

High Performance Execution of Prolog Programs
Based on a Static Data Dependency Analysis

By

Jung-Herng Chang

B.S. (National Taiwan University) 1977

M.S. (University of California) 1980

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

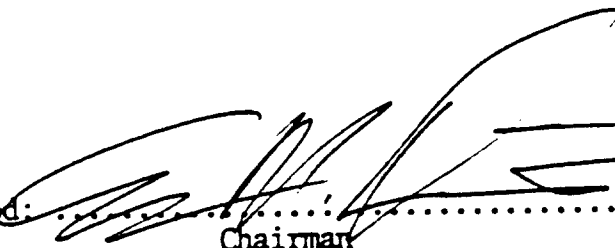
Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: 
Chairman Date September 10, 1985
.....
Boris Rubinfeld September 11, 1985
.....
Yuh N. Pan Sept 11, 1985
.....

.....

ACKNOWLEDGEMENTS

I would like to thank my research advisor, Professor Despain, for his advice, patience, and encouragement during this research. I would also like to thank my wife, Whei-Ling, for her love, support, and encouragement. Without these, I could not have pulled myself through all the ups and downs of these years.

There are several others who have helped me with this dissertation. Professor Yale Patt and Professor Boris Rubinsky of my thesis committee provided many good suggestions. Dr. Doug DeGroot of IBM research provided me a pleasant working environment in the summer of 1984 when the ground work for my thesis began. Tep Dobry helped me in modifying his PLM simulator and understanding the PLM machine. Barry Fagin, Wayne Citrin, Carl Ponder, and Dr. Vason Srinii all gave me constructive comments on my earlier work. I appreciate all their help.

Part of this research was sponsored by Defense Advance Research Projects Agency (DoD) Arpa Order No. 4871 Monitored by Naval Electronic Systems under Contract No. N00039-84-C-0089.

HIGH PERFORMANCE EXECUTION OF PROLOG PROGRAMS
BASED ON
A STATIC DATA DEPENDENCY ANALYSIS

Jung-Herng Chang

Ph.D.

CS Division (EECS)

ABSTRACT

Prolog programs are executed from left-to-right and top-to-bottom with backtracking to the most recently activated choice-point when a failure occurs. This execution strategy is based on a sequential execution model and has been implemented with modest efficiency in conventional computer systems [12,16]. In this thesis, two ways are explored to improve the performance of a Prolog system. The first way is a more intelligent form of backtracking. The second way is to exploit AND-parallel execution.

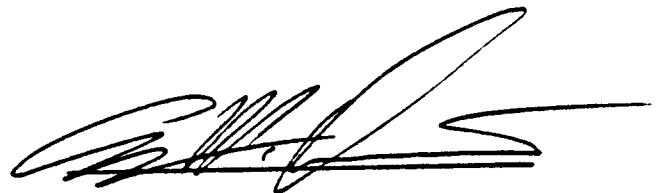
Both intelligent backtracking and AND-parallel execution require information about the dependency between body literals. This information can be derived either at compile-time by using a static analysis or at run-time. Although a run-time analysis is more effective than a static (hence worst-case) analysis, it incurs a lot of overhead at run-time and is thus inefficient. Therefore, this thesis has emphasized the use of compile-time analysis to improve run-time performance.

A methodology for a Static Data Dependency Analysis (SDDA) was developed. The SDDA is based on a worst-case analysis of variable bindings. To perform the SDDA, only one declaration, which describes the worst case activation, is necessary for each procedure which can be directly invoked from the top level query. This extra work can be handled quite easily by the programmer. The cost of doing the SDDA is shown to be comparable to the cost of

compilation of a Prolog program. The outputs from the SDDA are a collection of data dependency graphs, one for each clause in a Prolog program. From data dependency graphs, both intelligent backtracking and AND-parallel execution can be determined.

A scheme for compiling intelligent backtracking based on the SDDA has been designed. To take full advantage of dependency graphs, three different types of backtracking are differentiated. At run-time, when a subgoal fails, a backtrack literal can be determined by the type of the backtracking and its corresponding backtracking path. Execution including this intelligent backtracking is simulated for a sequential Prolog machine. It includes modifications of the hardware and the compiler. This scheme has been proved to be very effective for improving the execution of Prolog programs.

A scheme to exploit AND-parallelism is also proposed. It includes generating parallel executable tasks by the SDDA, using a set of message protocols to coordinate co-operating processes, exploiting both intelligent backtracking and parallel backtracking. It is shown that Prolog has potential in parallel processing because of its procedural invocation, non-deterministic execution, concise syntax, single-assignment variable bindings, and local variable scoping.



Alvin M. Despain

Chairman of Committee

TABLE OF CONTENTS

| | |
|---|----|
| Chapter 1. Introduction | 1 |
| 1.1. What Is Prolog | 1 |
| 1.1.1. Declarative Semantics of Prolog | 1 |
| 1.1.2. Procedural Semantics of Prolog | 3 |
| 1.2. Motivation | 4 |
| 1.3. Overview | 4 |
| 1.4. Efficient Execution | 5 |
| 1.5. Parallel Processing of Prolog | 5 |
| 1.6. A Static Data Dependency Analysis for Prolog Programs | 7 |
| 1.7. The Thesis | 7 |
| 1.8. Contributions | 7 |
| Chapter 2. A Static Data Dependency Analysis of Prolog | 9 |
| 2.1. Overview | 9 |
| 2.2. A Static Data Dependency Analysis (SDDA) for Prolog Programs | 9 |
| 2.2.1. Generating A Data Dependency Graph | 9 |
| 2.2.2. Special Characteristics of Prolog | 11 |
| 2.2.3. Activation Mode Declaration | 14 |
| 2.2.4. A Methodology | 14 |
| 2.2.5. Optimizations | 24 |
| 2.2.5.1. Structure Matching | 24 |

| | |
|--|----|
| 2.2.5.2. Refined Activation Mode Declaration | 25 |
| 2.2.5.3. Separating Graph Generation from Exit Mode Derivation | 25 |
| 2.2.6. Handling Control Predicates, I/O, and Global Effect Predicates | 26 |
| 2.3. A Hidden Problem | 26 |
| 2.4. Complexity Analysis of the SDDA | 26 |
| 2.5. Conclusion | 28 |
| Chapter 3. Compiling Intelligent Backtracking for A Prolog Machine | 29 |
| 3.1. Overview | 29 |
| 3.2. Backtracking | 29 |
| 3.2.1. Run-Time Intelligent Backtracking | 30 |
| 3.2.2. Semi-Intelligent Backtracking | 32 |
| 3.3. Compiling Intelligent Backtracking | 34 |
| 3.3.1. Determining Intelligent Backtrack Paths | 34 |
| 3.3.1.1. Type I Backtrack Path | 34 |
| 3.3.1.2. Type II Backtrack Path | 36 |
| 3.3.1.3. Type III Backtrack Path | 39 |
| 3.3.2. Architectural Support and Code Generation for Intelligent Backtracking | 40 |
| 3.3.2.1. The Berkeley Prolog Machine (PLM) | 40 |
| 3.3.2.2. Implementing Intelligent Backtracking in PLM | 43 |
| 3.3.2.3. Code Generation | 47 |
| 3.3.2.4. Simulations | 50 |
| 3.3.3. Applications | 50 |

| | |
|--|-----|
| 3.3.4. Backtrack Graph vs. Data Dependency Graph | 59 |
| 3.4. Comparison with Other Implementations/Approaches of Intelligent Backtracking | 61 |
| 3.4.1. Compiling Intelligent Backtracking vs. Run-Time Intelligent Backtracking | 61 |
| 3.4.2. Intelligent Backtracking Based on Annotated Programs | 63 |
| 3.5. Conclusion | 63 |
| Chapter 4. Compiling AND-Parallelism for A Parallel Architecture | 64 |
| 4.1. Overview | 64 |
| 4.2. Why Exploit AND-Parallelism ? | 64 |
| 4.2.1. An Example | 64 |
| 4.3. An AND-Parallel Execution Environment | 66 |
| 4.4. Memory Management | 68 |
| 4.5. Execution Flow Control in AND-Parallel Execution | 70 |
| 4.6. Semi-Intelligent Backtracking in the AND-Parallel Execution Environment | 73 |
| 4.6.1. Parallel Backtracking in AND-Parallel Execution Environment | 77 |
| 4.7. Performance Improvement of The AND-Parallel Execution - An Example | 77 |
| 4.8. Conclusion | 79 |
| Chapter 5. Conclusion | 80 |
| 5.1. Conclusions | 80 |
| Bibliography | 84 |
| Appendix A | A.1 |

Appendix B B.16

CHAPTER 1

INTRODUCTION

1.1. What is Prolog

Prolog [1], designed by Colmerauer and his colleagues around 1972, is a special implementation of a logic programming language. A logic programming language is, in short, the Horn clause logic [2] (a subset of first order logic) plus backward reasoning. Its inference rule is the resolution principle developed by Robinson [3] in 1965. In general, a logic program consists of two components. One is the logic component which describes the knowledge used in solving the problem. In Prolog, this is supplied by the programmer. The other is the control component which determines the problem-solving strategy in order to achieve efficiency [4]. The Prolog execution provides a default strategy, but this can be modified by the programmer if desired. The logic component and the control component of a Prolog program will be discussed separately by considering the declarative semantics and procedural semantics of Prolog.

1.1.1. Declarative Semantics of Prolog

Prolog programs can be understood declaratively. The correctness of the program is independent of its problem-solving strategy¹. Three types of statements are allowed in Prolog as shown below. The first two are used to construct Prolog programs. The third is used to interact with existing Prolog programs.

(1) **Fact:**

Facts can be treated as unconditional assertions. Two examples are given below. The first fact describes the 'father' relation between Tom and Mary. The second fact describes that Professor Despain teaches an architecture course, there are two sections with 60 students and 50 students respectively.

```
father(tom,mary).
```

```
class(architecture,instructor(despain),  
      enrollment([section(1,60),section(2,50)])).
```

(2) **Rule:**

Rules have the following form:

¹There are some side effects of Prolog programs that are sensitive to the order of evaluation of expressions. These side effects are ignored in the current discussion but will be discussed in section 2.2.6 & 2.3).

conclusion :- premise₁, ..., premise_n.

It can be understood as saying "The conclusion is true if all the premises are true". An example, which uses three rules to describe the grandfather relation, is shown below. Note that variables begin with an upper case character.

grandfather(Grandfather,Child) :- father(Grandfather,Parent),
parent(Parent,Child).

parent(Parent,Child) :- father(Parent,Child).

parent(Parent,Child) :- mother(Parent,Child).

Each rule is a *clause*. The conclusion is called the *head* of the clause. Premises constitute the *body* of the clause. **Fact** is also called 'unit clause' which is a clause without a body. Both conclusion and premise are expressed by literals. A literal consists of a *predicate* (functor) and a number of arguments. An argument is a *term*. A term can be a constant, a variable, or a function $f(t_1, \dots, t_m)$, where t_1, \dots, t_m are also terms. Clauses which have the same predicate and arity in their head literals constitute a *procedure*. Clauses in a procedure describe disjunctively a relation, e.g. the 'parent' relation above is described by two clauses in the procedure 'parent'. The AND/OR tree of the grandfather relation is shown in Figure 1.1. In the AND/OR tree representation, an AND subtree corresponds to a clause and an OR subtree corresponds to a procedure. Each branch of an OR subtree leads to a candidate clause (an AND subtree) of the procedure. (Parent1 (parent2) denotes the first (second) candidate clause of the parent procedure.) AND/OR tree is useful in illustrating search strategy as will be discussed in Chapter 3.

