goal (OR-node) and its clauses (AND-nodes) represents a unit computation of the problem.

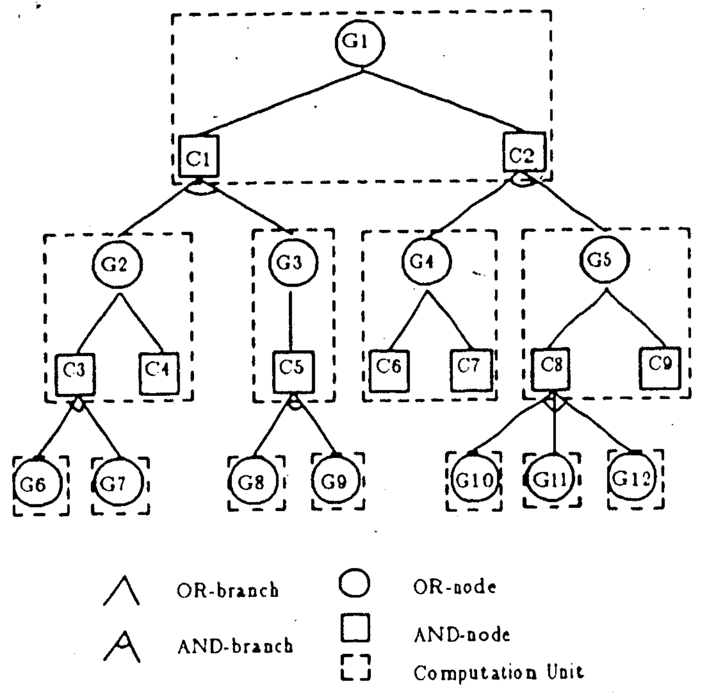


Figure 4.1 An AND-OR Tree Partitioned into Computation Units

During a user program execution, the computations represented by the AND-OR tree are advanced by a set of processes (each running on a processor). The goals which currently remain to be solved are divided into two groups: the ready goals and the suspended goals. The set of processes, each executing the same algorithm, share access to a ready goal pool. Each process repeatedly selects a goal from the pool, applies all the applicable clauses, and adds the generated subgoals (if any) into the pool. A goal suspended on a variable is inserted into the *suspension list* of that variable, and is re-activated until that variable is instantiated by another goal.

In this model, a goal reduction step is divided into different stages as depicted in Figure 2.2.[12] The computations associated with a computation unit are advanced by a

[12] This staged goal reduction concept does not properly fit the operational semantics of PARLOG and GHC. Considerations in supporting these languages are discussed in Chapter 5.

finite state machine whose state is implicitly recorded in the data structures discussed in Section 4.3. The computations in a specific goal reduction are depicted in Figure 4.2. A process solves a computation unit by attempting the applicable clauses in sequence. When the computation involving a clause cannot be further advanced, the next clause is tried. If a head unification succeeds (as in the first, third and fourth clauses), the goals in the guard are spawned and added into the ready goal pool. A head unification may also fail (as in the second clause) or be suspended on a variable (as in the fifth clause). For the latter case, it is re-tried when the variable is instantiated.

The goals in a spawned guard are computed concurrently by the processes. A spawned guard may fail (as in the first clause), result in commitment if it is the first one to finish (as in the fourth clause), or be aborted because of another clause's commitment (as in the third clause). Only one clause can reach the commitment stage. After commitment, the goals in the clause body are spawned and added into the ready goal pool.

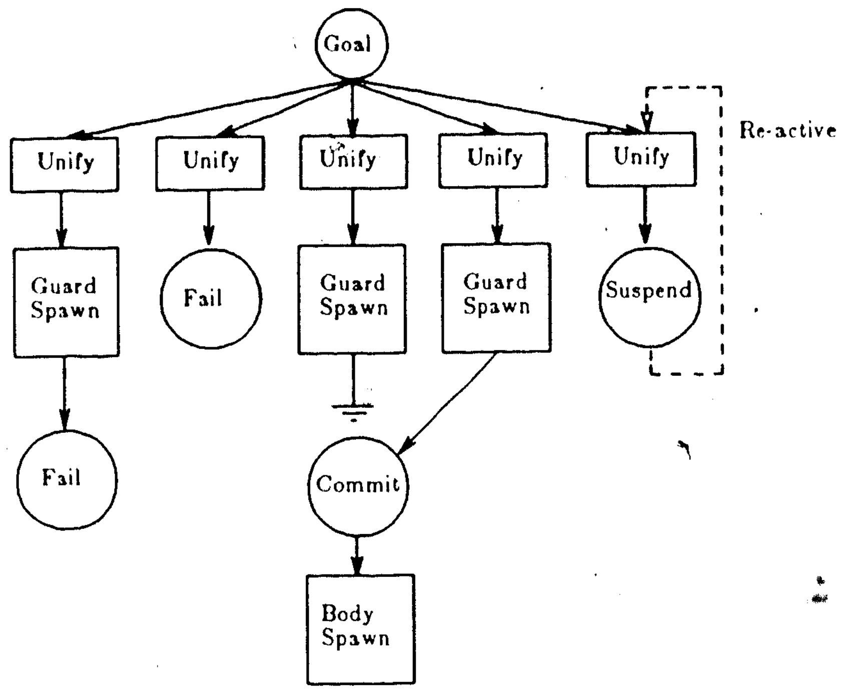


Figure 4.2 Computations in a Specific Goal Reduction

To achieve a unanimous point of view, each goal, except the top level one, is contained in some guard. When a goal commits to a clause, it is replaced by the clause body in the containing guard. A key requirement of the model is the ability to re-activate a computation unit when the spawned guard of one of its clauses is completely solved, so that the commitment step can be performed. This is accomplished by the *recursive commitment* mechanism, which occurs when the last goal G in a guard commits to a clause without body (and is thus solved), causing G's father goal to commit to G's father clause (see Section 4.4.2.5). The data structure *guard tuple* is provided to support this mechanism (see Section 4.3.3).

4.3. Data Structures

A computation unit is represented by a *goal record* and several guard tuples. A goal record contains all the relevant information for the computations involving a goal. A guard tuple is used to monitor the guard evaluation of a clause.

4.3.1. Goal Record

The fields of a goal record, depicted in Figure 4.3, have the following meaning:

(1) The *Procedure* field is a pointer to a list of all the applicable clauses for this goal.

(2) The *Arguments* field is a pointer to the list of arguments of this goal.

(3) The *Guard_Tuple* field is a pointer to the *guard tuple* (see Section 4.3.3) representing the guard containing this goal.

(4) The *Status_Vector* field is a vector of status, one for each of the clauses in the *Procedure* field. The different status of a clause is provided in Section 4.3.2.

(5) The *Env_Vector* field is a vector of environments, one for each of the clauses in the *Procedure* field.

(6) The *GT_Vector* field is a vector of pointers to guard tuples, one for each of the clauses in the *Procedure* field. The pointer for a clause is [illegible] before the guard is spawned (see Section 4.3.2).

(7) The *Flag* field shows the status of the goal, it is either READY (the goal is in the ready goal pool), DEQUEUED (the goal is dequeued from the ready goal pool), or DEAD (the goal has failed, or been reduced or aborted).

(8) The *R_Link* field is a pointer to the next goal record in the ready goal pool.

(9) The *G_Link* field is a pointer to the next goal record in the *Goal_List* of *Guard_Tuple*.

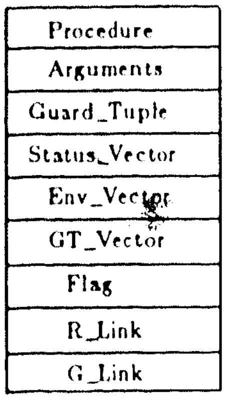


Figure 4.3 The Structure of a Goal Record

4.3.2. Clause Status

Each entry in the Status_Vector field of a goal record shows the status of the corresponding clause in the Procedure field. The following lists all the possible status of a clause:

(1) *UNIFY*: the clause is ready to start head unification.

(2) *UNIFYING*: the clause is now performing head unification.

(3) *SPAWN*: head unification succeeds, the clause is ready to spawn its guard.

(4) *SPAWNING*: the clause is now spawning its guard.

(5) *SPAWNED*: the clause has spawned its guard.

(6) *COMMIT*: the goal is ready to commit to this clause.

(7) *COMMITTED*: the goal has committed to this clause.

(8) *FAIL*: either head unification or guard evaluation of this clause fails.

(9) *KILLED*: the computation of this clause is aborted because of another clause's commitment.

4.3.3. Guard Tuple

The fields of a guard tuple, depicted in Figure 4.4, have the following meaning:

(1) The *Father* field is a pointer to the goal record of the goal invoking the clause containing this guard.

(2) The *Clause* field is a pointer to the clause containing this guard.

(3) The *Guard_Count* field records the number of goals in the guard still not yet solved. *Father* attempts to commit to *Clause* when it reaches zero.

(4) The *Goal_List* field contains a list of goal records. Initially the list contains the goals in the guard. When a goal reduces to its committed clause body, the goals in the body are added into the list. For the sake of efficiency, a reduced goal is not removed from the list, but marked as DEAD to be distinguished from other active goals.

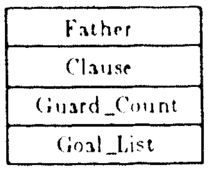


Figure 4.4 The Structure of a Guard Tuple

4.3.4. Suspension List

A goal may be suspended on some variable(s) during unification. The model records this fact by associating a suspension list with each variable. A suspension list is a list of suspension records, one for each of the goals suspended on that variable. A suspension record is depicted in Figure 4.5. It has two fields: the *Link* field points to the next suspension record in the suspension list, the *Goal_Record* field points to the goal record of the suspended goal. A suspension list is depicted in Figure 4.6.

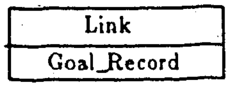


Figure 4.5 The Structure of a Suspension Record

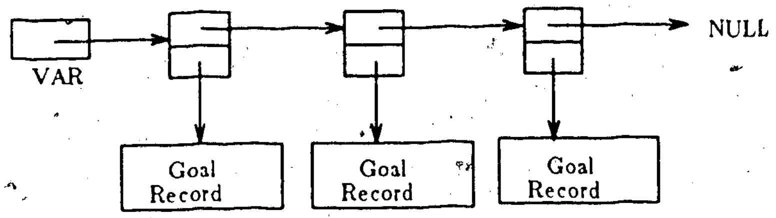


Figure 4.6 A Suspension List

4.3.5. Locks

More than one process may attempt to update the same run-time structure concurrently. To enforce exclusive access of common data, the model associates a lock with the ready goal pool, each goal record, each guard tuple and each variable. To obtain a lock, a process will busy-wait until it is available. This is acceptable provided the architectural assumptions in Section 4.1 are satisfied. A major problem in a system employing locking mechanism is the possibility of *deadlock*; the discussion of this issue is delayed until Chapter 5.

4.4. Algorithm

The algorithm, specified as pseudo code in Section 4.4.2, is executed by every process in the model. The followings are the general features:

(1) Given a computation unit, a process tries to advance the computations along the OR-branches in sequence. When the computation along an OR-branch cannot be advanced further in this computation unit (e.g. it fails, is aborted or spawned into other computation units), the process attempts another OR-branch. When the computations in all the OR-branches cannot be advanced further in this unit, the process tries another computation unit.

(2) The duration of a critical region is kept to minimal by releasing locks immediately when they are no longer needed and re-obtaining them later when necessary. Hence the pseudo code is interspersed with lock and unlock operations.