(3) **Query:**

A query is used to interact with existing Prolog programs. It describes a *goal* that needs to be proved. A goal consists of a list of subgoals each of which is a literal. If there are no variables in a goal, then the result of execution a query is either yes (true) or no (false). Otherwise, instances of variables which make the goal true can be derived as shown in the following examples.

:- father(tom,mary).
Answer: either yes or no.

:- father(tom,Child).
Answer: Child=mary.

(more)

AND/OR Tree:

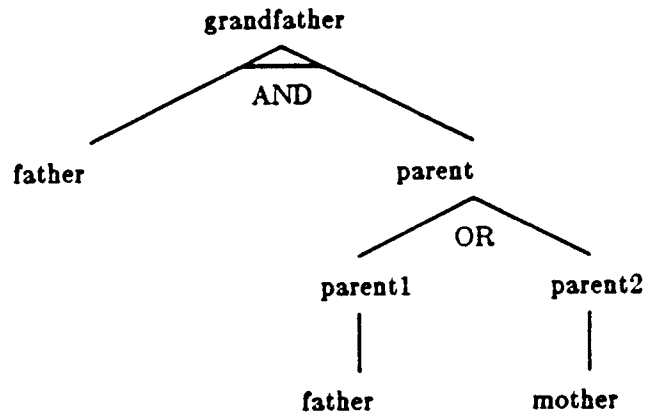


Figure 1.1 The AND/OR tree representation

1.1.2. Procedural Semantics of Prolog

Based on the resolution principle, a deduction step in the logic programming is as follows.

Let $(g_1, g_2, \dots, g_j, \dots, g_m)\Theta_1$ be the current goal list, where Θ_1 denotes the current binding of variables. Θ_1 is a set of variable/term pairs, $\{X_1/t_1, \dots, X_k/t_k\}$, in which X_1, \dots, X_k are distinct variables. We say that X_i is bound to the term t_i . Assume one of the subgoal, g_j , is selected to be executed first, and it can unify² [3]

$$f :- h_1, h_2, \dots, h_n$$

with variable binding Θ_2 , i.e. $(g_j)\Theta_2 = (f)\Theta_2$, then the resolvent is

$$(g_1, g_2, \dots, g_{j-1}, h_1, \dots, h_n, g_{j+1}, \dots, g_m)\Theta$$

$$\text{where } \Theta = \Theta_1 * \Theta_2.$$

²The result of unification between two terms t_1 and t_2 is a set of variable bindings Θ such that $(t_1)\Theta = (t_2)\Theta$. Θ is the most general unifier, that is any other unifying bindings Θ_i is such that $\Theta_i = \Theta * \Theta_i$ for some Θ_i . The composition, $\Theta_1 * \Theta_2$, of two sets of bindings

$$\Theta_1 = \{X_i/t_i, \dots, X_j/t_j\}, \Theta_2 = \{Y_k/s_k, \dots, Y_m/s_m\}$$

is the set of bindings $\Theta_1 \cup \Theta_2$, where $\Theta_1 = \{X_i/(t_i)\Theta_2, \dots, X_j/(t_j)\Theta_2\}$ and Θ_2 is Θ_2 with any bindings for the variables X_1, \dots, X_n deleted.

Logically, subgoals can be selected for execution in any order and clauses in a procedure invoked by a subgoal can be selected in any order. Prolog, however, in execution, adopts a *depth-first search* strategy. Subgoals are examined from left to right and clauses in a procedure are selected from top to bottom. Deduction steps are continued until either the goal is reduced to an empty list (a success), or a subgoal fails. In the latter case, it *backtracks* to the most recently invoked procedure which still has untried clauses. If such a procedure does exist, then the top untried clause of this procedure is selected and deduction continues, else fails. With this fixed problem-solving strategy, the programmer can control search by ordering the body literals in a clause and the clauses in a procedure³. A few control predicates are designed to allow pruning of the search space, e.g. *cut*, and force backtracking, e.g. *fail*. Throughout this thesis, the term 'procedure call' is used to refer to the execution of a subgoal.

1.2. Motivation

Prolog is an important symbolic manipulation language. It has been used successfully for knowledge-based systems [5,6,7], CAD [8,9], natural language processing [10], and compilers [11]. Prolog is comparable with LISP in expressive power as well as implementation efficiency [12]. It features some important mechanisms in symbolic processing: the procedural interpretation of deduction, pattern-directed procedure invocation, an assertional database, and nondeterministic execution (through backtracking). It has potential in parallel processing (discussed later). Its syntactic structure is concise, which makes static concurrency analysis feasible. Because of these features, it has been promoted by the Japanese as the host language for their Fifth Generation Computer System [13], which is a knowledge-based system capable of performing parallel inferences. Recognizing the potential of this language, I have focused my research efforts into ways and means to achieve high performance execution of Prolog programs.

1.3. Overview

Shown in Table 1.1 are benchmark timings [14] for three implementations of Prolog: the Dec-10 Prolog compiler (Prolog-10), the Dec-10 Prolog interpreter (Prolog-10I), and the PLM (abbreviation of Programmable Logic Machine) constructed here at Berkeley [15]. It is shown that compilation (Prolog-10) is about 20 times faster than interpretation (Prolog-10I) (both run on Dec 2060 with a 33ns clock). With special hardware and microcode to interpret compiled abstract code, the PLM (with a 100ns clock) is about 10 times faster than the Prolog-10. The improvement of performance in Prolog is similar to LISP, which starts with interpreters, then compilers, and finally Lisp machines like the Symbolics 3600.

A natural question to ask is "What is the next step to improve the performance of Prolog?". Two ways that can be explored are discussed in this thesis:

- (1) More efficient execution in general, and intelligent backtracking in particular.
- (2) Parallel processing in general, and AND-parallelism in particular.

³Even with the fixed execution strategy, since the Prolog is a universal language, the programmer can still write programs to implement any search strategy he wants. However, the important issue is whether the program's logic component can be as descriptive as before, or how much of the logic component can remain unchanged, when a different search strategy is adopted. Ideally, we would like to have the logic component unchanged even when a different search strategy is adopted [4]. This objective is yet to be achieved.

Table 1.1 Benchmark timings for a few Prolog interpreters/compiler and PLM.

| Benchmark Timings (in ms) | | | |
|---------------------------|---------------------------|-----------------------|----------------|
| Benchmark | Prolog-10I Interpreter | Prolog-10 Compiler | PLM Machine |
| reverse list30 | 1160 | 54 | 4.4 |
| qsort list50 | 1340 | 75 | 4.9 |
| deriv times10 | 76 | 3.0 | 0.38 |
| deriv divide10 | 84 | 2.9 | 0.43 |
| deriv log10 | 49 | 1.9 | 0.20 |
| deriv ops8 | 64 | 2.2 | 0.25 |
| serialize palin25 | 600 | 40 | 3.2 |
| query | 8900 | 185 | 17.3 |

1.4. Efficient Execution

A number of ways have been studied to speed up the sequential execution of Prolog programs. These are clause-indexing [16], goal caching [17], and intelligent backtracking [18,19].

In the scheme of clause indexing [16] the principal functor of the first argument of the calling literal is used as an index to select candidate clauses of the called procedure. It was first implemented in DEC-10 Prolog compiler. It is possible to extend this scheme to index on more than one arguments and/or use a perfect hashing technique [20].

Goal caching [17] is useful for some applications in which remembering part of the history of execution avoids a lot of redundant work. It is similar to using a scratch pad to record goal activations and their results, and retrieving the recorded results if the same activation occurs again. However, it may be feasible only for deterministic subgoals. For non-deterministic subgoals, the cached state must include information about untried clauses of its descendants. This overhead of bookkeeping and cache management is non-trivial.

As mentioned above, Prolog adopts the depth-first search strategy with backtracking to the most recently activated procedure call which still has untried clauses when a failure occurs. However, that procedure call may have nothing to do with the failure. Many users are disappointed when they encounter this naive behavior in Prolog. To backtrack intelligently is to avoid this kind of redundancy as much as possible. Run-time intelligent backtracking has been studied by Cox-Pietrzykowski-Matwin [21] and Bruynooghe-Pereira-Porto [18,19]. Their approaches entail considerable run-time overhead. More detailed critiques about their schemes as well as a new scheme which has much lower overhead are discussed in Chapter 3.

1.5. Parallel Processing of Prolog

Various ways of exploiting parallelism in logic programming have been identified. These are AND-parallelism [22], OR-parallelism [23,24], Stream-parallelism [25], and Unification-parallelism [26]. To exploit the parallelism, some implementations require using extensive annotations to denote parallel executable subgoals, e.g. in Epilog [27], or communication channels between two dependent subgoals, e.g. in Concurrent Prolog [28] and Parlog [29].

In Concurrent Prolog and Parlog, the first candidate clause whose guard literals (similar to the *guarded command* advocated by Dijkstra) are satisfied will be committed and executed with no retry of the other candidate clauses if the committed clause fails. Parallel execution can be exploited by concurrently examining guard literals of candidate clauses. Furthermore, literals in the same clause can be executed concurrently and synchronized through specially annotated communication channels. Although these languages can be used in system programming as well as parallel processing, they are semantically different from conventional logic programming languages and are not discussed further in this thesis.

In general, independent subgoals in the goal list can be executed concurrently by exploiting AND-parallelism. In real implementations, dependency checks⁴ among subgoals are normally done on a clause basis because of the costs associated with these checks. AND-parallelism has been studied by Conery in his thesis [22]. It can be applied to both deterministic and non-deterministic logic programs. However, to exploit AND-parallelism according to Conery's scheme, it is necessary to perform a data dependency analysis among subgoals at run-time. This can be very expensive. Furthermore, since data dependencies among subgoals may be changed after backward execution (backtracking), data dependency graphs have to be re-computed before re-starting forward execution. In Chapter 2, it will be shown that a static data dependency analysis performed at compile time can generate data dependency graphs (one for each clause) for Prolog programs. With static data dependency graphs available, it is possible to generate compiled code to exploit the AND-parallelism.

All the candidate clauses in a procedure can potentially be explored concurrently by exploiting OR-parallelism. However, OR-parallelism, either implemented with a parallel model [24] or a pipeline model [23], is not efficient for deterministic programs or non-deterministic programs which are used in a more deterministic way (i.e. only a few answers, instead of the full set of answers, are requested). Based on this consideration, OR-parallelism should be exploited only on spare resources (e.g. with lower priority), and the programmer should explicitly denote those procedure calls which can take advantage of OR-parallelism. Another problem in exploiting OR-parallelism is the need to keep competing OR-processes, one for each candidate clause in the invoked procedure, from simultaneously binding the same unbound variable. Some kind of "shadowing" mechanism must be supported. With a "shadowing" mechanism, the bindings are retained in the local memory of the process, and bindings are validated only when the results generated by the OR-process are requested. A proposed "shadowing" mechanism as well as the compiled code for OR-parallelism are described by Dobry et. al [30].

Stream-parallelism has been studied by researchers in functional [31] and data-flow language [32]. Streams can be used to implement infinite data structures and be lazily evaluated [31], or to achieve efficiency for passing finite data structures between producers and consumers and be eagerly evaluated [32]. Several attempts, e.g. Funlog [25], have been made to integrate functional programming language and logic programming language. A criticism of stream-parallelism is that the grain size of an entity to be synchronized has the potential for being too small to be efficient. Furthermore, the parallelism can be exploited only between the generator and the consumer of a stream.

Although it has been shown that in the worst case, unification cannot be sped up by

⁴Dependency checks among subgoals are used to determine whether they are dependent or not. In general, two subgoals are dependent if they contain common unbound variables at the time of invocation.

parallel computation⁵ [26], most of unifications encountered in actual benchmark programs can be. It seems feasible to take advantage of data dependency analysis to detect parallel unifiable terms and design special hardware to support parallel unification [33]. This is currently an active area of research.