(3) The status of a clause in the Status_Vector of a goal record is checked before making any change. The change is made only if the clause is in the expected status, otherwise the process assumes that the computation has already been advanced by another process, so it simply attempts another clause. This measure is provided to avoid duplicating the computations unnecessarily.

4.4.1. Initialization

Given a goal clause of the form:

$$<\text{-}\ G_1, \ldots, G_n. \qquad n \leq 0.$$

The model replaces it by the following clauses:[13]

$$<\text{-}\ \text{TOP\_GOAL}.$$
$$\text{TOP\_GOAL}\ <\text{-}\ G_1, \ldots, G_n|. \qquad [\text{C0}]$$

The new goal clause contains only a goal, *TOP_GOAL*, which cannot be confused with any goal in the user program. TOP_GOAL's only applicable clause, C0, is added to the user program.

A goal record and a guard tuple is created and initialize for TOP_GOAL. Since TOP_GOAL is not contained in any guard, the Clause, Father, and Guard_Count

[13] For illustration purpose here, clauses are expressed in the syntax of Concurrent Prolog.

fields of the guard tuple are set to NULL, NULL and 1, respectively, as shown in Figure 4.7. TOP_GOAL is the only goal in the model having such guard tuple; this feature is used to detect program termination (See Section 4.4.2.5).

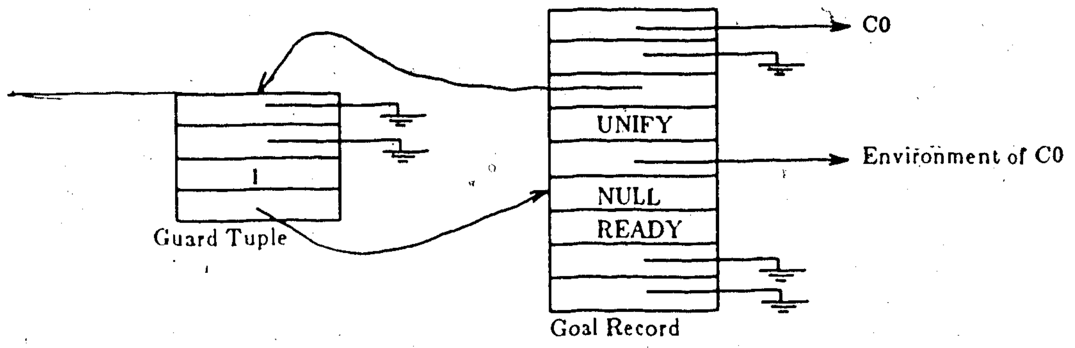


Figure 4.7 Data Structures for the Top Level Goal

The pseudo code in Section 4.4.2 assumes that the data structures in Figure 4.7 have already been set up, and TOP_GOAL has already been added into the ready goal pool.

4.4.2. Pseudo Code

The pseudo code of the algorithm executed by each process is presented below. The code segments are interspersed with comments explaining the design ideas. Inside the code, statement labels, subroutine names, macro names and locking operations are highlighted in **bold print**. Names of local variables and arguments are in *italics*. For the sake of conciseness, some abbreviations are used, for example, "**lock** *current goal*" means "lock the goal record referenced by the local variable current goal."

4.4.2.1. Getting Started

After initialization, every process tries to get a goal record from the ready goal pool using the following code segment:

```
START:
    GET_CURRENT_GOAL. {explained below}
    lock current goal.
    if Flag(current goal) ≠ READY {not an unsolved goal}
        unlock current goal.
        goto START. {try a new computation unit}
    end if.
    Flag(current goal) <- DEQUEUED.
    unlock current goal.
    current clause <- first clause in Procedure(current goal).
START_UNIFY:
    guard tuple <- Guard_Tuple(current goal).
    goto UNIFY. {can start head unifications}
```

A goal reduction starts at the **START** label. The **START_UNIFY** label is used when a recursive commitment occurs (see Section 4.4.2.5). The macro **GET_CURRENT_GOAL** dequeues current goal from the ready goal pool. It has the following code segment:

```
GET_CURRENT_GOAL:
    registered <- FALSE.
LOOP:
    if ready goal pool is empty
        if not registered {first time can't get a goal}
            lock idle_num.
            idle_num <- idle_num + 1. {I am idle!}
            unlock idle_num.
            registered <- TRUE.
            if idle_num = number of processes in the system
                report deadlock.
                exit.
            end if.
        end if.
        goto LOOP. {busy wait until there is a ready goal}
    end if.
    lock ready goal pool.
    if ready goal pool is empty {other processor seizes the goal}
        unlock ready goal pool.
        goto LOOP. {bad luck, try again}
    end if.
    if registered {has previously claimed idle, reset it}
        lock idle_num.
        idle_num <- idle_num - 1.
        unlock idle_num.
    end if.
    current goal <- dequeued goal record from ready goal pool.
    unlock ready goal pool.
```

The major concern here is the detection of deadlock caused by user program. A simple case is shown in the following program:

```
        <- p(X?), q(X?).

        p(1).
        q(1).
```

When the goal clause is invoked, both p and q will be suspended on X, and will be re-activated only when another goal instantiates X (which will never happen). While this simple case may be detected by a compiler, the general case needs a run-time deadlock detection mechanism.

The deadlock detection mechanism employed in this model is simple. It is based on the idea: "the system deadlocks when all the processes are idle before program termination." A process increments the shared variable idle_num, used to record the number of idle processes, in its first time to obtain a goal record from an empty ready goal pool. It decrements idle_num when it finally gets a goal record, and has previously incremented idle_num. A deadlock is implied when idle_num reaches the number of processes in the system. The nestedly locked critical region for getting a goal record and decrementing idle_num is necessary, otherwise a process may report deadlock while another process gets a goal record and is able to proceed.

4.4.2.2. Head Unification

The code segment for head unification is shown below:

```
UNIFY:
    if *current clause* = NULL {no more clause}
        **NO_MORE_CLAUSE**. {explained below}
    end if.
    **lock** *current goal*.
    case (status of *current clause*) of
     COMMITTED: {*current clause* has already committed}
        **unlock** *current goal*.
        goto **START**. {try a new computation unit}
     FAIL, KILLED, SPAWN, SPAWNING, SPAWNED, UNIFYING:
      {*current clause* has failed or been aborted, or is being attempted by another process}
        **unlock** *current goal*.
        **TRY_NEXT_CLAUSE**. {explained below}
     COMMIT: {*current clause* is ready to commit}
        **unlock** *current goal*.
        goto **COMMIT**.
     UNIFY: {*current clause* is ready to unify}
        status of *current clause* <- UNIFYING.
        **unlock** *current goal*.
        attempt the unification.
        case (result of unification) of
         SUCCESS:
            **lock** *current goal*.
            **VERIFY_UNLOCK**(UNIFYING). {explained below}
            status of *current clause* <- SPAWN.
            **unlock** *current goal*.
            goto **GUARD_SPAWN**.
         FAILURE:
            **lock** *current goal*.
            if status of *current clause* ≠ KILLED
                status of *current clause* <- FAIL.
            end if.
            **unlock** *current goal*.
            **TRY_NEXT_CLAUSE**.
         SUSPENSION:
            **lock** *current goal*.
            if status of *current clause* = UNIFYING
                status of *current clause* <- UNIFY.
            end if.
            **unlock** *current goal*.
            *V* <- set of variables causing the suspension.
            call **DO_SUSPEND**(*current goal*, *V*).
            **TRY_NEXT_CLAUSE**.
        end case.
    end case.
```

The **UNIFY** label is reached when the process attempts to unify a goal with a clause head. If the clause has already committed, the reduction is abandoned and the process tries another computation unit. If the head unification has previously failed or been aborted, or it is currently being attempted by another process, the process tries another clause. If the clause is ready to commit, the commitment step is attempted. Otherwise the clause status is set to **UNIFYING** and the head unification is attempted. If it succeeds, the computation proceeds to the guard spawning step. If it is suspended, the clause status is reset to **UNIFY** (so that head unification can be retried later), the goal is added into the appropriate suspension list(s) and the process tries another clause. If it fails, the clause status is set to **FAIL** if that clause has not yet been aborted (by other process), and another clause is tried.

A process tries the next clause using the macro **TRY_NEXT_CLAUSE**. Its code segment is shown below:

```
TRY_NEXT_CLAUSE:
    current clause <- next clause in Procedure(current goal).
    goto UNIFY
```

A process always verifies the clause status, using the macros **VERIFY** and **VERIFY_UNLOCK**, before making any status modification. If the current clause status is not expected, it assumes that another process has already advanced the computation of that clause (e.g. has aborted the clause); so it proceeds to try another clause. The two macros are similar, except that **VERIFY_UNLOCK** also unlocks current goal when unexpected status is encountered:

```
VERIFY(expected status):
    if status of current clause ≠ expected status
        TRY_NEXT_CLAUSE.
    end if.

VERIFY_UNLOCK(expected status):
    if status of current clause ≠ expected status
        unlock current goal.
        TRY_NEXT_CLAUSE.
    end if.
```

The macro **NO_MORE_CLAUSE** is used when all the clauses in the Procedure field are exhausted. Its code segment is shown below:

```
NO_MORE_CLAUSE:
    if Procedure(current goal) = NULL  {no applicable clause}
        call DO_FAILURE(current goal).
    else
        fail <- FALSE.
        lock current goal.
        if (every entry in STATUS_VECTOR(current goal) is FAIL)
            fail <- TRUE.
        end if.
        unlock current goal.
        if fail
            call DO_FAILURE(current goal).
        end if.
    end if.
    goto START.  {try a new computation unit}
```

The goal fails if it has no applicable clause or all its clauses have failed. In those cases the goal is marked as **DEAD** and the failure is propagated (by calling **DO_FAILURE**). Note that **DO_FAILURE** is recursive and hence the code avoids calling it directly in the critical region. The subroutines **DO_SUSPEND** and **DO_FAILURE** are explained in Section 4.4.2.6.

4.4.2.3. Guard Spawning

The following code segment implements the guard spawning step:

```
GUARD_SPAWN:
    pending goal pool <- NULL.  {initialize}
    lock current goal.
    VERIFY_UNLOCK(SPAWN).  {is current clause status still SPAWN?}
    status of current clause <- SPAWNING.
    unlock current goal.
    if guard(current clause) is empty
        lock current goal.
        VERIFY_UNLOCK(SPAWNING).
        status of current clause <- COMMIT.
        unlock current goal.
        goto COMMIT.  {empty guard, commits immediately}
    end if.
    create a guard tuple T for this guard.
    Clause(T) <- current clause.
    Father(T) <- current goal.  {link from T to current goal}
```

```
for each goal g in this guard
      VERIFY(SPAWNING).
      call CREATE_GOAL(g, T).
            {the goals are added into pending goal pool as a side effect}
end for.
add goal records in pending goal pool into Goal_List(T).
Guard_Count(T) <- number of goal records in pending goal pool.
lock current goal.
VERIFY_UNLOCK(SPAWNING).
current clause's entry in GT_Vector(current goal) <- T.
      {link from current goal to T}
lock ready goal pool.
add goal records in pending goal pool into ready goal pool.
unlock ready goal pool.
status of current clause <- SPAWNED.
unlock current goal.
TRY_NEXT_CLAUSE.
```

The **GUARD_SPAWN** label is reached when a clause is ready to spawn its guard. If the guard is empty, the process proceeds immediately to the commitment step. Otherwise a guard tuple for the guard is created, initialized, and bi-directionally linked to its father goal record. The clause status is updated to **SPAWNING**, and the goal records for the guard goals are created and added to ready goal pool. Since a clause may be aborted in the middle of the spawning step, the use of the *pending goal pool* and the nestedly locked critical region ensures that the change of clause status to **SPAWNED** and the insertion of pending goals into the ready goal pool constitute an atomic action. The subroutine **CREATE_GOAL** is explained in Section 4.4.2.6.