1.6. A Static Data Dependency Analysis for Prolog Programs

From the above discussions, it is clear that efficient ways to exploit AND-parallelism and intelligent backtracking are important to improve the execution of Prolog. Current approaches of exploiting them are unrealistic because of the excessive overhead at run-time. By shifting part of the run-time overhead to compile-time using a static analysis, they can become realistic.

Using a static data dependency analysis to detect concurrency assumes that if two literals do not share any unbound variables then they are independent. This assumption is true in general, except that I/O, control predicates (e.g. cut, fail, repeat) and global effect predicates (e.g. assert and retract) have to be specially treated. With a static data dependency analysis, a data dependency graph is generated at compile time for each clause. Literals on the same 'layer' of the graph are independent in forwarded execution. It turns out that it is also possible to take advantage of data dependency graphs to achieve semi-intelligent backtracking and parallel backtracking. The information that must be supplied by programmers is minimal. Programmers are relieved from the burden of doing excessive annotations as required in a lot of other schemes. Although static analysis may not be as precise and refined as a run-time analysis, there is little run-time overhead and it can realistically lead to better performance.

A methodology to perform a static data dependency analysis for Prolog programs is described in Chapter 2. Its applications to intelligent backtracking and AND-parallelism are discussed in Chapter 3 and 4 respectively. Conclusions are given in Chapter 5.

1.7. The Thesis

My thesis is to show:

- (1) The performance of executing PROLOG programs can be improved through a static data dependency analysis.
- (2) Both AND-parallelism and intelligent backtracking can be exploited with low run-time overhead by using the generated data dependency graphs.
- (3) Changes in both the compiler and machine architecture will be considered in order to improve performance using this approach.

1.8. Contributions

Contributions of this research are:

- (1) Development of an efficient methodology for static data dependency analysis of PROLOG programs.
- (2) The use of static data dependency analysis to achieve intelligent backtracking.
- (3) The use of static data dependency analysis to reduce run-time overhead of AND-parallel execution.

⁵It is like general programming in this regard since it is possible to always explicitly create some strictly serial program.

- (4) An AND-parallel execution model to exploit both AND-parallelism and intelligent backtracking.
- (5) An evaluation of the cost and effectiveness of the static data dependency analysis.

CHAPTER 2

A STATIC DATA DEPENDENCY ANALYSIS OF PROLOG

2.1. Overview

As stated in chapter 1, the most important guideline for this research is to emphasize compile-time efforts in order to improve the performance of executing Prolog programs. In this chapter, it is shown that a static data dependency analysis can be done for Prolog programs [34] at compile time with a minimal amount of additional annotations given by programmers. In the next two chapters, it will become clear how both AND-parallelism and intelligent backtracking can be exploited with much lower overhead at run-time given a static data dependency analysis. The complexity of this data dependency analysis is about the same as that of automatic mode generation [35] for Dec-10 Prolog, and it shares a similar flavor and philosophy. But, instead of figuring out whether or not an argument is instantiated at the entry of a clause, this data dependency analysis generates a collection of data dependency graphs, one for each clause, and a specification of backtrack literals for each body literal in a clause. In Chapter 3, issues in backtracking are discussed, and a scheme to compile intelligent backtracking by using data dependency graphs is described. In Chapter 4, it is shown that this dependency analysis can also be applied to achieve AND-parallel execution.

2.2. A Static Data Dependency Analysis (SDDA) for Prolog Programs

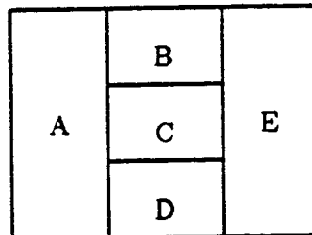
In this section, the methodology to do a static data dependency analysis (SDDA) for Prolog programs is illustrated. A map-coloring problem is first used as an example to show what it takes to construct a data dependency graph. Then the special characteristics of Prolog which affect the complexity of SDDA are examined. A few examples are chosen to illustrate difficulties in SDDA for Prolog programs because of these characteristics. Finally, the construction of the static data dependency analyzer is described. The analyzer itself is written in Prolog and appears in Appendix A.

2.2.1. Generating A Data Dependency Graph

A piece of program which solves a map-coloring problem with five areas to be colored (such that no two bordering regions are assigned the same color) is shown in Figure 2.1. The map is described by the 'map' clause. Colors for two bordering region are selected by the 'next' predicate. In order to derive the data dependency graph for the first clause, it is necessary to know:

- (1) How the clause is activated; That is, whether the arguments are still unbound variables, bound to *ground* terms¹, or aliases of each other when the clause is invoked.
- (2) The worst case binding of a variable after a procedure call.

¹A *ground* term is a term which does not contain any unbound variable.



```

map(A,B,C,D,E) :-
    next(A,B), next(A,C), next(A,D), next(B,C),
    next(C,D), next(B,E), next(C,E), next(D,E).
next(X,Y):- next1(X,Y).
next(X,Y):- next2(X,Y).
next1(green,red).
next1(green,yellow).
next1(green,blue).
next1(red,blue).
next1(red,yellow).
next1(blue,yellow).
next2(X,Y) :- next1(Y,X).

```

Figure 2.1 A map-coloring problem

From the above information, the status of variables can be determined and the data dependencies between the body literals can be derived.

For this example, assume that at the entry of the procedure call 'map' all variables are unbound. Then, by visually examining this program, it is not difficult to observe that at the exit of each procedure call 'next' all the arguments are ground terms. Based on the above assumption and this observation, the data dependency graph of the first clause (Figure 2.2) can be derived. In Figure 2.2, each node in the graph corresponds to a body literal. Literals on the same layer of a data dependency graph are independent during forward execution. That is, a literal can be executed as soon as all its predecessors are done, e.g. next(C,D) can be executed when next(A,C) and next(A,D) are done.

In the following sections, the method that the data dependency analyzer uses to determine (instead of the 'assume' and the 'visually observe' processes as described above) the activation and exit mode of a procedure call is developed. First, however, to help describe the problems involved, the special characteristics of Prolog are now examined.

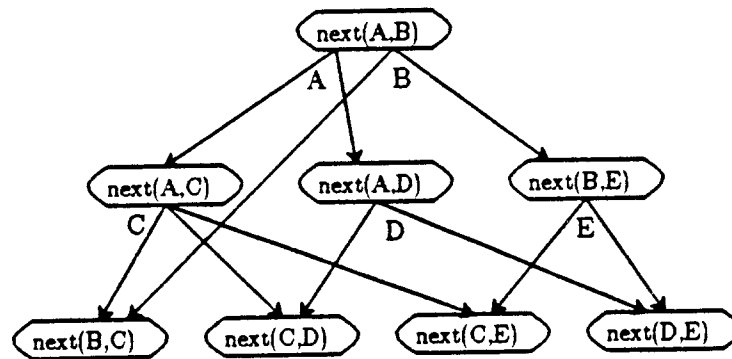


Figure 2.2

2.2.2. Special Characteristics of Prolog

Logic programming languages have several special characteristics which are different from conventional HLLs (high level languages) such as FORTRAN and PASCAL. Some characteristics which affect the SDDA are listed below:

- (1) A procedure can be invoked in multiple ways, e.g.

```

:-grandfather(GrandFather,mary).
:-grandfather(john,GrandChild).

```

The 'grandfather' procedure is invoked with the first argument a unbound variable in the first query, and the second argument a unbound variable in the second query.

- (2) Variables in Prolog are called *logical variables*. Logical variables may remain unbound or not grounded at the exit of a procedure call, e.g.

```

b :- ..., a(X,Y), ...
variable bindings at the exit of 'a' may be:
  X : unbound
  Y : bound to [2,Z] where Z is unbound

```

- (3) All the candidate clauses in a procedure may be tried one by one. Each candidate clause can generate a different set of variable bindings.
- (4) The scope of a variable is a single clause. The same variable name in different clauses represents different variables.
- (5) Variables are singly-bound. The binding of a variable can not be modified unless the clause which generates the binding fails and backtracking occurs. However, it is

possible that a logical variable is bound to a non-ground term and later on the unbound variables contained in this non-ground term become bound.

In the characteristics listed above, the first three make the SDDA more difficult, while the others make it easier as compared with the other HLLs like FORTRAN or PASCAL. This is illustrated by the following examples:

```
example_1:- alias(A,B), useA(A), useB(B).
alias(X,X).
```

```
example_2:- coupled(A,B), useA(A), useB(B).
coupled([X|L1],[2,Y|L2]):- alias(X,Y).
```

```
example_3 is a query:
:- coupled([X|L1],[2,X|L2])?
coupled(A,B):- useA(A), useB(B).
```

```
example_4(X,Y):- goal1(X,Y), goal2(X,Y), goal3(Y).
goal1(3,3).
goal1(3,X).
goal2(X,2).
goal3(Z):- more(Z), - - -
```

In the first two examples, if we look only at the first clauses, we may wrongly assume that useA(A) and useB(B) are independent and can be executed in parallel after returning from the first procedure call, 'alias', in the body. However, because of logic variables, it is possible that two variables become aliases or coupled. Two variables are coupled if they are bound to two coupled terms, and two terms are coupled if they share at least a common, unbound variable. The introduction of aliases or coupling may be several levels deep in the proof tree² or within terms as shown in the second example. The problem of executing useA(A) and useB(B) concurrently in the first two examples is the possibility of creating binding conflicts for shared unbound variables. UseA(A) and useB(B) can be executed concurrently, however, if the complete sets of answers from useA(A) and useB(B) are "joined" together; or if the set of answers generated by useA(A) are "streamed" to useB(B). But, both schemes are impractical if users are interested in a small part of the solution set; or are inefficient for a deterministic program. The static data dependency graphs for the first two examples are shown in Figure 2.3 (a) and 2.3 (b). In Figure 2.3, variables which are aliases are shown in parentheses, variables which are in the same coupling group are put in a set.

In the third example, there is no way to tell at compile time whether useA(A) and useB(B) are independent unless the compiler is supplied with information about how each *entry procedure* is to be invoked³. An *entry procedure* is a procedure which is called directly

²A proof tree is an AND/OR tree with only one OR branch, which corresponds to the currently activated clause of the procedure, shown for each OR subtree.

³Alternatively, a separate data dependency analyzer can be compiled for each possible query (combination of modes).

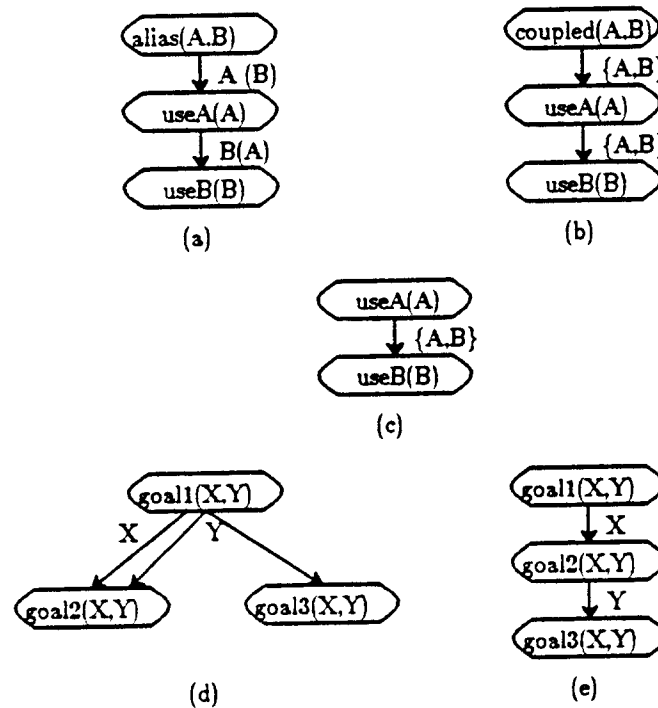


Figure 2.3

from the top level query. If the programmer has denoted that the procedure 'coupled' will be invoked with its two arguments coupled together, then the correct data dependency graph (Figure 2.3 (c)) can be generated. In the fourth example, there are two possible dependency graphs depending on whether `goal1(3,3)` or `goal1(3,X)` is selected to unify with `goal1(X,Y)`. `Goal2(X,Y)` and `goal3(Y)` are independent in the first case (Figure 2.3 (d)), but are dependent in the second case (Figure 2.3 (e)). If only one data dependency graph is generated for a clause, then the analyzer must examine all the possible candidate clauses and do a worst-case analysis.

It is worthwhile to examine the cost of doing a dynamic data dependency analysis [22]. In order to find at run time whether or not two literals are independent, it is necessary to find all the unbound variables contained in the arguments of the literals. Two literals are independent if they do not contain any common unbound variable. To determine this, it requires traversing all the argument terms. Furthermore, this run-time analysis needs to be performed again after backtracking in order to take into account of the new set of variable bindings generated by a new candidate clause. Obviously, this can be quite expensive. A much cheaper scheme [36] based on tagging has been designed. However, it still involves some run time costs and its dependency check is not as thorough as the SDDA. In that

scheme, two literals are claimed to be independent only when

- (1) One of the literals is grounded, or
- (2) All the arguments of the literals are unbound variables and the literals do not share any unbound variable.

2.2.3. Activation Mode Declaration

A clause is invoked either from the top level query or from literals in the body of some clause invoked by the query. This implies that the minimal information a programmer must supply is the worst-case activation mode of the query, e.g. the worst-case activation mode for `example_3` may be declared as

```
entry_(coupled(s,s)).
```

This means that the predicate "coupled", with two arguments, may be a top level query with its two arguments coupled together in the worst case. The symbols used in declaring the worst-case activation modes of entry procedures are summarized below:

- "sN" - coupled term in Nth coupled group.
- "s" - coupled term. (implies there is only one coupled group)
- "g" - ground term.
- "i" - independent term.⁴

Only one declaration, which describes the worst case activation, is necessary for each entry procedure in the program. This extra work can be handled quite easily by the programmer. If it is desirable, the system can verify that the activation of the top level query is consistent with the declared worst-case activation mode. Throughout this chapter, the term "worst case mode" is mentioned frequently. Worst case mode is determined in the following way. In the above classification, "s" is worse than "i" and "i" is worse than "g". For example, if a predicate 'foo' with three arguments has the following possible activations:

```
(foo,3,[g,s,s]).
(foo,3,[i,g,g]).
(foo,3,[g,g,g]).
```

Then the worst case activation mode should be `(foo,3,[i,s,s])`.

2.2.4. A Methodology

Up to this point, it is assumed that if two literals do not share any unbound variables then they are independent. This assumption is not true in general, e.g. I/O and control predicates such as *cut* and *fail* have to be handled in a special way. However, let us assume it is true now and discuss special handling in section 2.2.6.

Prolog programs are executed from left-to-right. In SDDA, literals in a clause are

⁴An independent term is a term which is neither a ground term nor a coupled term.

examined according to the order of execution⁵. Variables in a clause are classified by the analyzer into three sets G, C, and I:

- G - {V | V is bound to ground term}
- C - {V | $V \in EC$, EC is an equivalence class which contains variables that may be coupled together}
- I - {V | V is neither in G, nor in C}

Again, two terms are coupled if they share at least a common, unbound variable. Two variables are in the same coupling group (equivalence class) if the analyzer detects that it is possible for them to be bound to two coupled terms. A unbound variable belongs to set I, but not all variables in set I are unbound. It is possible that a variable is instantiated to a non-ground term and still be in set I so long as it is not coupled with the other variables. The triple (G,C,I) is called the *variable status* (i.e. the status of the variables).

The data dependency graph of a clause can be derived when the data dependency analyzer examines body literals from left to right and keeps track of the worst case variable status. To do this, the analyzer must find the worst case binding of a variable at the exit of a procedure call. It is similar to the problem encountered in the automatic mode generation for Dec-10 prolog [35]. But, the major difference (besides the methodologies) between the SDDA and the automatic mode generation is that, instead of just figuring out whether or not at the entry of a procedure call the arguments are instantiated or not, the SDDA also keeps track of the variable status in clause bodies and generates data dependency graphs.

The way that the analyzer processes a clause with a known activation mode is shown in Figure 2.4. From the head literal and the activation mode of this clause, the analyzer first constructs the initial variable status (G_0, C_0, I_0) . Assume that the current variable status before executing g_1 is $(G_{t_1}, C_{t_1}, I_{t_1})$. The analyzer uses the current variable status to figure out the activation mode of g_1 . Then, it derives the exit mode of g_1 for this activation. From the exit mode, it knows the worst case variable bindings of this procedure call. With this, it updates the old variable status $(G_{t_1}, C_{t_1}, I_{t_1})$ to get the new variable status (G_p, C_p, I_p) . This process is continued until the end of the clause is reached, and the final variable status (G_n, C_n, I_n) is derived. From the final variable status and the head literal, it derives the exit mode of this clause for this activation. Since a procedure call may invoke all the candidate clauses in a procedure, all candidate clauses have to be processed in the same manner in order to get the exit mode of a procedure call for a given activation. The exit mode of a procedure call for a given activation is the worst case exit mode among all the candidate clauses. An Example is shown in Figure 2.5.

During the derivation, a variable can be in only one set at any time, yet it may be switched from one set to another as each body literal is examined, e.g. a unbound variable in a literal g_1 may be switched from the set I to the set G when the analyzer detects that it will be grounded when g_1 is executed. Two equivalence classes may be merged together if two variables, one from each class, become coupled together. If an equivalence class has only one variable in it, then this equivalence class should be removed and the variable should be added to the set I. An example is shown in Figure 2.6 (a). It shows snapshots of the variable status as each literal is examined by the analyzer.

⁵In general, to perform the SDDA any *static* ordering algorithm can be chosen.

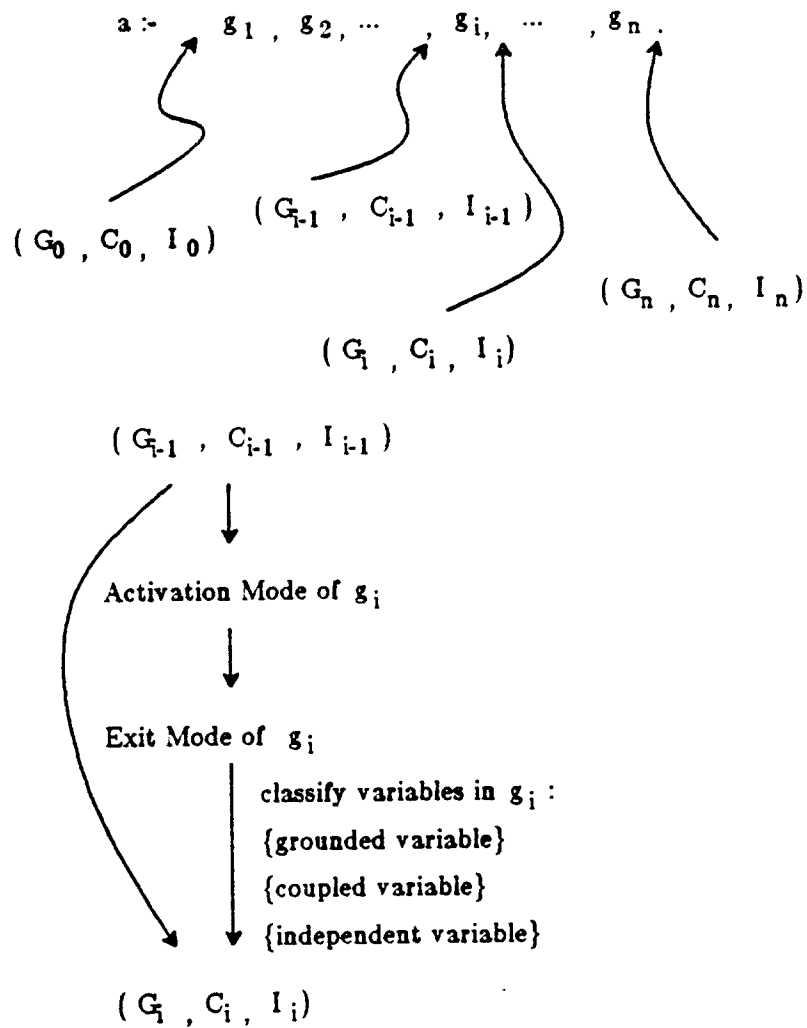


Figure 2.4 Processing a clause

Activation Mode of the clause

↓
(G_0 , C_0 , I_0)

↓
⋮
↓
(G_i , C_i , I_i)

↓
⋮
↓
(G_n , C_n , I_n)

↓
Exit Mode of the clause
for this activation

Figure 2.4 Processing a clause (continued)

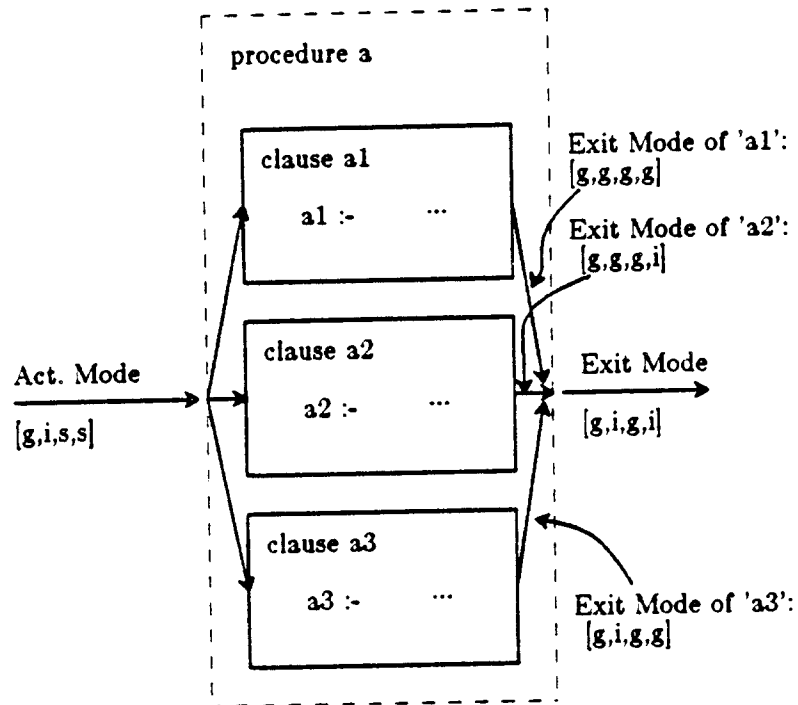


Figure 2.5 Processing a procedure

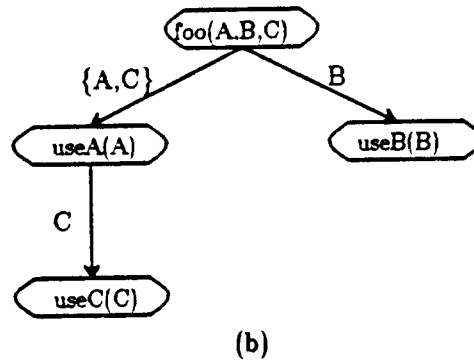
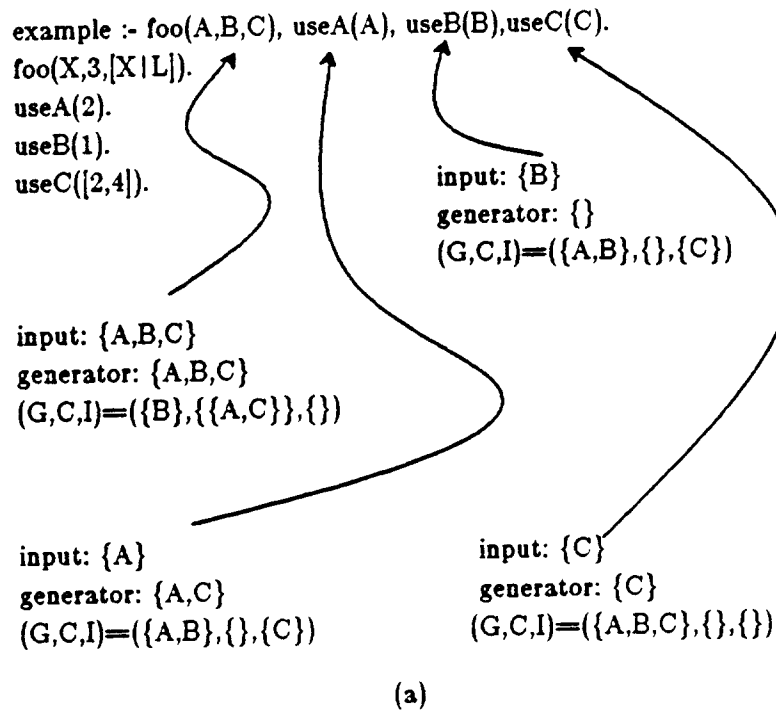