4.4.2.4. Commitment

The following code segment implements the commitment step:

```
COMMIT:
      lock current goal.
      VERIFY_UNLOCK(COMMIT).
      do language-specific commitment actions.
      case (result of commitment actions) of
        FAILURE:
            status of current clause <- FAIL.
            unlock current goal.
            TRY_NEXT_CLAUSE.
        SUSPENSION:
            unlock current goal.
        $V$ <- set of variables causing the suspension.
        call DO_SUSPEND(current goal, $V$).
        TRY_NEXT_CLAUSE.
    SUCCESS:
        status of current clause <- COMMITTED.
        for each clause $C$ in Procedure(current goal) $\neq$ current clause
            status of $C$ <- KILLED. {abort sibling clauses}
        end for.
        unlock current goal.
        for each aborted clause $C$'s entry $T$ in GT_Vector(current goal)
            if $T \neq$ NULL {guard already spawned}
                for each non-DEAD goal $G$ in Goal_List($T$)
                    call DO_KILL($G$). {propagate abortion}
                end for.
            end if.
        end for.
        if language-specific actions instantiated variables
            $V$ <- set of instantiated variables.
            call DO_WAKEUP($V$). {re-activate suspended goals}
        end if.
        lock current goal.
        Flag(current goal) <- DEAD. {mark as reduced}
        unlock current goal.
        committed clause <- current clause.
        current clause <- Clause(Guard_Tuple(current goal)). {father clause}
        current goal <- Father(Guard_Tuple(current goal)). {father goal}
        goto BODY_SPAWN.
    end case.
```

The **COMMIT** label is reached when an empty guard is encountered in the guard spawning step (see Section 4.4.2.3), or when the guard of a clause is solved (see Section 4.4.2.5). The language-specific commitment actions are simple for PARLOG and Guarded Horn Clauses, but it involves *environment unification* for Concurrent Prolog. If the result of these actions fails, the process sets the clause status to **FAIL** and tries another clause. If it is suspended, the goal is added into the appropriate suspension list(s), and another clause is tried. If it succeeds, the process sets the clause status to **COMMITTED**, aborts all sibling clauses and their descendants (by calling **DO_KILL**), re-activates suspended goals (if any), claims to be reduced and proceeds to the body spawning step. The subroutines **DO_KILL** and **DO_WAKEUP** are explained in Section 4.4.2.6.

4.4.2.5. Body Spawning

The following code segment implements the body spawning step:

```
BODY_SPAWN:
    pending goal pool <- NULL. {initialize}
    if body(committed clause) is empty
        lock guard tuple.
        if Guard_Count(guard tuple) = 1 {last goal in a guard solved}
            unlock guard tuple.
            if current goal = NULL {top level goal solved}
                report program termination.
                exit.
            end if.
            lock current goal. {father of reduced goal}
            if status of current clause ≠ SPAWNED {not in expected status}
                unlock current goal.
                goto START. {try a new computation unit}
            end if.
            status of current clause <- COMMIT. {recursive commitment}
            unlock current goal.
            goto START_UNIFY.
        end if.
        decrement Guard_Count(guard tuple) by 1.
        unlock guard tuple.
        goto START. {try a new computation unit}
    end if.
    for each goal G in body(committed clause)
        if status of current clause ≠ SPAWNED
            goto START.
        call CREATE_GOAL(G,guard tuple).
            {the goals are added into pending goal pool as a side effect}
    end for.
    lock current goal.
    VERIFY_UNLOCK(SPAWNED).
    lock guard tuple.
    add goal records in pending goal pool into Goal_List(guard tuple).
    Guard_Count(guard tuple) <- Guard_Count(guard tuple) +
        number of goal records in ending goal pool - 1.
    unlock guard tuple.
    lock ready goal pool.
    add goal records in pending goal pool into ready goal pool.
    unlock ready goal pool.
    unlock current goal.
    goto START. {try a new computation unit}
```

The label **BODY_SPAWN** is reached after a clause has committed. If the committed clause body is empty, then the reduced goal is solved. The process examines the guard count to handle the following cases: (1) If guard count is 1, then the solved goal is the last one in the guard. It is a program termination if the goal has no father goal, or a recursive commitment otherwise. In the latter case the father goal commits to the father clause. (2) If guard count is not 1, it is decremented and another computation unit is tried.

If the committed clause body is not empty, the process creates and initializes the goal records for its goals, updates the guard count, adds the new goals into the ready goal pool and the goal list, and tries a new computation unit.

4.4.2.6. Subroutines

The followings are the subroutines used in the pseudo code:

```
CREATE_GOAL(g, GT):  {g is a goal, GT is a guard tuple}
    create a goal record G for g, add it to pending goal pool.
    initialize the Goal, Procedure, Arguments, and Env_Vector field.
    Flag(G) <- READY.
    Guard_Tuple(G) <- GT.  {point to father guard tuple}
    set every entry in Status_Vector(G) to UNIFY.  {clauses ready to unify}
    set every entry in GT_Vector(G) to NULL.  {guards not yet spawned}
    return.
```

CREATE_GOAL is used to create and initialize a goal record. It is called in the guard spawning and body spawning steps. The created goals are added into the pending goal pool as a side effect.

```
DO_FAILURE(G):  {G is a goal record}
    if Flag(G) ≠ DEAD  {G still exists}
        lock G.
        Flag(G) <- DEAD.  {mark as failed}
        unlock G.
        f <- Father(Guard_Tuple(G)).  {father goal}
        if f = NULL  {G is the top level goal}
            report computation failure.
            exit.
        end if.
        for each sibling goal S of G[14]
            call DO_KILL(S).  {propagate failure across conjunction}
        end for.
        c <- Clause(Guard_Tuple(G)).  {father clause}
        lock f.
            status of c <- FAIL.  {father clause fails}
        unlock f.
        if Status_Vector(f) is all FAILs
            call DO_FAILURE(f) {propagate failure up the AND-OR tree}
        end if.
    end if.
    return.
```

DO_FAILURE is called when a goal fails. If the goal is already the root of the AND-OR tree, then the computation fails. Otherwise the failure should be propagated across the conjunction since a conjunction containing a failed goal also fails. The failed conjunction causes the failure of its father clause. Hence the father goal is re-activated to check if all of its applicable clauses have failed. If so the father goal fails and propagates the failure to its siblings and up the AND-OR tree (by recursively calling **DO_FAILURE**). Note that **DO_KILL** is called in cross-conjunction failure propagation, since the descendants of the sibling goals should also be aborted.

```
DO_KILL(G): {G is a goal record}
    if Flag(G) ≠ DEAD {G still exists}
        lock G.
        Flag(G) <- DEAD. {marked as aborted}
        set all entries in Status_Vector(G) to KILLED.
            {propagate abortion across the clauses}
        unlock G.
        for each entry gt in GT_VECTOR(G)
            if gt ≠ NULL {guard already spawned}
                for each goal record d in Goal_List(gt)
                    call DO_KILL(d).
                     {propagate abortion down the AND-OR tree}
                end for.
            end if.
        end for.
    end if.
    return.
```

DO_KILL is called when a goal is aborted. An aborted goal is marked **DEAD** and has all its clauses aborted (i.e. status set to **KILLED**).[15] The abortion should be propagated down the AND-OR tree so that the computations in all the spawned descendants of the aborted clauses can also be aborted. Prompt propagation of failure and abortion is important when the AND-OR tree is large (as can be imagined in many

[14] A sibling of G is a goal record S in the Goal_List of Guard_Tuple(G) where $G \neq S$ and Flag(S) $\neq$ DEAD.

non-toy applications), it avoids redundant efforts spent in exploring a sub-tree rooted by a DEAD goal and hence may result in substantial performance improvement.

```
DO_SUSPEND(G,V): {G is a goal record, V is a set of variables}
    if Flag(G) ≠ DEAD {G still exists}
        for each variable v in V
            if G is not currently suspended on v
                create a suspension record r.
                Goal_Record(r) <- G.
                lock v.
                add r into the suspension list of v.
                unlock v.
            end if.
        end for
    end if.
    return.
```

DO_SUSPEND is called when a goal is suspended on some variable(s) during unification. Suspension record(s) are created for the goal and added into the appropriate suspension list(s). The existence check before suspension record insertion avoids multiple suspensions of a goal on the same variable. It is not strictly necessary, but is employed here with the assumption that the suspension list is usually short; the cost of the check is lower than that of handling duplicated suspension records.

```
DO_WAKEUP(V): {V is a set of variables}
    for each variable v in V
        if v's suspension list l ≠ NULL
            for each suspension record r in l
                G <- Goal_Record(r).
                if Flag(G) = DEQUEUED {G still exists and not ready}
                    lock G.
                    Flag(G) <- READY.
                    unlock G.
                    lock ready goal pool.
                    add G to ready goal pool
                    unlock ready goal pool.
                end if.
            end for.
        end if.
    end for.
    return.
```

[16] Since all the clauses are indiscriminately aborted, it is possible that a COMMITTED or FAILED clause may also be KILLED. This is anticipated in the model and does not create chaos.

DO_WAKEUP is called when some variables are instantiated. The goals suspended on these variables are re-activated by inserting them back into the ready goal pool. The existence check before ready goal pool insertion is necessary; since the model keeps only one record for a goal, two occurrences of the same goal cannot be in the ready goal pool simultaneously. Without the check, an insertion may spoil the link structure of the pool, resulting in lost goals.

It is worth mentioning that in the model, a goal may be suspended on a set of variables, but re-activated on a per-variable basis. It is hence possible that a goal may go through several iterations of re-activation and re-suspension before it is actually waken up. This is acceptable since such an iteration is not expensive, and only simple data structures are needed to support the scheme. Also, for the sake of simplicity, no information of the clause(s) containing the variable(s) causing the suspension is made. A re-activated goal simply re-tries all the applicable clauses in sequence to find the appropriate clause(s).

4.5. Summary

We have proposed an execution model for the three parallel logic programming languages Concurrent Prolog, PARLOG and GHC on a shared memory multiprocessor architecture. The architectural assumptions are presented with emphasis on cheap lock operations, as they are heavily used in the design. For the data structures used in the model, the Guard_Tuple and GT_Vector fields of the goal record, and the Father and Goal_List fields of the guard tuple collectively provide all the required linkages to reveal the AND-OR tree structure, allowing failure and abortion to be propagated up, down and horizontally in the tree. The guard count field of a guard tuple supports the detection of recursive commitment, which is the only means in the model to re-activate a goal when the spawned guard of one of its clauses has successfully been solved, urging it to commit. For the algorithm, the major concepts are the viewing of the computation necessary to satisfy a goal with its applicable clauses as a computation unit, and the encoding of computations by a finite state machine. This coarse-grained approach makes parallelism easier to control, which is significant for a moderately parallel system. Appendix A3 depicts the run-time structures created in the model when executing a small Concurrent Prolog program. Chapter 5 provides an analysis of the model.

Chapter 5

Analysis of the Proposed Execution Model

This chapter is an analysis of the execution model proposed in Chapter 4. The following issues are considered in the analysis: (1) the correctness of the execution model; (2) the kinds of parallelism supported, and the execution model's potential for exploiting them; (3) the effects of architectural issues on the design decisions; (4) the considerations in supporting the three target languages; and (5) the justification of the execution model based on the factors stated in Section 1.1.1.