Figure 2.6 Generating a data dependency graph

The data dependency graph can be generated as follows:

- (1) The default "generator" of a variable is the calling literal.
- (2) A literal which contributes to the binding of a independent variable will be made the "generator" of that variable.
- (3) A literal which contributes to the binding of at least one variable in an equivalence class will be made the "generator" of all the variables in the equivalence class.

- (4) A predecessor of a literal, say g_p , is the closest "generator" of one of the variables in g_p . Of course, a literal can have several predecessors each of which is the closest "generator" of at least one of the variables in the literal.

The data dependency graph of the example in Figure 2.6 (a) is derived according to the above scheme. It is shown in Figure 2.6 (b). In Figure 2.6 (a), the set of variables for which a literal is the generator is labeled as "generator", the set of variables of a literal is labeled as "input".

This analyzer starts the analysis by first examining all the entry procedures with declared activation modes. After processing all the clauses of the entry procedures, a collection of activation modes are available for the other procedures. The analyzer can explore either all the possible activation modes of a procedure or just the worst case activation. In the latter case, only one data dependency graph is generated per clause. In the current implementation, the analyzer explores only the worst case activation of a procedure. It is clear that the worst case activation mode of a predicate may be changed as the analysis proceeds. Even the entry procedures may later on have activation modes which are worse than the initial declarations given by the programmer. Whenever that happens, the data dependency analyzer will re-analyze those procedures.

As stated above, given the activation mode of a literal, in order to derive its exit mode, it is necessary to find the exit mode of each candidate clause for this activation. The exit mode of the procedure with a given activation is the worst case exit mode among all candidate clauses. This exit mode is recorded in a data base of the analyzer as a quadruple (F,N,Act,Exit), i.e. (functor of the literal, its arity, activation mode, exit mode). If, later on, the same activation occurs, then the corresponding exit mode can be retrieved without re-deriving it. For the map-coloring problem in Figure 2.1, if an activation mode is declared for the entry procedure 'map' as follows:

```
entry_(map(i,i,i,i)).    % arguments are independent variables
```

the analyzer can derive exit modes for all the possible activations as listed in Figure 2.7. A representation of the graphs generated by the analyzer is shown in Figure 2.8. One graph is generated for each clause. These representations are ordered in the same order as corresponding candidate clauses in a procedure. In Figure 2.8, the predicate "pred_" describes the data dependency graph. Pred_(map,5,list) means that "list" is the data dependency list of a candidate clause of map/5. Each element in the data dependency list describes the data dependency of a literal, for example [2,1,0] means that the predecessor literals of literal #2 in the body of the clause are the literal #1 and the calling literal (#0). The structure "back_" describes the backtrack literals of each literal in a clause. Its structure is similar to "pred_", for example [6,1,3] means that the backtrack literals of literal #6 are literal #1 and literal #3 (these two backtrack literals correspond to two different types of backtracking as discussed in Chapter 3).

In the above description, there is a "chicken and egg" relationship that must be resolved. Because, in order to determine the exit mode of an activation, it is necessary to determine the exit mode of each body literal. But what if a body literal has the same activation (F,N,Act) as the calling literal? This circularity can happen for recursive clauses or mutually recursive clauses. Circularity can be detected by maintaining an invocation chain. An invocation chain is a list of activations, i.e. (F,N,Act)s, of all the ancestor literals. Since a recursive clause generally must terminate on some other candidate clauses of the same procedure, we can ignore this recursive clause when the cycle is detected and find the exit

| Functor | Arity | Act Mode | Exit Mode. |
|---------|-------|-----------|-------------|
| next1 | 2 | [i,i] | [g,g] |
| next2 | 2 | [i,i] | [g,g] |
| next | 2 | [i,i] | [g,g] |
| next1 | 2 | [g,i] | [g,g] |
| next1 | 2 | [i,g] | [g,g] |
| next2 | 2 | [g,i] | [g,g] |
| next | 2 | [g,i] | [g,g] |
| next1 | 2 | [g,g] | [g,g] |
| next2 | 2 | [g,g] | [g,g] |
| next | 2 | [g,g] | [g,g] |
| map | 5 | [i,i,i,i] | [g,g,g,g,g] |

Figure 2.7 Mode quadruples

```

pred_(next1,2,[]).
back_(next1,2,[]).
pred_(next1,2,[]).
back_(next1,2,[]).
pred_(next1,2,[]).
back_(next1,2,[]).
pred_(next1,2,[]).
back_(next1,2,[]).
pred_(next1,2,[]).
back_(next1,2,[]).
pred_(next1,2,[]).
back_(next1,2,[]).
pred_(next,2,[[1,0]]).
back_(next,2,[[1,0]]).
pred_(next2,2,[[1,0]]).
back_(next2,2,[[1,0]]).
pred_(next,2,[[1,0]]).
back_(next,2,[[1,0]]).
pred_(map,5,[[1,0],[2,1,0],[3,1,0],[4,1,2],[5,2,3],[6,1,0],[7,2,6],[8,3,6]]).
back_(map,5,[[1,0,0],[2,1,1],[3,1,2],[4,2],[5,3],[6,1,3],[7,6],[8,6]]).

```

Figure 2.8 Output of the SDDA

mode of the body literal by considering the rest of the candidate clauses. It can be illustrated with a simple example:

```

concat([],L,L).
concat([X|L1],L2,[X|L3]):- concat(L1,L2,L3).

```

Assume the analyzer wants to determine the exit mode of an activation ($\text{concat}, 3, [g, g, i]$). It first examines the first candidate clause. From the activation mode it knows that L will be grounded, so the initial variable status is $(G_0, C_0, I_0) = (\{L\}, \emptyset, \emptyset)$. Since there are no body literals, this is also the final variable status. From the final variable status and the head literal, it deduces that the exit mode corresponding to the first clause should be $[g, g, g]$. It then examines the second clause. From the activation modes it gets $(G_0, C_0, I_0) = (\{X, L1, L2\}, \emptyset, \{L3\})$. From this initial variable status, it derives that the activation mode of the body literal is also $[g, g, i]$. With the same (F, N, Act) as the head, a cycle exists. So, in the derivation of the exit mode of the body literal, it considers only the first clause. Since the exit mode of the first clause for this activation is known to be $[g, g, g]$, at the exit of the second clause, the variable status (G, C, I) becomes $(\{X, L1, L2, L3\}, \emptyset, \emptyset)$. From this final variable status, it derives that the exit mode of the second clause for the activation ($\text{concat}, 3, [g, g, i]$) should be $[g, g, g]$. By comparing the exit modes of both clauses, the analyzer deduces

$(F,N,Act,Exit)=(concat,3,[g,g,i],[g,g,g]).$

This information is asserted into a data base of the analyzer for future references. Another example is shown in Figure 2.9 (a). It is a quicksort program using a difference list [37]. The result of mode derivations is shown in Figure 2.9 (b) and the output of the analyzer is shown in Figure 2.9 (c). It can be observed that the exit mode of the activation (qsort,2,[g,i]) is [g,i] instead of the anticipated [g,g]⁶. The reason for this is that the current implementation of the analyzer has been simplified in order to achieve efficiency. Some optimization techniques which can improve the quality of the exit mode derivation are discussed in the next section.

```

entry_(quicksort(g,i)).
quicksort(Unsorted,Sorted):-
  qsort(Unsorted,Sorted-[]).
qsort([],Rest-Rest).
qsort([X|Unsorted],Sorted-Rest) :-
  partition(Unsorted,X,Smaller,Larger),
  qsort(Smaller,Sorted-[X|Sorted1]),
  qsort(Larger,Sorted2-Rest),
  Sorted2=Sorted1.
partition([],_,[],[]).
partition([X|Xs],A,Smaller,[X|Larger]) :-
  A < X,
  partition(Xs,A,Smaller,Larger).
partition([X|Xs],A,[X|Smaller],Larger) :-
  A >= X,
  partition(Xs,A,Smaller,Larger).

```

Figure 2.9 (a) Quicksort

| Functor | Arity | Act Mode | Exit Mode. |
|-----------|-------|-----------|------------|
| partition | 4 | [g,g,i,i] | [g,g,g,g] |
| qsort | 2 | [g,i] | [g,i] |
| quicksort | 2 | [g,i] | [g,i] |

Figure 2.9 (b) Mode quadruples

⁶Note that this is not an error. It does not matter when the analyzer fails to analyze the exact variable status so long as the variable status is better than, as [g,g] is better than [g,i], what the analyzer has derived. Of course, it may result in detecting less concurrency.

```

pred_(partition,4,[]).
back_(partition,4,[]).
pred_(partition,4,[[1,0],[2,0]]).
back_(partition,4,[[1,0],[2,0]]).
pred_(partition,4,[[1,0],[2,0]]).
back_(partition,4,[[1,0],[2,0]]).
pred_(qsort,3,[[1,0],[2,1,0],[3,1,2,0]]).
back_(qsort,3,[[1,0,0],[2,1,1],[3,2]]).
pred_(qsort,3,[]).
back_(qsort,3,[]).
pred_(quicksort,2,[[1,0]]).
back_(quicksort,2,[[1,0]]).

```

Figure 2.9 (c) Output of the SDDA

2.2.5. Optimizations

Three optimization schemes which can be incorporated into the analyzer in order to extract more precise information about the status of variables are discussed here. They are (1) structure matching, (2) refined activation mode declaration, and (3) separating the graph generation phase from the exit mode derivation phase.

2.2.5.1. Structure Matching

Structure matching can be very useful in SDDA as illustrated in the following example. Assume a clause has a head literal $\text{foo}(W, h(Y, Z))$. Consider the following two calling literals:

- (1) $\text{foo}(3, h(X, X))$.
- (2) $\text{foo}(3, h(X, 3))$.

Assume X is not grounded. According to the methodology described in the previous section, the analyzer will derive the fact that both literals have the same activation mode $[g, i]$. Clearly, the initial variable status (G_o, C_o, I_o) of the first case should be $(\{W\}, \{\{Y, Z\}\}, \{\})$, and of the second case should be $(\{W, Z\}, \{\}, \{Y\})$. However, since only the activation mode is used to derive the initial variable status, the analyzer treats both cases in the same way. That is, for the second calling literal, the initial variable status derived by the analyzer will be the same as the first. This is not desirable. To make the SDDA perform a little bit better, in the current implementation an independent term which contains coupled arguments are denoted by an intermediate mode 'r'. If an argument in the head literal has 'r' as its activation mode, and contains more than one variables, then all its variables are put into the same coupling group; if the argument of the head literal only contains one variable, then the variable is considered as an independent variable. With this scheme, in the above example the first calling literal will have the activation mode $[g, r]$ and the second $[g, i]$. Then, the initial variable status for the second calling literal will be $(\{W\}, \{\}, \{Y, Z\})$ which, although not as good as the ideal variable status, is better than before. However, if the structure matching

is incorporated into the SDDA, the ideal variable status can always be derived. Also consider the quicksort program in Figure 4. During the data dependency analysis, the analyzer tries to find the exit mode of the literal `qsort(Unsorted,Sorted-[])`. Since it does not perform any structure matching between the literal and the head of the first candidate clause (only the activation mode `(qsort, 2, [g,i])` is passed over), the initial variable status (G_0, C_0, I_0) is $(\{X, \text{Unsorted}\}, \emptyset, \{\text{Sorted}, \text{Rest}\})$. If structure matching is performed, the analyzer can detect that `Rest` will be matched with the empty list `[]`. In that case, (G_0, C_0, I_0) will become $(\{X, \text{Unsorted}, \text{Rest}\}, \emptyset, \{\text{Sorted}\})$ which is more precise than before.

Another advantage of incorporating the structure matching into the SDDA is that, in deriving the exit mode of a calling literal, the analyzer can ignore clauses whose head literals have structures incompatible with the calling literal. For example, calling literal `foo([X|L], - - -)` can not activate a clause whose head is, say, `foo(9, - - -)` or `foo([], - - -)`.

The disadvantages of exploiting structure matching is that the SDDA becomes more complicated, e.g. the activation mode may contain structure in each argument position, the worst-case activation modes takes longer time to resolve, worst-case variable bindings are more difficult to resolve at the exit of a procedure call, etc.

2.2.5.2. Refined Activation Mode Declaration

In the current scheme, programmers can only classify an argument into three different types of terms, i.e. ground terms, coupled terms, and independent terms, in the activation mode declarations. More refined activation mode declaration can be implemented together with the structure matching, e.g.

```
entry_(foo(g,[g|i])).
```

which means that the entry procedure 'foo' is activated with the first argument a ground term and the second argument a list with its head a ground term. If the head of a candidate clause is `foo(W,[Z|L])`, then the analyzer can derive $(G_0, C_0, I_0) = (\{W, Z\}, \emptyset, \{L\})$ instead of $(\{W\}, \emptyset, \{Z, L\})$.

2.2.5.3. Separating Graph Generation From Exit Mode Derivation

In the current implementation, dependency graphs are generated at the same time the analyzer derives the exit mode of each body literal and updates the variable status. This approach simplifies the implementation, and takes advantage of the fact that both exit mode derivation and graph generation requires the knowledge of the variable status. There is, however, a fundamental difference between exit mode derivation and graph generation. In graph generation, literals in a clause have to be examined from left-to-right (or in any fixed order determined by a static ordering algorithm). The exit mode of an activation of a clause, however, should be order independent. It turns out that for a literal, which invokes a recursive clause, a more accurate exit mode of the activation can be derived by considering that a unbound variable may be bound later on by other literals in the same clause. A more detailed discussion on this is in the automatic mode generation paper by Mellish [35].

By separating the graph generation phase from the exit mode derivation phase, the analyzer can perform better in exit modes derivation and graph generation.

The data dependency analyzer has been tested on several benchmark programs, e.g. the queens problem, the mu-math problem, an automatic circuit design problem, a map-coloring problem, and the quicksort. The analyzer is able to detect as much parallelism as exists in

the programs. This result is very encouraging.

2.2.6. Handling Control Predicates, I/O, and Global Effect Predicates

A program, for any practical purpose, must have I/O and other side-effects. One of the difficulties in static data dependency analysis in Prolog is that the global effect predicates, namely *assert* and *retract*, can be used in a fairly unrestricted way. For example, in Waterloo Prolog a new clause can be asserted into any position in a procedure, i.e. as the first clause or as the Nth clause. Also, an assertion can show up at any place in the program. These raise the cost of the static data dependency analysis, because the analyzer must find all the procedures being affected by *assert* or *retract* under a literal and find out the axiom-dependencies between literals in the same clause. A literal is axiom-dependent if it invokes procedure calls which are affected by *assert* or *retract*, or vice versa⁷. I/O operation is treated in the following way. All the previous literals have to be completed before an I/O can be started. Control predicates also have to be specially treated because their significance can not be detected by the data dependency analysis. In the current implementation, all the literals before control predicates, e.g. *cut*, *fail*, and *repeat*, have to be completed before these control predicates can start, and the subsequent literals can not backtrack to previous literals without going through these control predicates.

2.3. A Hidden Problem

In the previous section, it is shown that a pure data dependency analysis is not enough to detect dependency, e.g. control predicates have to be handled in a special way. In this section, a hidden problem in the approach of the SDDA is pointed out with the following example.

```
a(X):- test_for_ok(X), work_on(X).
```

In this example, if X is grounded at the entry of this clause, the data dependency analyzer will allow both literals in the body to proceed concurrently. However, there is a logical dependence between these two literals. The second literal may contain meaningless, inaccurate or unbounded work unless the first literal succeed. To handle this problem, the programmer should either use a dummy variable or use a special annotation to denote the logical dependence. The programmers, who are accustomed to the left-to-right sequential execution of Prolog, need to have a special mind-set to detect this potential problem.

2.4. Complexity Analysis of the SDDA

As mentioned in Chapter 1, a Prolog program can be treated as an AND/OR search tree in which an AND subtree corresponds to a clause and an OR subtree corresponds to a procedure. In the SDDA, if ignoring activation modes and recursions, the analyzer basically performs a depth-first tree traversal of the AND/OR tree starting from the OR subtrees of the entry procedures. The complexity of the algorithm would be linear. However, because of activation modes, an OR subtree may need to be examined several times, each time with a worse activation mode. Assume the number of arguments of all the procedures in a program is bound by a constant K. Then, the number of different activation modes of a procedure is bound by 3^k . The constant 3 is introduced because that an argument can have one of the

⁷A simpler and cheaper scheme is to treat a literal, which invokes global effect predicates, as an I/O (see the static data dependency analyzer shown in Appendix A).

three possible activation mode, i.e. g, i, or s (to simplify the analysis, sN is not included here). The complexity of this algorithm would still be linear. Here, it is assumed that an existing mode quadruples can be retrieved with constant time (for example, hashing is used rather than a linear search). That is, it costs almost nothing to retrieve the exit mode of an old activation. Recursions are handled by considering all the other clauses except the clauses of which the recursive call is a descendant. Now, assume that there are M recursive clauses in a procedure, each of which contains one recursive call. If all the recursive calls happen to have the same activation mode, then the number of recursive clauses that need to be examined is M! (M factorial). This would be quite expensive especially when the AND subtree of a recursive call is non-trivial. For example, in one of the benchmark programs, the automatic circuit design problem⁸, the recursive procedural call 'ngate' calls another recursive procedure call 'tgate', where the procedure 'tgate' includes several clauses. To reduce the cost of the SDDA, the programmer can use declarations to declare a mode quadruple, for example:

```
mode_(ngate,3,[i,i,i],[i,i,i]).
```

This declares that the exit mode of the activation (ngate,3,[i,i,i]) is [i,i,i]. When the SDDA searches its data base and finds a mode quadruple with the given activation, it stops exploring the corresponding OR subtree and simply retrieves the exit mode.

Table 2.1 compares speeds of the data dependency analyzer and the PLM compiler [14] for several benchmark programs. It can be observed that the cost of the SDDA is cheaper than the compilation except for programs which contains many recursive clauses (as in mu-math and automatic circuit design problem).

| Benchmark | SDDA | PLM Compiler |
|--|-----------|--------------|
| map-coloring | 20.23sec | 25.23sec |
| determinate concat | 2.67sec | 7.87sec |
| quicksort | 17.67sec | 62.90sec |
| population query | 18.48sec | 61.77sec |
| serialize | 59.02sec | 62.30sec |
| mu-math | 247.22sec | 83.10sec |
| queens | 19.53sec | 51.92sec |
| automatic ckt (w/o mode declaration) | 197.63sec | 116.38sec |
| automatic ckt (with mode declaration) | 38.85sec | 116.38sec |

Table 2.1 Comparison between the SDDA and the PLM compiler

⁸Most of the benchmark programs mentioned in this thesis are in Appendix B for reference.

2.5. Conclusion

In this chapter, a method to accomplish a SDDA for Prolog programs has been derived. The cost of performing a SDDA is cheaper than compilation for most of the benchmark programs. The SDDA and the automatic mode generation [35] share a similar flavor and philosophy, both are concerned with deriving, automatically, worst-case variable bindings. But, instead of just figuring out worst-case variable bindings at the entry of a clause, the SDDA also generates a collection of data dependency graphs, one for each clause. In the automatic mode generation, Mellish [35] estimated that the cost of deriving instantiation modes is about four times as expensive as compiling the same program by the Dec-10 compiler. The approach used in the automatic mode generation is to first construct a dependency network⁹, then iteratively propagate the mode constraints through the network until a fix-point is reached. This method is designed to handle recursive clauses, but is less efficient in handling non-recursive clauses. On the other hand, the dependency analyzer described in this chapter walks through the AND/OR tree in the depth-first order. It keeps track of variable status and constructs the dependency graph at the same time, re-examines an OR subtree only when the worst case activation mode has been changed. It may cost more in handling nested recursive clauses, but it wastes no extra effort in handling non-recursive clauses. Although the objective of the SDDA is different from automatic mode generation, the methodology used in this chapter can be adopted for that purpose. In the next two chapters, the SDDA will be employed to achieve intelligent backtracking and AND-parallelism.

⁹This dependency network is constructed by looking only the name of the variable. It is different from the dependency analysis discussed in this chapter where the worst case variable bindings are considered.

CHAPTER 3

COMPILING INTELLIGENT BACKTRACKING FOR A PROLOG MACHINE

3.1. Overview

Prolog programs are executed from left-to-right and top-to-bottom. A depth-first execution strategy is employed. It backtracks to the most recently activated procedure call having a choice point (untried alternative) when a failure occurs. However, the most recently activated choice point may not be able to generate any new variable bindings which can help solve the subgoal that fails. To backtrack intelligently is to avoid this kind of redundancy as much as possible. *Intelligent* backtracking based on run-time bookkeeping has been studied by Cox-Pietrzykowski-Matwin [21] and Bruynooghe-Pereira-Porto [18,19]. Their approach entails considerable run-time overhead. A static data dependency analysis for Prolog programs (as discussed in Chapter 2) will allow the generation of compiled code with improved backtracking behavior. Taking advantage of data dependency graphs to achieve better backtracking behavior was first suggested by Conery in his thesis [22]. In his approach, however, it is necessary to determine data dependencies among literals at run-time. Furthermore, since data dependencies among literals may be changed upon backtracking, Conery's method requires re-computing data dependencies before forward execution is resumed. Obviously, this is very costly. Instead, static data dependency analysis can generate a data dependency graph for each clause. From the data dependency graph, the literal which a literal should backtrack to in case of failure can be determined. However, since the analysis is static and based on worst-case considerations, it is not as accurate as a run-time analysis. Thus, it is called *semi-intelligent* backtracking [38]. In this chapter, a scheme to compile semi-intelligent backtracking for a Prolog machine (PLM@Berkeley) is described. The advantages of this approach are demonstrated with simulation results. It is shown that although the cost of compiling a program is increased, the overhead required to support semi-intelligent backtracking at run-time is very small.