5.1. Correctness Verification

As the correctness of the execution model depends on the proper function of its locking mechanism and the precise implementation of the desired operations, the problem of correctness verification can itself be split into the following subproblems:

(1) Verify that the execution model is deadlock-free.

(2) Verify that the execution model enforces exclusive access in update and protects all critical actions.

(3) Verify that for every goal reduction, a goal will commit to at most one clause.

(4) Verify that the goal suspension/re-activation scheme is correct.

(5) Verify that the failure/abortion propagation scheme is correct.

Sections 5.1.1 to 5.1.5 sketch the informal solutions to these subproblems, the ideas to solve the first three subproblems originate from Levy's work [Lev86].

5.1.1. Deadlock Avoidance

To avoid deadlock, the algorithm in Section 4.4 ensures that a process requires to lock at most two objects at any time. When locks on two objects are needed, they are obtained in a strict order; so deadlock will never happen. However, deadlock can still be introduced into the system by a user program. This situation cannot be avoided but only be detected, a simple detection algorithm is provided in Section 4.4.2.1.

5.1.2. Enforcing Exclusive Access and Protecting Critical Actions

To enforce exclusive access, every update to shared objects (e.g. the ready goal pool, goal records and guard tuples) is locked. There are two kinds of critical actions that should be protected: (1) complex atomic actions whose execution, when being interrupted, may cause inconsistency; (2) actions with side effects which should only be performed once. An example of the former is the atomic action for dequeueing a goal and updating the idle_num in the macro **GET_CURRENT_GOAL** (see Section 4.4.2.1). Examples of the latter are binding environment updates (in unification and commitment) and goal spawning.

Each complex atomic action in the pseudo code is protected by a lock or nested locks so that the action as a whole can be carried out to completion without interruption. A process changes the clause status (e.g. from UNIFY to UNIFYING, from SPAWN to SPAWNING) before performing any action with side effects so that the other processes, detecting the status change, can avoid repeating the action.

5.1.3. Uniqueness of Committed Clause

To guarantee that a goal commits to at most one clause, the code segment in Section 4.4.2.4 ensures that the status change of the committed clause to **COMMITTED** and the status changes of its siblings to **KILLED** constitute an atomic action. Since a process checks the clause status before making any change, a **KILLED** clause, even if it is ready for commitment, will never be tried.

5.1.4. Correctness of the Goal Suspension/Re-activation Scheme

A correct goal suspension/re-activation scheme should ensure that (1) every suspended goal which is re-activatable (i.e. not caused by user-program deadlock) will eventually be awakened up, and (2) only active goals are re-activated. As stated in Section 4.4.2.6, a re-activated goal tries the applicable clauses in sequence to continue suspended computations. In cases where computations in several clauses are suspended, if OR-parallelism is not supported, this unfair re-activation scheme may result in indefinite neglect of a clause as the re-activation of its computation may always be intercepted by the preceding one(s). This situation is problematic since the indefinitely neglected clause may be the only one that can lead to a successful computation. However, committed OR-parallelism, though in a restricted form, is supported by the execution model (see Section 5.2); so the abovementioned situation will never happen.

Since the subroutine **DO_WAKEUP** never adds a goal whose flag is not **DEQUEUED** into the ready goal pool, the execution model ensures that only active goals are re-activated, and the link structure in the pool is never spoiled. Moreover, since every non-**READY** goal dequeued from the pool is discarded (see Section 4.4.2.1), even a goal which becomes **DEAD** (because of its own reduction, or the failure/abortion propagation) after its re-activation will not be picked up by a process and indiscriminatedly processed.

5.1.5. Correctness of the Failure/Abortion Propagation Scheme

A correct failure/abortion propagation scheme should ensure that (1) the failure or abortion is propagated to the relevant goals only; (2) nonterminating propagation is avoided (optional). It is easy to verify (1) by inspecting the pseudo code in Section 4.4.2.6. To verify (2), it should be noted that the failure/abortion propagation is directional (i.e. either up, down, or horizontal in the AND/OR tree), a goal is marked as **DEAD** after the propagation, and the propagation stops at a **DEAD** goal; thus each goal in the AND/OR is visited at most once. As a result, the propagation will eventually stop if the number of goals in the AND/OR tree is finite.

However, nonterminating propagation may happen if abortion is propagated downwards an infinite branch, and the process aborting the nodes always lags behind the process(es) [illegible] the nodes in the branch [CiH84]. This situation is rare and not fatal (i.e. [illegible]ects performance), so the algorithm in Chapter 4 provides no means of handling [illegible] although a correct scheme can quite easily be devised. The reason for the nontermination is that the generation of the goals in an aborted subspace is faster than the abortion of those goals. If the initial caller of **DO_KILL** locks the ready goal pool before the call and unlocks it afterwards, no new goal can be generated before the abortion finishes; thus the abovementioned race condition is avoided. This solution is only a first attempt, it limits the exploitable parallelism and is practical only if the number of processors in the system is small. Instead of letting the caller of *DO_KILL* monopolize the abortion, a second solution is to provide a shared work-pool for abortable goals, and have all the idle processes (because of the locked ready goal pool) participate the abortion. A process fetches an abortable goal from the work pool, marks it as **DEAD**, changes all the clause status to **KILLED**, and adds the aborted goal's sons into the work pool. Normal operation is resumed when the work pool is empty (i.e. abortion is completed). Results of simulation or pilot implementation are needed to analyze the trade-off between the benefits (i.e. the exploitation of idle resources) and costs (i.e. the overheads of maintaining the abortion work-pool) of this scheme.

5.2. Parallelism Potential

The execution model exploits AND-parallelism, stream parallelism and committed OR-parallelism in logic programs. Regarding AND-parallelism, it is clear that the goals from a spawned guard or clause body can be evaluated in parallel, the only restrictions are the number of available processors in the system and the synchronization requirements. Regarding stream parallelism, the evaluation of the consumer and producer of a stream can be synchronized by the goal suspension/re-activation scheme as the stream variable is incrementally instantiated; the only deficiency is the execution model's inability to adjust the degree of stream parallelism according to system workloads.

Although the applicable clauses of a goal are tried in sequence, and a **DEQUEUED** goal is monopolized by a process most of the time; a restricted form of committed OR-parallelism can still be exploited by the execution model. It should be noted that the unfairness resulting from sequential clause application is bounded; a process switches attention to the next clause when the computation of a clause cannot be advanced further (i.e. either failed, suspended, or spawned). As a result the computation of any clause is never indefinitely neglected. Committed OR-parallelism is exploited in the execution model when a goal spawns the guard of a clause and then tries the next clause, leaving the spawned goals to be processed by others. This leads to the concurrent computations in disjunctive OR-branches. Also, many processes can concurrently attempt to reduce a goal with its applicable clauses. This happens when a process is still applying some untried clauses to a goal, but either (1) suspended computations in some of the goal's previous clauses are re-activated and processed by others, or (2) a recursive commitment propagates to the goal, urging it to commit to a previous clause, or both.

5.3. Architectural Issues

The presence of shared memory strongly influences the design decisions of the execution model, especially the policies of busy-waiting for a lock and frequent releasing and re-holding of locks to minimize the period of a critical region. These policies may not be profitable if the execution model is to be implemented on non-shared memory multiprocessor architectures. The features of a modified model, designed for execution of Flat Concurrent Prolog [Mie84] on those architectures, is provided in Levy's work [Lev86]. The most notable design decisions are to allow only one process to advance the computation in a computation unit at any time, and to avoid lock operations unless they are necessary.

It is desirable if the execution model can be adapted to be suitable for implementation on non-shared memory multiprocessor architectures. A first step should be the minimization of inter-connections among the run-time structures. A naming scheme for goal records, like that in the OR-parallel token machine [Cil184], which reveals the AND/OR tree structure can be employed for this purpose. In general, supporting the execution of logic programs on a non-shared memory multiprocessor architecture is more difficult than on a shared memory one. For the former, not only the control of parallel activities but also the management of binding environments (see Chapter 6) are more complicated than the latter.

5.4. Considerations in Supporting the Target Languages

As can be seen from Chapters 2 and 4, the staged goal reduction concept of the execution model fits the operational semantics of Concurrent Prolog, but deviates from those of the other two languages. In this section, the issues to be considered in supporting the three target languages are discussed.

5.4.1. Supporting Concurrent Prolog

The only deficiency of the model in supporting Concurrent Prolog is in the commitment step (see Section 4.4.2.4) where the language-specific commitment action (i.e. environment unification) may lock the goal record for an unacceptable long period. This problem can be solved by introducing a new clause status *COMMITTING*. A clause with this status means that a goal is committing to it. The language-specific commitment action can be extracted out of the critical region in the following way: after having verified that the clause status is COMMIT, the process changes it to COMMITTING and unlocks the goal record. Then the language-specific commitment action is performed. When the action completes, the process changes the clause status to FAIL if the result of the action is failure. If the result is success or suspension, the process verifies that the clause status is still COMMITTING, and if so changes it to COMMITTED, or back to COMMIT, respectively. The other parts of the code is unchanged. Note that the introduction of the new clause status COMMITTING is necessary since environment unification has side effects and hence should only be performed once (see Section 5.1.2).

5.4.2. Supporting PARLOG

There are two major discrepancies between the proposed execution model and PARLOG's operational semantics: (1) in PARLOG, the head unification's input matching is evaluated in parallel with the guard, and the output matching is evaluated in parallel with the clause body; (2) PARLOG provides sequencing constructs (i.e. the "&" and ";" operators) in both clause selection and goal evaluation (see Section 2.2.2.2).

The first discrepancy disappears after a PARLOG program P has been compiled into its Kernel PARLOG counterpart KP (see Section 2.2.2.2). To narrow the second discrepancy is tricky. For the sequential OR ";" operator, the procedure field of a goal record is changed to a list of sequential components, each of which is a list of parallel-OR clauses. The sequential components are tried in sequence; when all the clauses in a component fail, the next component is tried. The number of entries in the Status_Vector, Env_Vector and GT_Vector fields of a goal record is changed to the maximum number of clauses in all components. The methods of getting the next clause and detecting the end of a procedure are changed accordingly.

Since the sequential AND operator "&" and the parallel AND operator "," have no precedence, any arbitrarily complex expression is hence possible; this makes the support of the language in the execution model difficult. A first-attempt solution is to add a *clause descriptor* field in the guard tuple. The purpose of this field is to control the sequencing of goal evaluation in the guard. At the beginning of a guard computation, or every time a goal is reduced, the descriptor is updated and then parsed to find the goal(s) that should be created and added into the ready goal pool. The idea of the clause descriptor is inspired by the AND-OR tree model in Gregory's thesis ([Gre85], Chapter 6), but the hierarchy of the AND-processes in that model is collapsed into the a clause descriptor in this proposal. At present the idea is not sufficiently develo how to represent the clause descriptor and how to parse it are still vaguely defined. However, it should be noted that for efficient execution, a PARLOG program should be compiled into Kernel PARLOG, and further compiled into other abstract models (e.g. the AND-OR tree model) that fits the characteristics of the target architectures ([Gre85], pages 175-177). There is a gap between Kernel PARLOG and the proposed execution model as their target architectures have different granularities, most likely some abstract model(s) should be placed between them to bridge the gap. The solution to the sequential AND problem had better be delayed until those intermediate models are defined.

5.4.3. Supporting GHC

The semantics of GHC allow no restriction in parallel activities in goal reductions provided that the suspension rule and trust rule are not violated. This freedom may be exploitable in a fine-grained architecture [KKI85], but should be suppressed in a moderately-parallel architecture where striving for low overheads is a key objective. There is, in general, no harm in performing the different activities in a goal reduction step in sequence, as suggested by the proposed execution model. In fact, for a moderately-parallel architecture, the advanced clause body evaluation before commitment is not profitable since most of the efforts, except that for the committed clause, are wasted. If unification and guard/clause evaluation are needed to be performed in parallel, it may still be achieved by adopting the compilation method like that in PARLOG.

5.5. Justification

The proposed execution model has quite high scores on some of the justification factors in Section 1.1.1. It is relatively easy to realize, as indicated by the closeness of the pseudo code to an implementation language (e.g. C or Pascal). Its overheads are low: (1) a process is idle only when the ready goal pool is empty, (2) busy-waiting for a lock is a cheap way to synchronize the execution of processes (vs. message passing), and (3) the costs to manage the data structures are low. The model also exploits various kinds of parallelism, although committed OR-parallelism is implemented in a restricted way. The coarse-grain approach in viewing the computation necessary to satisfy a goal with its applicable clauses as a unit implies a lower potential of parallelism, however, this fits the target architecture, which is moderately parallel, properly.

Chapter 6

Conclusions

6.1. Summary of the Thesis

In this thesis, we investigated the various aspects of exploiting parallelism in logic programs, and proposed an execution model on a shared memory multiprocessor architecture for the parallel logic programming languages Concurrent Prolog, PARLOG and GHC. We started by reviewing the basic concepts in logic programming where one of its distinguished features, non-determinism, is the major source from which various forms of parallelism originate. A conventional implementation of Prolog, the first logic programming language, does not exploit this potential, but simulates the non-determinism by sequential search and backtracking. However, improving the execution speeds of logic programs by incorporating parallelism is promising (though difficult) and is reinforced by the advances in hardware technologies.

In Chapter 2, we studied the four different forms of parallelism in logic programs, and the difficulties of their realization. We then examined the features of the three abovementioned parallel logic programming languages which were designed to support concurrent computation and parallel execution. In Chapter 3, we surveyed the typical parallel execution models for Prolog, its variances and the three parallel logic programming languages. Most of these models are parts of some ambitious long-range projects, where various problems must effectively be solved before the models can actually be realized. These considerations have led us to propose a simpler execution model, described in Chapter 4, for the three parallel logic programming languages on a shared memory multiprocessor architecture. The aim of the model is to exploit moderate parallelism, with emphasis on ease of realization and low overheads.

The proposed execution model is strongly influenced by Levy's work [Lev86], but incorporates a number of extensions and modifications. The major extension is the additional pointers used to set up the linkages among the goal records and guard tuples, revealing the AND/OR tree structure. This allows failure and abortion to be propagated up, down and horizontally in the tree. Compared with the original model, the proposed model consumes more memories and incurs additional overheads in maintaining the link structure. However, it is justifiable as memories are cheap, and the additional overheads are low (i.e. only one-shot set-up costs are needed), but the resulting prompt failure/abortion propagation may result in substantial performance improvement, especially for a wide and deep AND/OR tree. Other extensions include the initialization of data structures for the top-level goal, and the detection of program termination and software deadlock. A few modifications are also made, mainly to improve the original model.

The proposed execution model, carrying over Levy's ideas, is based on the work-pool approach. The major concepts are the viewing of the computation necessary to satisfy a goal with its applicable clauses as a unit, and the separation of the step into different stages so that computations can be encoded by a finite state machine. This coarse-grained approach allows parallelism to be easier to control, which is significant in a moderately parallel system. Finally in Chapter 5, we provided an informal correctness verification of the proposed model, an analysis of its parallelism potential, and some considerations of supporting the three parallel logic programming languages on the model. The proposed model is justified for its fitness to the target architecture, ease of realization, low overheads, and exploitation of various forms of parallelism.

6.2. The Management of Binding Environments

While this thesis concentrated on the discussion of controlling parallel activities in an execution model, another important problem, the management of binding environments, was given little attention. This problem is important since for effective computation, the variable bindings generated in a unification must be maintained if they are necessary for later unifications or the final answer constructions (see Section 1.2.4). In this section, a brief discussion of the problem is provided.

In a conventional Prolog implementation, there are basically two alternative approaches of representing the binding environments: *structure-sharing* [BoM72, Bru82] and *structure-copying* [Mel82]. In both approaches, the binding of a variable to a constant, another variable, or a ground complex term can be a pointer to the direct representation of the value. To represent the binding to a non-ground complex term, the structure-sharing approach uses two pointers: one to the pure code of the term, another to the binding environment containing the variables occur in the term; while the structure-copying approach creates a new copy of the term for the binding. A comparison of the two approaches, mainly on memory consumptions, is provided by Mellish [Mel82].

Additional problems reveal themselves when a parallel implementation is considered, though the basic ideas of the two above approaches can still be applied. A major problem is the need to manage separate OR-parallel binding environments, due to the mutually exclusive bindings made to the same goal variable by the OR-parallel computations. This problem is not serious for PARLOG and GHC, since such bindings are either disallowed or suspended before commitment, and only one of them can be made after commitment. However, the situation is quite complicated in Concurrent Prolog. The semantics of Concurrent Prolog allow bindings to goal variables to be made locally, but disallow them to be accessed outside the guard before commitment.

A natural approach to cope with this is to use the copying scheme, i.e. to make a copy of the goal for each unification so that goal variables are bound ~~in~~ the copied goal, not the original one. The copied and original goals are unified at commitment. However, because of the copying, the links for information flow from goals to guards are broken, creating unexpected results in guard evaluation that depend on this information. Two recently proposed schemes: ***multiple commitment*** and ***eager broadcasting*** [LeF86] are still not satisfactory; the former does not solve the problem, the latter solves the problem, but requires complex run-time structures and incurs high overheads. For parallel Prolog, various techniques such as the schemes of directory trees, hash windows, etc. are proposed to avoid unnecessary duplication of binding information; a brief survey of these schemes is provided by Levy and Friedmann ([LeF86], pages 19-24).

Another problem is the efficient support of binding environments when a distributed implementation is considered. The binding environments generated during computations are highly inter-connected, distributing them among different processing elements implies a complex variable reference scheme and additional message overheads in dereferencing a variable, which is a most basic operation in the system (e.g. Port Prolog [Lee84]). A possible solution to the above problem may be to completely ban the notion of binding environment, but literally substitute the bound variables in a goal with the terms (e.g. the AND/OR Process Model [Con83], and PIE [GTM84]). No dereference of variable is needed in this scheme, but the copying overheads may be unacceptably high, especially when there is a large amount of structure data. More research is needed in order to reach a satisfactory solution.

6.3. Future Work

As the work reported in this thesis represents only a preliminary step towards an actual implementation of a parallel logic programming system, many extensions and improvements are remained to be done. In this section, some suggestions for possible directions of future work are provided.

Before making any full-scale implementation of the execution model, a first step may be to obtain some performance measurements through simulations or pilot implementations to verify the design decisions. A possible direction is to implement the execution model on a uniprocessor running an operating system (e.g. Unix) where multiple processes are supported so that a multiprocessor architecture can be simulated. The major benefits of this approach are the ease of making architectural changes for experimental purposes, and the alleviation of the difficulties in porting the developed programs in the prototyping environment to the target system, due to their closeness. In a pilot implementation, unifications and binding environment management operations can be simulated; however, simulation parameters such as the average times of a unification step and each of those operations must be provided.

The performance measurements should provide information such as the effectiveness of the model's failure/abortion propagation scheme and the goal suspension/re-activation scheme, the ratio of lock operations to unifications performed by each processor, the ratio of processor utilization time to idle time, the growth rate of memory contentions with the increasing number of processors, and the effectiveness of the model in exploiting the parallelism in logic programs and supporting architectures (e.g. to produce plots of the speedups in running different programs on architectures with different number of processors). An analysis of this information can help to pinpoint the shortcomings of the model and make enhancements.

A second step may be to address the problems in managing the binding environments. Since the three target languages have different unification algorithms and different binding environment management requirements, a decision must be made to specifically support one of the languages first. A good starting point may be to support GHC or a simpler version of PARLOG (e.g. PARLOG without the sequential AND/OR operators) where the management of OR-parallel binding environments is not that complicated, when compared with Concurrent Prolog. Finally, to extend the implementation to a logic programming system, various problems, such as the design of the internal program codes, the device of a garbage collection scheme (on a multiprocessor system), and the provision of program development tools (e.g. debugging facilities), must be effectively solved.

References

[BaM73] G. Battani and H. Meloni, *Interpreteur du Language de Programmation Prolog*, Groupe d'Intelligence Artificielle, University d'Aix Marseille, Luminy, France, 1973.

[Bic84] L. Bic, Execution of Logic Programs on a Dataflow Architecture, *Proc. of the 11th Annual International Symposium on Computer Architecture*, Ann Arbor, Michigan, U.S.A., 1984, 290-296.

[Bis86] P. Bishop, *Fifth Generation Computers: Concepts, Implementations, and Uses*, Ellis Horwood Limited, 1986.

[Bow82] K. A. Bowen, Concurrent Execution of Logic, *Proc. of the First International Logic Programming Conference*, Marseille, France, September 1982, 26-30.

[BoM72] R. S. Boyer and J. S. Moore, The Sharing of Structure in Theorem Proving Programs, in *Machine Intelligence*, vol. 7, Bernard Meltzer and Donald Michie (ed.), Edinburgh University Press, 1972.

[Bru82] M. Bruynooghe, The Memory Management of Prolog Implementations, in *Logic Programming*, K. L. Clark and S.-A. Tarnlund (ed.), 1982, 83-98.

[BrP84] M. Bruynooghe and L. M. Pereira, Deduction Revision by Intelligent Backtracking, in *Implementations of Prolog*, J. A. Campbell (ed.), Ellis Horwood Limited, 1984, 194-215.

[Che84] M. H. M. Cheng, The Design and Implementation of the Waterloo Unix Prolog Environment, Master Thesis, Department of Computer Science, University of Waterloo, 1984.

[CiH84] A. Ciepielewski and S. Haridi, Control of Activities in the OR-Parallel Token Machine, *Proc. of IEEE International Symposium on Logic Programming*, Atlantic City, NJ, U.S.A., 1984, 49-57.

[Cla78] K. L. Clark, Negation as Failure, in *Logic and Data Bases*, Herve Gallaire and Jack Minker (ed.), Plenum Press, New York, 1978, 293-322.

[ClG81] K. L. Clark and S. Gregory, A Relational Language for Parallel Programming, *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, October 1981, 171-178.

[CMG82] K. L. Clark, F. G. McCabe and S. Gregory, IC-PROLOG Language Features, in *Logic Programming*, K. L. Clark and S.-A. Tarnlund (ed.), 1982, 253-266.

[ClT82] K. L. Clark and S.-A. Tarnlund, *editors, Logic Programming*, Academic Press, 1982.

[ClG84] K. Clark and S. Gregory, Notes on Systems Programming in PARLOG, *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, 299-306.

[ClM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[CoK81] J. S. Conery and D. F. Kibler, Parallel Interpretation of Logic Programs, *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, October 1981, 163-170.

[Con83] J. S. Conery, The AND/OR Process Model for Parallel Interpretation of Logic Programs, Technical Report 204, Ph.D. Thesis, University of California at Irvine, June 1983.

[Cox84] P. T. Cox, Finding Backtrack Points for Intelligent Backtracking, in *Implementations of Prolog*, J. A. Campbell (ed.), Ellis Horwood Limited, 1984, 216-233.