3.2. Backtracking

In general, there are two kinds of backtracking [16], shallow backtracking and deep backtracking. Shallow backtracking is backtracking to alternative candidate clauses when the head of the current candidate clause can not unify with the calling literal. Deep backtracking occurs when there are no untried candidate clauses for the current procedure call, so backtracking occurs to any previously invoked procedure call having a choice point. Backtracking can be done more effectively by using clause indexing (hash on the arguments of the calling literal) [16]. Clause indexing selects qualified candidate clauses of a procedure call. Its effect is local to a procedure. On the other hand, intelligent backtracking reduces the number of redundant, deep, backtracking steps, and its effect is inter-procedural. Current Prolog interpreters/compiler implement shallow backtracking and deep backtracking uniformly: They always backtrack to the most recently activated procedure call having a choice point when a failure occurs. Since this style of backtracking is purely mechanical, it is called *naive* backtracking. A simple example is shown below to illustrate naive versus intelligent backtracking.

Example 3.1.

```

g(X,Y) :- a(X), b(Y), c(X).
a(X):- d(X), e(X).
a(X):- ...
b(2):- ...
b(3).
b(Y):- ...
c(Z):- s(Z), t(Z).
d(3).
e(3).
s(3).
t(1).
t(2).
t(4).

```

The AND/OR tree of this example is shown in Figure 3.1. In this figure, the candidate clause of a procedure 'a' is labeled as 'a1', 'a2', ... It is clear that when the literal c(X) fails during forward execution, backtracking should occur directly to the choice point of 'a'. Since the failure of 'c' is due to the binding of variable X bound by 'a', backtracking to 'b' will be useless.

3.2.1. Run-Time Intelligent Backtracking

Run-time intelligent backtracking has been studied by Cox-Pietrzykowski-Matwin [21] and Bruynooghe-Pereira-Porto [18,19]. Their approaches are based on finding minimal (or maximal) deduction subtrees¹ such that unification is impossible (or possible). Since these two approaches are complementary to each other, in the following only the method developed by Bruynooghe-Pereira-Porto is outlined.

Intelligent backtracking by finding minimal failing deduction subtrees is based on the observation that a naive interpreter always considers the *whole* proof tree as the failing tree when a failure occurs. If a smaller deduction *subtree* can be determined to cause the failure, then the redundant search space can be pruned. Their method can be summarized as follows:

- (1) Construct the deduction subtree of a term. The deduction subtree of a term consists of procedure calls which contribute, directly or indirectly, to the binding of the term. The deduction subtree of a term is constructed during unification. An inconsistent deduction subtree is derived when two terms fail to unify.
- (2) A leaf of the inconsistent subtree is selected as the culprit and removed. The fact that the remaining subtree causes failure of the current candidate clause of the culprit is asserted into a data base. Next an attempt is made to grow the tree again (normally by selecting another candidate clause, if any, of the culprit).
- (3) If all the candidate clauses of a procedure fail, their associated failing instances being asserted in the database are combined to become the unsolved subtree of this procedure call.

¹A deduction tree looks like a skeletal proof tree. It is a proof tree with all substitutions completely ignored.

approach, can be quite useful for many applications and is easily supported. The technique is briefly described in the next section.

3.2.2. Semi-Intelligent Backtracking

As discussed in Chapter 2, with SDDA a data dependency graph can be generated for each clause in a Prolog program. Literals on the same layer of a data dependency graph are independent during forward execution. Thus, a failure of a literal occurring during forward execution has nothing to do with the other literals on the same layer. During backward execution, literals on the same layer are not independent (this will be explained later). In spite of this, a more intelligent form of backtracking can still be achieved by using the data dependency graph based on worst case considerations.

One immediate limitation of this approach is that the intelligent backtracking behavior is confined within a clause, as illustrated with the following examples.

Example 3.2.

```
a(X,Y,Z) :- b(X,Y), c(Y,Z), d(X).
b(X,Y) :- e(X), f(Y).
```

In this example, assume X, Y, and Z are bound to integers by 'e', 'f', and 'c' respectively. The AND/OR tree and the data dependency graph of the first clause is shown in Figure 3.2. When 'd' fails during forward execution, it should backtrack to 'b' without going through 'c'. This can be easily compiled with the help of the SDDA. But without further information, at best 'd' can only backtrack to the last choice-point under 'b', which could be the choice-point of 'f'. However, the desired behavior is for 'd' to backtrack directly to the choice-point of 'e'. In order to achieve this, it would be necessary to know which literal in a clause binds which variables and to have a method to determine the cause of failures.

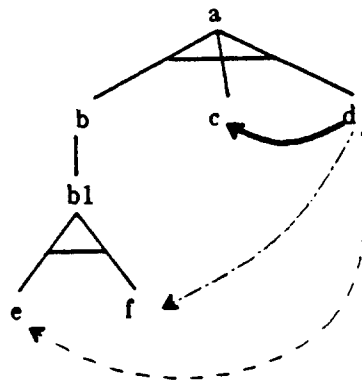
Example 3.3.

```
:- perm([1,2,3],S), safe(S).
perm([],[]).
perm([X|L],[U|V]) :- del(U,[X|Y],W),perm(W,V).
del(X,[X|Y],Y).
del(U,[X|Y],[X|V]) :- del(U,Y,V).
safe(S) :- ...
```

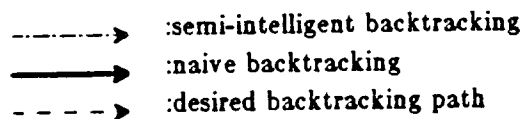
This is a fragment of the program which solves the 3-queen problem [18]. The procedure 'perm' generates a possible configuration S, and the procedure 'safe' checks whether the configuration is safe or not. It can be seen that a configuration, which contains three positions of queens, is generated by four tail-recursive calls of 'perm', each of which has a choice-point in its child procedure 'del'. Now, assume that in 'safe' it is found that the configuration is not safe because of the position of the first queen which is generated by the first recursive call of 'perm'. One would certainly like to throw away choice-points associated with the other recursive calls of 'perm' and backtrack directly to the untried clauses of the culprit. But as in the first example, from the limited information gathered from the SDDA, this ideal scenario can not be achieved. It could be achieved, however, by a run-time intelligent backtracking scheme.

$a(X,Y,Z) :- b(X,Y), c(Y,Z), d(X).$
 $b(X,Y) :- e(X), f(Y).$

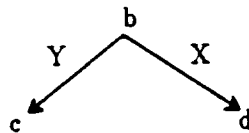
AND/OR Tree:



Legend:



$a(X,Y,Z) :- b(X,Y), c(Y,Z), d(X).$



Data Dependency Graph

Figure 3.2 A Map-coloring Problem

Given its limitations, one may wonder whether it is worthwhile to pursue semi-intelligent backtracking. It turns out that there are many programs for which semi-intelligent backtracking suffices. A very important advantage is that overhead at run-time is very small. Furthermore, semi-intelligent backtracking is based on a static analysis. The mechanism is called for only when there are advantages to so doing. A detailed comparison of run-time intelligent backtracking to semi-intelligent backtracking is shown later.

3.3. Compiling Intelligent Backtracking

In this section, the scheme to incorporate semi-intelligent backtracking into the PLM architecture [16,15] and compiler [14] is described in detail.

3.3.1. Determining Intelligent Backtrack Paths

In previous sections, *forward execution* refers to the continuation of deduction steps and *backward execution* refers to all the backtracking activities. In this section, the terms *forward state* and *backward state* are defined for *clause instances*. A clause instance is an activation of a clause. There can be several clause instances of the same clause being invoked during the execution of a program. Forward and backward states are used to distinguish three different types of backtracking.

Forward State -

The forward state of a clause instance starts from the forward execution of a body literal until a failure of another body literal of the same clause instance.

Backward State -

When a body literal fails, the clause instance is said to be in backward state. It remains in backward state until a previously invoked body literal of the same clause instance has another solution.

Since semi-intelligent backtracking is confined by a clause boundary, one needs to examine only the data dependency graph of a clause (ignoring graphs associated with proof subtrees underneath them) in order to determine the backtrack paths of its body literals. Three types of backtrack paths in a clause are distinguished, as shown below, in order to take full advantages of the data dependency analysis.

Type I -

Backtracking between body literals that occurs at the end of a forward state of a clause instance. It is discussed in section 3.3.1.1.

Type II -

Backtracking between body literals that occurs during a backward state of a clause instance and *before* its first successful exit. It is discussed in section 3.3.1.2.

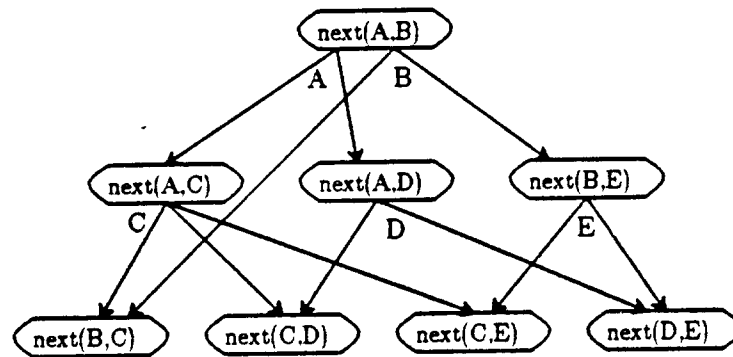
Type III -

Backtracking between body literals that occurs during a backward state of a clause instance and *after* its first successful exit. It is discussed in section 3.3.1.3.

For each type of backtracking, the *backtrack literal* of a literal can be determined from the data dependency graph.

3.3.1.1. Type I Backtrack Path

The data dependency graph for the 'map' clause in the map-coloring problem is shown again in Figure 3.3. In forward execution, if $\text{next}(A,D)$ is called, after exiting from $\text{next}(A,C)$, and fails, then it should backtrack directly to $\text{next}(A,B)$ without going through $\text{next}(A,C)$. This is because failure of $\text{next}(A,C)$ is due only to the binding of variable A generated by $\text{next}(A,B)$. In general, for this type of backtracking, the literal should backtrack to its closest predecessor node in the data dependency graph. The backtrack path for the type I backtracking in the map-coloring problem is shown in Figure 3.4 (a).