[DaR81] J. Darlington and M. Reeve, Alice: A Multi-processor Reduction Machine for the Parallel Evaluation of Applicative Languages, *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, 1981, 65-75.

[DeG84] D. DeGroot, Restricted And-Parallelism, *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, 471-478.

[Fer81] R. J. Ferguson, Prolog Interpreter for the Unix Operating System, Master Thesis, Department of Computer Science, University of Waterloo, 1981.

[GTM84] A. Goto, H. Tanaka and T. Moto-oka, Highly Parallel Inference Engine PIE - Goal Rewriting Model and Machine Architecture, *New Generation Computing 2*, 1 (1984), 37-58.

[Gre85] S. Gregory, *Design, Application and Implementation of a Parallel Logic Programming Language*, Ph.D. Thesis, Imperial College of Science and Technology, University of London, September 1985.

[Hal86] Z. Halim, A Data-Driven Machine for OR-Parallel Evaluation of Logic Programs, *New Generation Computing 4*, 1 (1986), 5-33.

[ISK85] N. Ito, H. Shimizu, M. Kishi, E. Kuno and K. Rokusawa, Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog, *New Generation Computing 3*, 1 (1985), 15-41.

[KKM83] S. Kasif, M. Kohli and J. Minker, PRISM: A Parallel Inference System for Problem Solving, *Proc. of Logic Programming Workshop 83*, Portugal, 1983, 123-152.

[KKI85] M. Kishi, E. Kuno, N. Ito and K. Rokusawa, The Dataflow-Based Parallel Inference Machine to Support Two Basic Languages in KL1, Technical Report TR-114, ICOT, Tokyo, Japan, 1985.

[KoK71] R. A. Kowalski and D. Kuehner, Linear Resolution with Selection Function, *Artificial Intelligence 2*, (1971), 227-260.

[Kow74] R. A. Kowalski, Predicate Logic as Programming Language, *Proceedings of IFIP Congress 74*, Stockholm, Sweden, August 1974, 569-574.

[Kow79a] R. A. Kowalski, *Logic for Problem Solving*, Elsevier, North-Holland, 1979.

[Kow79b] R. A. Kowalski, Algorithm = Logic + Control, *Comm. ACM 22*, 7 (July 1979), 424-436.

[KuL86] V. Kumar and Y. J. Lin, *A Framework for Intelligent Backtracking in Logic Programs*, Computer Science Department, University of Texas at Austin, 1986.

[Lee84] K. S. Lee, Concurrent Prolog in a Multi-process Environment, CS-84-46, Department of Computer Science, University of Waterloo, November 1984.

[Lev84] J. Levy, A Unification Algorithm for Concurrent Prolog, *Proc. of the Second International Logic Programming Conference*, Uppsala, July 1984, 333-341.

[Lev86] J. Levy, Shared Memory Execution of Committed-Choice Languages, CS86-29, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, October 1986.

[LeF86] J. Levy and N. Friedmann, Concurrent Prolog Implementations - Two New Schemes, CS86-13, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, May 1986.

[Llo84] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984.

[Mai83] D. Maier, *The Theory of Relational Databases*, Computer Science Press, 1983.

[Mel82] C. S. Mellish, An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter, in *Logic Programming*, K. L. Clark and S.-A. Tarnlund (ed.), 1982, 99-106.

[Mie84] C. Mierowsky, Design and Implementation of Flat Concurrent Prolog, CS84-21, The Weizmann Institute of Science, Rehovot, Israel, 1984.

[MTA84] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata and T. Maruyama, The Architecture of a Parallel Inference Engine - PIE, *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, 479-488.

[MoK85] T. Moto-oka and M. Kitsuregawa, *The Fifth Generation Computer: the Japanese Challenge*, John Wiley & Sons, 1985.

[Nil80] N. J. Nilsson, *Principles of Artificial Intelligence*, SRI International, 1980.

[OAS85] R. Onai, M. Aso, H. Shimizu, K. Masuda and A. Matsumoto, Architecture of a Reduction-Based Parallel Inference Machine: PIM-R, *New Generation Computing 3*, 2 (1985), 197-228.

[Pol81] G. H. Pollard, *Parallel Execution of Horn Clause Programs*, Ph.D. Thesis, University of London, Imperial College of Science & Technology, 1981.

[Rob65] J. A. Robinson, A Machine Oriented Logic Based on the Resolution Principle, *J. ACM 12*, 1 (1965), 23-44.

[ShT83] E. Y. Shapiro and A. Takeuchi, Object Oriented Programming in Concurrent Prolog, *New Generation Computing 1*, (1983), 25-48.

[Sha83] E. Y. Shapiro, A Subset of Concurrent Prolog and its Interpreter, CS83-06, The Weizmann Institute of Science, Rehovot, Israel, February 1983.

[TaK84] N. Tamura and Y. Kaneda, Implementing Parallel Prolog on a Multiprocessor Machine, *Proc. of IEEE International Symposium on Logic Programming*, Atlantic City, NJ, U.S.A., 1984, 42-48.

[TLM84] S. Taylor, A. Lowry, G. Q. Maguire Jr. and S. J. Stolfo, Logic Programming Using Parallel Associative Operations, *Proc. of IEEE International Symposium on Logic Programming*, Atlantic City, NJ, U.S.A., 1984, 58-68.

[TBH82] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, Data-Driven and Demand-Driven Computer Architecture, *ACM Computing Surveys 14*, 1 (March 1982), 93-143.

[Ued85] K. Ueda, Guarded Horn Clauses, Technical Report TR-103, ICOT, Tokyo, Japan, 1985.

[UmT83] S. Umeyama and K. Tamura, ~~A Parallel~~ Execution Model of Logic Programs, *Proc. of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, 349-355.

[vaK76] M. H. van Emden and R. A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *J. ACM 23*, 4 (October 1976), 733-742.

[vad79] M. H. van Emden and G. J. de Lucena, Predicate Logic as a Language for Parallel Programming, CS-79-15, University of Waterloo, April 1979.

[van81] M. H. van Emden, An Interpreting Algorithm for Prolog Programs, CS-81-28, University of Waterloo, September 1981.

[War77] D. H. D. Warren, Implementing Prolog: Compiling Predicate Logic Programs, D.A.I. Research Report No. 39/40, University of Edinburgh, Edinburgh, Scotland, 1977.

[War80] D. H. D. Warren, Logic Programming and Compiler Writing, *Software-Practice and Experience 10*, (1980), 97-125.

[WAD84] D. S. Warren, M. Ahamad, S. K. Debray and L. V. Kale, Executing Distributed Prolog Programs on a Broadcast Network, *Proc. of IEEE International Symposium on Logic Programming*, Atlantic City, NJ, U.S.A., 1984, 12-21.

[Wis82] M. J. Wise, A Parallel Prolog: the Construction of a Data Driven Model, *1982 ACM Symposium on Lisp and Functional Programming*, August 1982, 56-66.

Appendix A1

Example Programs

In this appendix, example programs in Concurrent Prolog, PARLOG and GHC are provided with the intention to give the reader some ideas of the programming techniques supported by these languages. The programs are very simple and their executions incur no guard hierarchies; more programming examples can be found in the references quoted in Section 2.2.

The first example is a stack process written in Concurrent Prolog as shown in Program 1:

```
[C0] stack(S) <- stack(S?, []).
[C1] stack([pop(X)|S], [X|Xs]) <- stack(S?, Xs).
[C2] stack([push(X)|S], Xs) <- stack(S?, [X|Xs]).
[C3] stack([], []).
```

Program 1: A Stack Process Written in Concurrent Prolog

A stack process (implemented by the clauses C1 to C3) has two arguments: the first is an input stream of stack operations represented as a list of commands, the second is the stack represented as a list of elements. The clause C0 is used as an interface to the caller; its purpose is to initialize the stack whose internal representation (i.e. a list) is hidden from the caller.

A stack process is invoked by a goal *stack(S?)*, where S is the input stream of stack operations. The goal unifies with the clause head of C0, invoking the stack process with an empty stack. The invoked stack process then iterates, processing the commands in the input stream. If no command is available (i.e., S is uninstantiated), the process suspends, due to the read-only annotation S? in the recursive calls to stack. Otherwise, one of the following three cases must apply:

(1) If the next command in the input stream is pop(X) and the stack is nonempty, the clause C1 is invoked. It unifies X with the top of the stack, and iterates with the rest of the input stream and the rest of the stack.

(2) If the next command in the input stream is push(X), the clause C2 is invoked. It adds X to the top of the stack, and iterates with the rest of the input stream and the new stack.

(3) If both the input stream and the stack are empty, the clause C3 is invoked to terminate the stack process.

If none of these cases applies (e.g. when the next command is pop(X) but the stack is empty), the stack process fails. Stream parallelism can be exploited if the input stream S is incrementally constructed by the caller, in that case the execution of the stack process is synchronized by the availability of data in S. In Concurrent Prolog, the absence of a guard from a clause means that the guard is true. Hence when the clauses C1 to C3 are tried in parallel, the first one with successful head unification will become the committed clause.

The second example is a simple Unix-like shell written in PARLOG. The shell, as shown in Program 2, handles a stream of commands to run foreground and background processes without input and output:

```
mode shell(?).

[C1] shell([]).
[C2] shell([bg(Proc)|Cmds]) <- call(Proc) , shell(Cmds).
[C3] shell([fg(Proc)|Cmds]) <- call(Proc) & shell(Cmds).
```

Program 2: A Simple Shell Written in PARLOG

The procedure shell is invoked by a goal *shell(Cmdlist)*, where Cmdlist is the input stream represented as a list of commands. Each command should be either bg(Proc) or fg(Proc), which denotes a background process or a foreground process, respectively. If no command is available (i.e. Cmdlist is uninstantiated), the clauses C1 to C3 become suspended since they violate the input constraint of the input argument, as indicated by the mode declaration shell(?). As a result, the goal will be suspended since all its applicable clauses are suspended. Otherwise, one of the following three cases must apply:

(1) If Cmdlist is empty, the clause C1 is used to terminate the shell.

(2) If the next command is bg(Proc), the clause C2 deals with this background command by concurrently (because of the parallel AND operator ",") invoking the process Proc via the metalevel facility call, and resuming the shell to process the next command.

(3) If the next command is fg(Proc), the clause C3 deals with this foreground command by invoking the process Proc via the metalevel facility call, and waiting for it to terminate successfully (because of the sequential AND operator "&") before resuming the shell to process the next command.

The goal fails if none of these cases applies. Stream parallelism can be exploited if Cmdlist is incrementally constructed by the caller. As in Concurrent Prolog, the absence of a guard from a clause means that the guard is true.

The example program for GHC is a binary merge process as shown in Program 3:

```
[C1] merge([A|Xs], Ys, Zs) <- true | Zs=[A|Zs1], merge(Xs, Ys, Zs1).
[C2] merge(Xs, [A|Ys], Zs) <- true | Zs=[A|Zs1], merge(Xs, Ys, Zs1).
[C3] merge([], Ys, Zs) <- true | Zs=Ys.
[C4] merge(Xs, [], Zs) <- true | Zs=Xs.
```

Program 3: A Binary Merge Process Written in GHC

The procedure merge is invoked by a goal *merge(Is1, Is2, Os)* which merges the two input streams Is1 and Is2 into the output stream Os. All the streams are implemented as lists of elements. If no data is available in both input streams (i.e., both Is1 and Is2 are uninstantiated), the clauses C1 to C4 become suspended since they violate the rule of suspension. As a result, the goal will be suspended since all its applicable clauses are suspended. Otherwise, one of the following four cases must apply:

(1) If data in the first input stream is available, the clause C1 is invoked to add the data to the output stream, and iterates with the rest of the first input stream.

(2) If data in the second input stream is available, the clause C2 is invoked to add the data to the output stream, and iterates with the rest of the second input stream.

(3) If the first input stream is empty, the clause C3 is invoked to connect the second input stream to the output stream.

(4) If the second input stream is empty, the clause C4 is invoked to connect the first input stream to the output stream.

If data are available in both input streams, which of C1 and C2 is selected first depends on the particular implementation. In a good parallel implementation, the elements from the two input streams are expected to appear in the output stream almost in the order of arrival. Note that in GHC, a true guard must be explicitly stated. The binding of Zs to [A|Zs1] in C1 and C2 must be done explicitly in the clause body, instead of implicitly in the clause head, to avoid violating the rule of suspension.

Appendix A2
Levy's Execution Model

The proposed execution model described in Chapter 4 of this thesis is a modified version of Levy's execution model, which was designed to support the parallel execution of Concurrent Prolog, PARLOG and GHC on shared memory multiprocessor systems. Many of the design ideas in the proposed execution model are inherited from Levy's work; these include the viewing of the computation necessary to satisfy a goal with its applicable clauses as a unit, the encoding of the computations by a finite state machine, the heavy usage of locking operations and the concept of recursive commitment. As these features have already been covered in Chapter 4, to avoid repetition, this appendix will only give a brief description of Levy's execution model and highlight its major differences with the proposed execution model.

In Levy's execution model, the AND/OR tree of a logic program is partitioned into computation units each of which includes the computation necessary to satisfy a goal with all its applicable clauses. A computation unit is represented by a goal record and several guard tuples: the goal record represents the goal to be solved, the guard tuples represent the computations of individual applicable clauses (after their guards are spawned). Computation units are placed in the ready goal pool and are commonly accessed by a set of processes, each of which executes a copy of a finite-state machine that repeatedly selects a goal from the pool, advances its state, and places the generated goals (if any) back into the pool.

A goal record contains all the information necessary to satisfy a goal. It, as described in Levy's model, has the following fields: Procedure, Arguments, Guard_Tuple, Status_Vector, Env_Vector, Ready and Link. The first five fields are carried over to the proposed execution model; their meanings are described in Section 4.3.1. The Ready field is set to NULL when the goal record is not in the ready goal pool. It is set to some non-NULL value if it is currently in the ready goal pool. The Link field is the same as the R_Link field in the proposed execution model; it is used to manage the pool of ready goals.

Since Concurrent Prolog, PARLOG and GHC are committed choice languages, a goal will commit to one of its applicable clauses when the guard of that clause is solved, and the computations involving the other applicable clauses are useless after the commitment. Also, a guard is a conjunction; it fails when one of its goals fails. To avoid unnecessary computations, information of failure and abortion (due to commitment) should be propagated promptly: failure should be propagated across the conjunction of goals in a guard and up the AND-OR tree; abortion should be propagated across the disjunction of applicable clauses of a goal and down the AND-OR tree. In Levy's execution model, the goal records are not physically interconnected as a tree, so only the upward propagation of failure, by checking the Status_Vector field of a goal record, is supported. It is possible that Levy's model will waste its resources in the computations involving a subtree rooted by an aborted/failed goal. To avoid this situation, the proposed execution model adds two fields to the goal record which, together with the Father and Goal_List fields of the guard tuple (see below), reveal the AND-OR tree structure among the goal records along which failure and abortion can be propagated. These two fields are GT_Vector and G_Link, their meanings are described in Section 4.3.1.

Another modification in the proposed execution model is to replace the Ready field in the goal record by the Flag field. The Flag field is set to READY if the goal record is currently in the ready goal pool. It is set to DEQUEUED if it is removed from the ready goal pool and is in some execution status. It is set to DEAD if it has failed, been reduced or aborted. This further classification of goal status is necessary since the computations involving a goal in the ready goal pool may become useless, say, because of the propagation of failure and abortion to that goal. The classification of goal records into READY and non-READY, as in Levy's execution model, is not adequate to reflect this fact.

A guard tuple is used to monitor the computation of a guard. It, as described in Levy's model, has three fields: Goal_Record, Status_Word, and Guard_Count. The Goal_Record field contains a reference to the goal for which this guard is computing, it is the same as the Father field in the proposed model. The Status_Word field points to the location of the status vector of the goal corresponding to the clause containing this guard. In the proposed model, it is replaced by a pointer to the clause so that the committed clause can be located in recursive commitment. The Guard_Count field records the number of currently unsolved goals in this guard; it is used in recursive commitment (see Section 4.4.2.5). A fourth field, Goal_List, is added to the guard tuple in the proposed model; it is used to manage the goals in this guard (see Section 4.3.3).

In Levy's model, a goal reduction step is divided into seven stages: UNIFY, UNIFYING, SPAWN, SPAWNED, COMMIT, COMMITTED and FAIL. This staged goal reduction concept is carried over to the proposed model, the meanings of the stages are described in Section 4.3.2. Because of the introduction of failure/abortion propagation in the proposed model, a new stage, KILLED, is added. When a goal commits to a clause, the status words of other applicable clauses of that goal are set to KILLED (in Levy's model the status words are set to FAIL); this is necessary for the failure/abortion propagation scheme to work properly (see Section 4.4.2.6). Finally, as the guard spawning operation involves the creation of a guard tuple and goal records, it may take a relatively long time (when compared with the other operations). A new stage, SPAWNING, is added in the proposed model so that the completion of the spawning operation can be reflected by a stage changed: from spawning to spawned.

Appendix A3
Run-Time Structures in the Proposed Execution Model

This appendix demonstrates the creation and interconnection of the run-time structures in the proposed execution model. The following Concurrent Prolog program is used as an example:

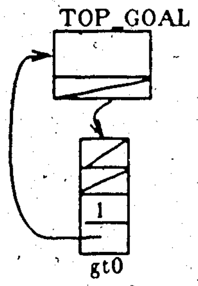
```
<- p(X),q(X?),r(Y).

[C1] p(1).
[C2] q(Y) <- u(Y) | t(Y).
[C3] q(Z) <- s(Z) | u(Z).
[C4] r(Z) <- s(Z) | .
[C5] s(1).
[C6] t(1).
[C7] u(1).
```

For the sake of simplicity, only the *Guard_Tuple* and *GT_Vector* fields of a goal record are shown with the latter placed inside the box representing the goal record, and the former along the boundary. During program execution, snapshots of the run-time structures will be displayed.

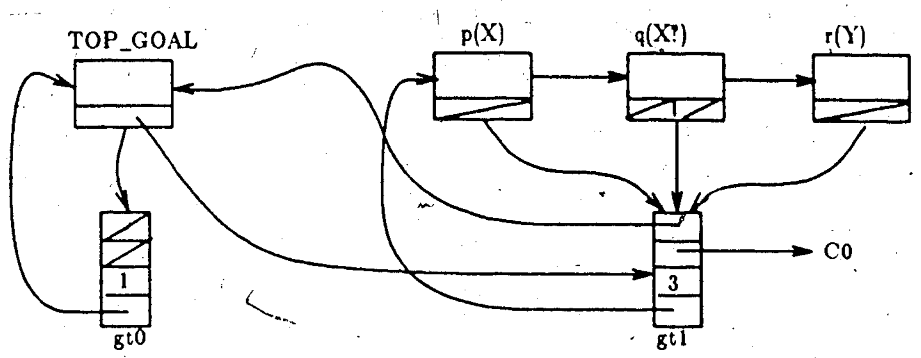
As explained in Section 4.4.1, before program execution, a top level goal TOP_GOAL is created whose only applicable clause C0 has a guard containing the goals in the original goal clause and an empty body:

```
<- TOP_GOAL.                         [new goal clause]
TOP_GOAL <- p(X),q(X?),r(Y) | .      [C0]
```

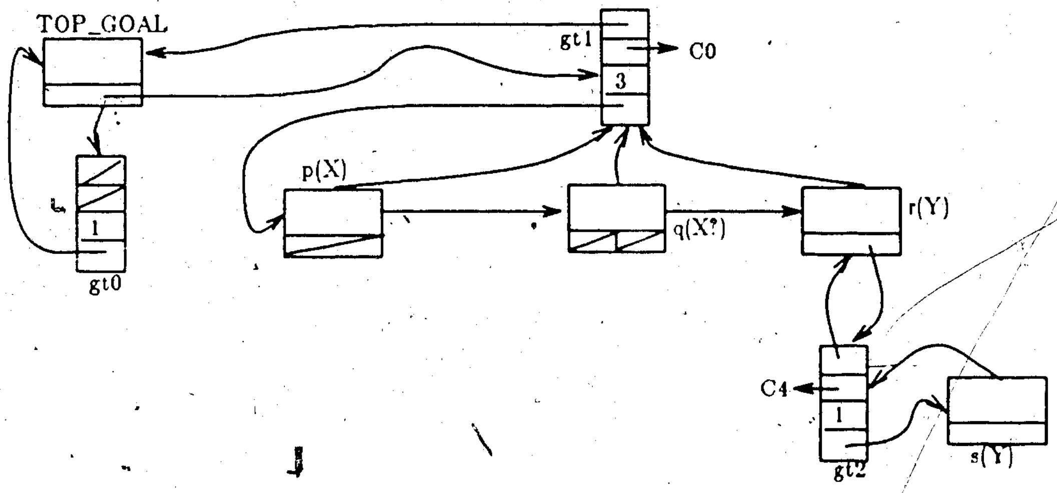
The system creates a goal record for TOP_GOAL. Since TOP_GOAL is the root of the AND/OR tree and hence not a part of any guard, a dummy guard tuple gt0 is created with NULL father and clause fields (used for program termination detection).



The system adds TOP_GOAL's goal record into the ready goal pool. Some time later it will be dequeued and processed. The clause C0 is applied and the head unification succeeds, the guard is spawned with a guard tuple gt1 created to monitor it. The goal records for p(X), q(X?) and r(Y) are created and added into the ready goal pool. The ready goal pool is now: {p(X), q(X?), r(Y)}.

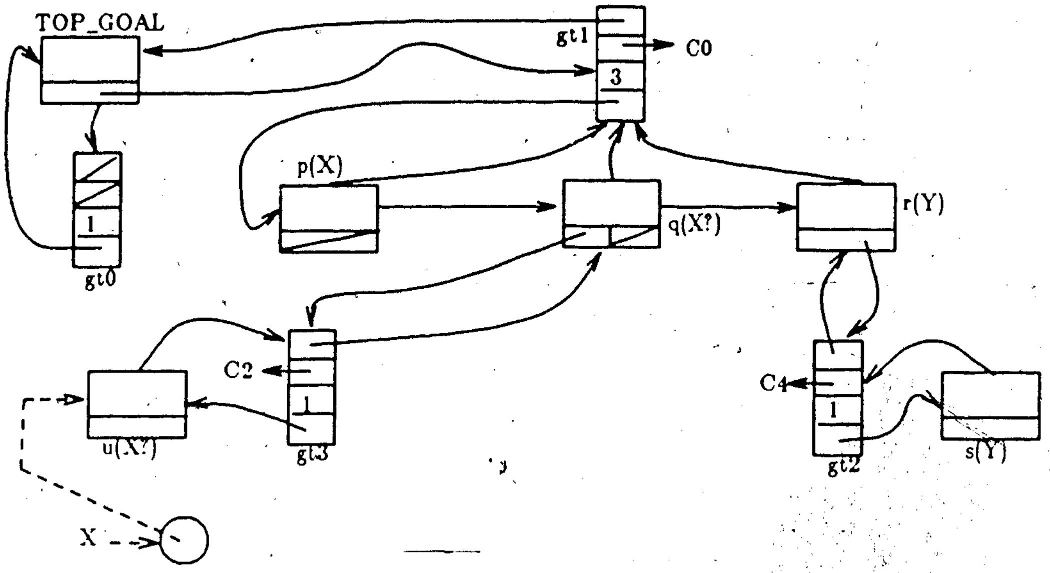


Suppose the goal record of r(Y) is dequeued and processed first. The head unification of r(Y) and C4 succeeds, binding Z to Y. The computation proceeds to spawn the guard in C4: a guard tuple gt2, and a goal record for s(Y) are created with the latter being added into the ready goal pool. The ready goal pool is now: {p(X), q(X?), s(Y)}.



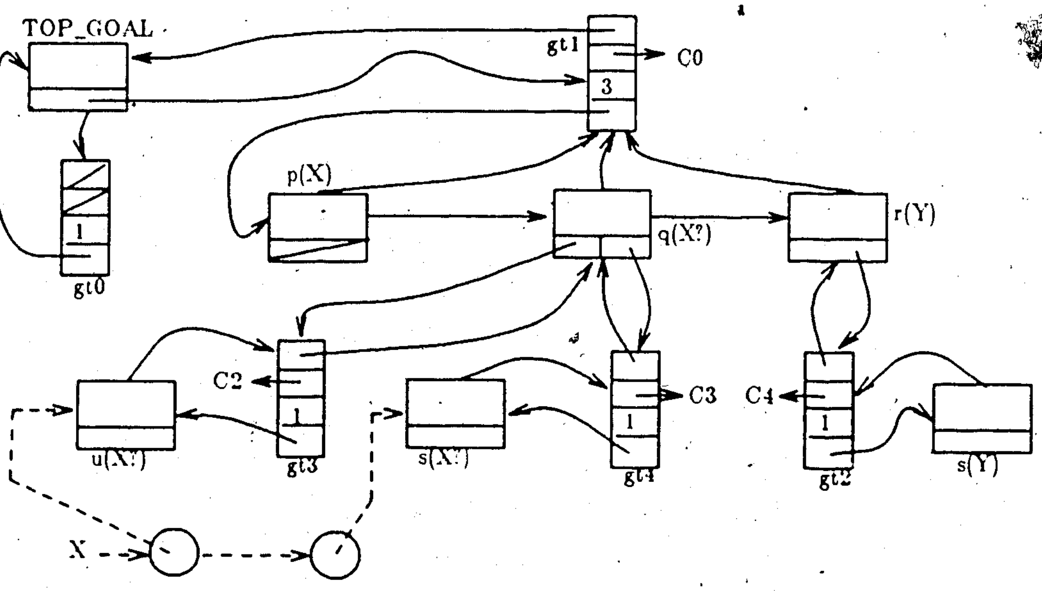
Suppose at this time the goal record of q(X?), which has two applicable clauses, is dequeued and processed. As the system tries the clauses *in sequence*, C2 is applied first. The head unification of q(X?) and C2 succeeds, binding Y to X?. The computation proceeds to spawn the guard in C2: a guard tuple gt3, and a goal record for u(X?) are created with the latter being added into the ready goal pool. The ready goal pool is now: {p(X), s(Y), u(X?)}.

Suppose the goal record of u(X?) is immediately dequeued by a process. The head unification of u(X?) and C7 requires the instantiation of the read-only variable X?, hence it is suspended. A suspension record (represented by a circle) referencing the goal record of u(X?) is created and added to the suspension list of variable X. The ready goal pool is now: {p(X), s(Y)}.

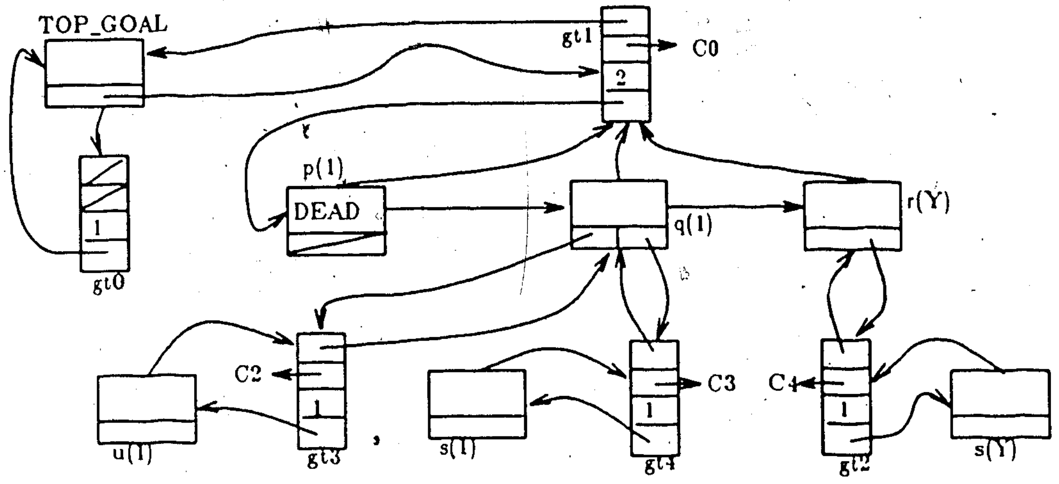


Meanwhile, the process evaluating q(X?), after having spawned C2, tries its second clause C3. The head unification of q(X?) and C3 succeeds, binding Z to X?. The guard is then spawned: a guard tuple gt4 and a goal record for s(X?) are created with the latter being added into the ready goal pool. The ready goal pool is now: {p(X), s(Y), s(X?)}.

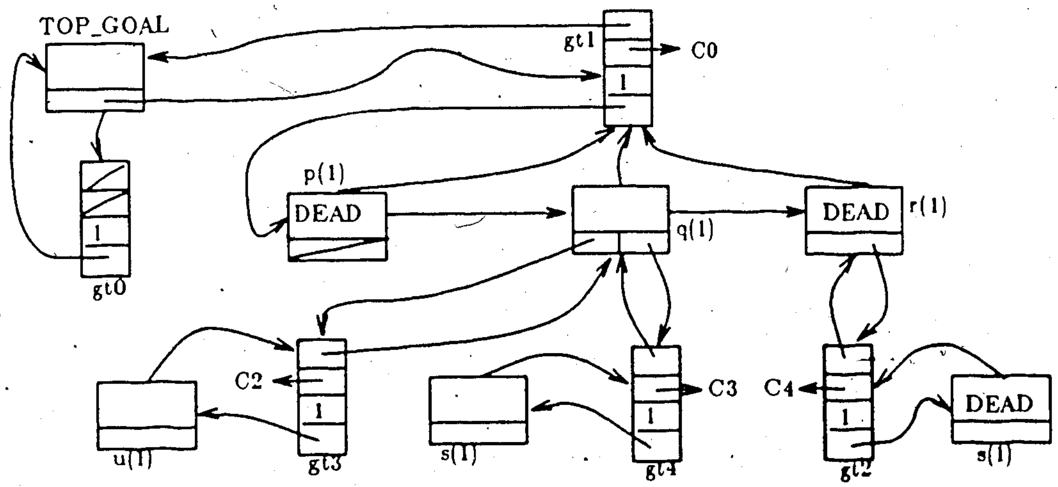
Suppose the goal record of s(X?) is immediately dequeued by a process. The head unification of s(X?) and C5 requires the instantiation of the read-only variable X?, hence it is also suspended. A suspension record is created for it and added into X's suspension list. The ready goal pool is now: {p(X), s(Y)}.



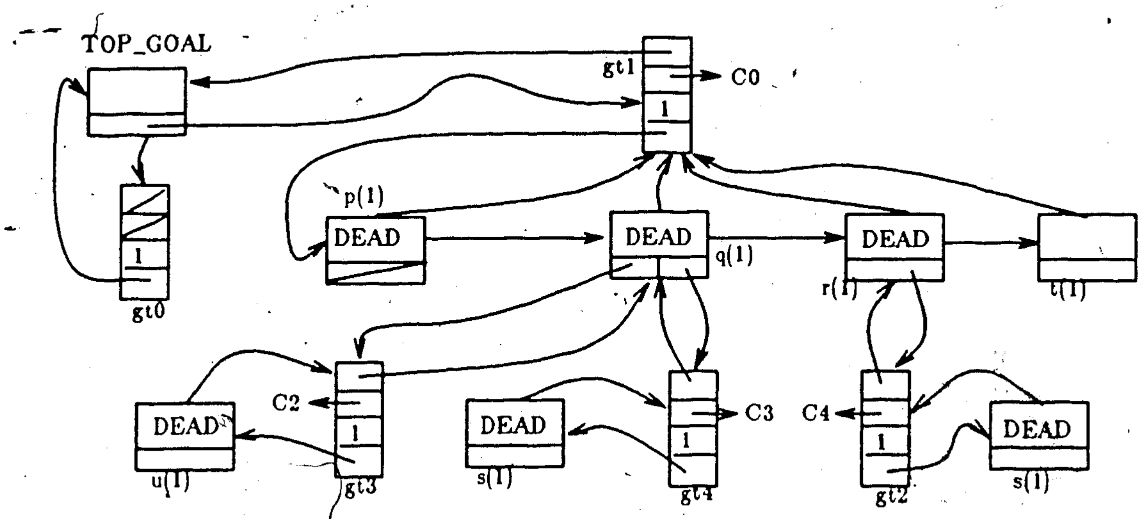
At this time the goal record of p(X) is dequeued and processed. The head unification of p(X) and C1 succeeds, binding X to 1. C1 has an empty guard, hence the computation proceeds immediately to COMMIT without creating any guard tuple. After the commitment action succeeds, the suspended goals u(1) and s(1) (note that X is now bound to 1) are added back into the ready goal pool. P(1) is reduced (marked as DEAD) to C1's empty body, creating no new goals. The guard count in gt1 is decremented. The ready goal pool is now: {s(Y), u(1), s(1)}.



Suppose now the goal record of s(Y) is dequeued and processed. The head unification of s(Y) and C5 succeeds, binding Y to 1. Since C5 has an empty guard, s(1) commits immediately to C5 and is later solved (marked as DEAD). Also, the guard count in gt2 is 1, so a *recursive commitment* occurs: r(1) (i.e., s(1)'s father) is reactivated and commits to C4. As C4 has an empty body, r(1) is solved (marked as DEAD). The guard count of gt1 is decremented. The ready goal pool is now: {u(1), s(1)}.



Suppose now the goal record of u(1) is dequeued and processed. Eventually u(1) commits to C7 and is later solved (marked as DEAD). As the guard count in gt3 is 1, q(1) is re-activated and commits to C2 (recursive commitment). C2's sibling clause C3 is KILLED, the goal s(1) in its spawned guard is also aborted (marked as DEAD). q(1) is reduced (and henced is marked as DEAD) to the committed clause body t(1). The goal record for t(1) is created and added into the ready goal pool and gt1's Goal_List. The ready goal pool is now: {s(1), t(1)}.



Suppose now the goal record of s(1) is dequeued. As its flag is not READY (since it is aborted after being inserted into the ready goal pool), it is discarded immediately. Finally the goal record of t(1) is dequeued and processed. t(1) commits to C6 and is later solved. As the guard count in gt1 is 1, TOP_GOAL is reactivated and commits to C0 (recursive commitment). As C0 has an empty body and the guard count in gt0 is 1, TOP_GOAL's father is referenced. Since it is NULL, the system knows that it is not a recursive commitment, but a *program termination*. The answers X=1, Y=1 are then reported.