`map(A,B,C,D,E) :-`
 `next(A,B), next(A,C), next(A,D), next(B,C),`
 `next(C,D), next(B,E), next(C,E), next(D,E).`

Figure 3.3 A Data Dependency Graph

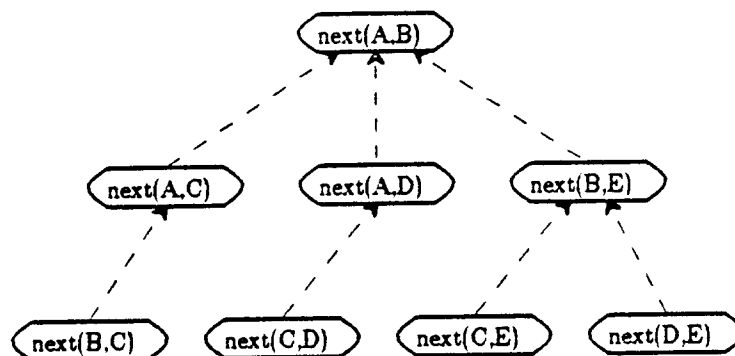


Figure 3.4 (a) Type I Backtrack Paths

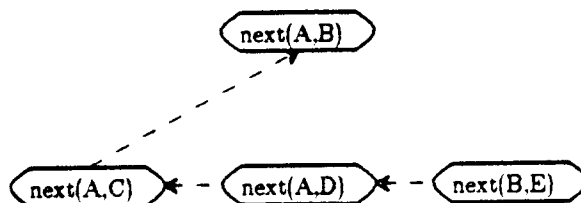


Figure 3.4 (b) Type II Backtrack Paths

3.3.1.2. Type II Backtrack Path

In forward execution, two literals on the same layer of the data dependency graph are independent. However, during backtracking, this is not the case. For example, in the data dependency graph in Figure 3.3, $\text{next}(A,C)$ is the "generator" of variable C and $\text{next}(B,E)$ is the "generator" of variable E; they are independent in forward execution. Now, let us assume $\text{next}(B,E)$ fails to respond to backtracking from $\text{next}(C,E)$. Should $\text{next}(B,E)$ backtrack intelligently to $\text{next}(A,B)$, the generator of variable B? or to $\text{next}(A,C)$, the generator of variable C? or some other literal? In this case, $\text{next}(B,E)$ should backtrack to $\text{next}(A,D)$. It does seem a bit strange that $\text{next}(B,E)$ cannot backtrack intelligently to either $\text{next}(A,C)$ or $\text{next}(A,B)$. This is because the ultimate failure of $\text{next}(B,E)$ may contribute to both $\text{next}(C,E)$ and $\text{next}(D,E)$ which reject bindings on variable E generated by $\text{next}(B,E)$. To backtrack to $\text{next}(A,D)$, there is a chance that one instance of E, when combined with a new instance of D, becomes acceptable by $\text{next}(D,E)$. In the proposed scheme there is no attempt to analyze the cause of failure, nor to remember the history of backtracking. A failure of a literal is assumed to be due to all its input arguments.

The *back-from set* of a literal determines the type II backtrack path. It is defined recursively as follows:

- (a) For each leaf literal in a data dependency graph, the back-from set is the empty set.
- (b) For each non-leaf literal g_i , the back-from set is the union of the succeeding (or right-hand side) literals whose backtrack literal (of type I for a leaf literal or of type II for a non-leaf literal) is g_i and their back-from sets.
- (c) For each non-leaf literal g_j , the backtrack literal of type II is the closest preceding (or left-hand side) literal g_k such that at least one literal in the back-from set of g_j is reachable² from g_k . If no such g_k exists, then the backtrack literal of type II of g_j is the calling literal.

Basically, the type II backtrack literal of a literal g_i is the closest literal which lies in any of the possible intelligent backtracking paths that go through g_i . From the above definition, the back-from set of a literal g_i can be seen to consist of succeeding literals whose failure may contribute (by type I or type II backtracking) to the backtracking to g_i . If g_i does not have any more untried candidate clauses, then it will backtrack to a literal g_k which still has a choice-point and can reach at least one of the literals in the back-from set of g_i (thus have a chance to solve the failed subgoal).

An algorithm which determines the back-from set and the backtrack literal of type II for each literal in the clause is shown below:

[Algorithm I]

Let G_1, G_2, \dots, G_n be the ordered sequence of literals in a clause. This sequence is partitioned into m layers according to data dependencies. Let S_i denote the back-from set of G_i .

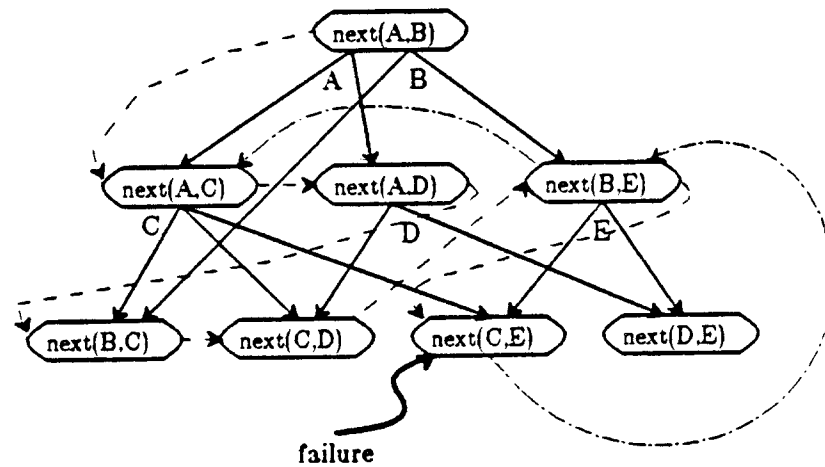
- (1) For each literal G_i in the bottom layer, set $S_i = \emptyset$. The backtrack literal (of type I) of G_i is its closest predecessor.
- (2) For layer $n = m-1$ down to 1: Let n_1, \dots, n_l be the indexes of literals in layer n .
For $j = n_l$ down to n_1 :
 - (2a) Let $A_j = \{x \mid G_j \text{ is the backtrack literal (of type I for leaf literals or of type II for non-leaf literals) of } x\}$. Then the back-from set of G_j is $S_j = \{x \mid x \in A_j \text{ or } G_i \in A_j \text{ and } x \in S_i\}$.
 - (2b) Find the largest index k , $0 < k < j$, such that at least one of the literals in S_j is reachable from G_k . If there is no such k then the backtrack literal of G_j is the calling literal; otherwise the backtrack literal of G_j is G_k .

Let R_i be the reachable set of G_i . Note that if $R_i \cap S_{i+1} = \emptyset$ then backtracking to G_i does not help in finding successful instances for literals in S_{i+1} . In naive backtracking, backtracking to G_i in this case amounts to a redundant step. In Algorithm I, these redundant steps are avoided; otherwise, nothing has been changed. For the map-coloring problem, the backtrack

²A literal g_i is reachable from literal g_j if there is a directed path from g_j to g_i in the data dependency graph.

paths for the type II backtracking are shown in Figure 3.4 (b). Note that all the literals at the bottom layer do not have backtrack literals of type II.

A type II backtrack literal is determined by the worst-case consideration. This is illustrated in Figure 3.5. In this figure, it is assumed that forward execution has advanced to literal #7 ($\text{next}(C,E)$), and literal #8 ($\text{next}(D,E)$) has not been called after the activation of this clause instance. If literal #7 fails, it should backtrack to literal #6 ($\text{next}(B,E)$) by type I backtracking. Now, if $\text{next}(B,E)$ can not yield another successful instance, it can very well backtrack to literal #2 (that is, $\text{next}(A,C)$). This is because that literal #8 has not been called yet, so the set of variable bindings generated by literal #6 are rejected only by literal #7. In this case, literal #6 does not need to backtrack to literal #3. On the other hand, if literal #8 has been invoked before, then it is possible that a subset of variable bindings generated by literal #6 are rejected by literal #8 because of the binding on variable D generated by literal #3. In that case, a retry on literal #3 to generate another binding for



Legend:

- > Data Dependency Arc
- - - - -> Forward Execution
(up to $\text{next}(C,E)$)
- - - - -> Ideal Backtrack Path

Figure 3.5

variable D may solve the problem.

Now, even if literal #8 has been invoked before, if the system keeps track of the history of backtracking (e.g. in run-time intelligent backtracking) and finds that all the binding generated by literal #6 are rejected by literal #7, then literal #6 can still backtrack directly to literal #2.

It is felt that the payoff would not justify the trouble to either compile a separate list of type II backtrack paths for each execution stage of a clause or to keep track of the history of backtracking. So, only the worst-case type II backtracking paths (that is, it is assumed that all the literals in a clause have been visited) are compiled.

3.3.1.3. Type III Backtrack Path

When a clause instance is backtracked into after its first successful exit, the static data dependency graph does not provide enough information to achieve intelligent backtracking. This is illustrated by the following example:

```

r(W,Z) :- u(W,Z), t(W,Z).
u(X,Y) :- v(X), w(Y).
u(X,Y) :- . . .

```

Assume the analyzer has deduced that, at the entry of the second clause shown above, X and Y are independent variables, v is the generator of variable X, and w is the generator of variable Y. The data dependency graph of the second clause is shown in Figure 3.6. At runtime, after the first successful exit of u(W,Z), t(W,Z) is executed. If t(W,Z) fails and backtracks to w(Y), and w(Y) fails to generate any other solution, should w(Y) backtrack to the calling literal u(W,Z) to try alternative candidate clause or backtrack to v(X) which seems to have nothing to do with w(Y)? Clearly in this case it is necessary to backtrack to v(X) because the failure of t(W,Z) is presumably due to both input arguments. (Recall that in this

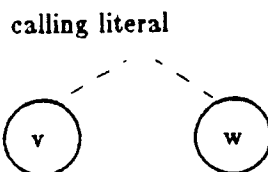


Figure 3.6

scheme, no attempt has been made to determine the cause of failure. A failure of a literal is assumed to be due to *all* its input arguments.) It is possible to generalize intelligent backtracking of type II to type III if, in algorithm I, all the possible extensions of the current proof tree are analyzed for backward independence. This global analysis can be seen to be very expensive. Since type II backtracking is based on the worst-case analysis, it is conjectured that not many intelligent backtracking paths would result from global analysis.

In the current scheme, type III backtracking is treated as naive backtracking, i.e. backtracking sequentially through all choice points.

3.3.2. Architectural Support and Code Generation for Intelligent Backtracking

In this section, the way to incorporate semi-intelligent backtracking into a Prolog machine (PLM@Berkeley) [15] and its compiler [14] is described.

3.3.2.1. The Berkeley Prolog Machine (PLM)

The PLM is a special purpose Prolog machine based on an extension of Warren's abstract Prolog instruction set [39]. It has a set of special registers and specially designed data paths for efficient execution of compiled Prolog programs.

The PLM uses four memory areas. Three of the memory areas are used as stacks to store data. The fourth is the code space.

(1) Heap -

The heap is used to store all structures created during execution of the program.

(2) Stack -

The stack contains two kinds of objects:

(a) Environments -

An environment is used to store *permanent* variables which cannot be stored in the temporary registers of the machine because these variables must survive across calls. It also contains some other information used to continue execution when the clause is completed and to aid the implementation of the cut operator. An environment is shown in Figure 3.7 (a). The E, B, CP, and N fields are copies of registers E, B, CP, and N (see the Register Description below) when the environment is created.

(b) Choice-Points -

A choice-point is shown in Figure 3.7. It contains copies of registers TR, H, A₁-A₈, CP, E, B (see the Register Description below) and a pointer (BP) to the next candidate clause of the procedure. A choice-point is created so that the correct machine state can be restored upon backtracking.

(3) Trail -

The trail contains pointers to all variables which must explicitly be reset to unbound upon backtracking.

(4) Code Space -

The code space contains all executable code of the running Prolog program.

The state of the PLM at any instant is defined by the following registers:

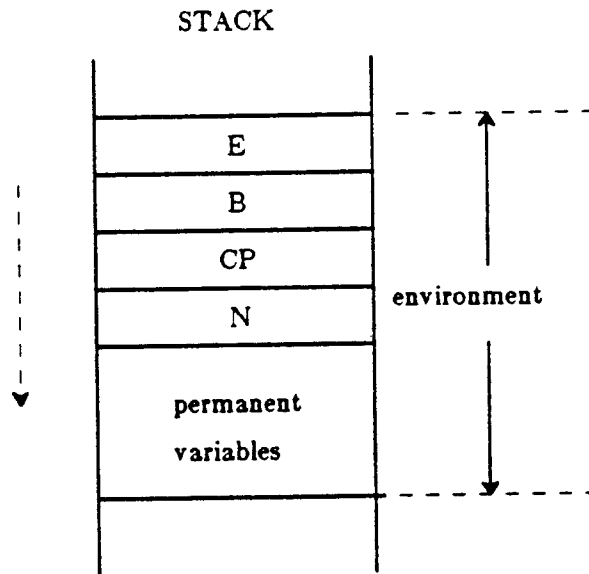


Figure 3.7 (a) An Environment

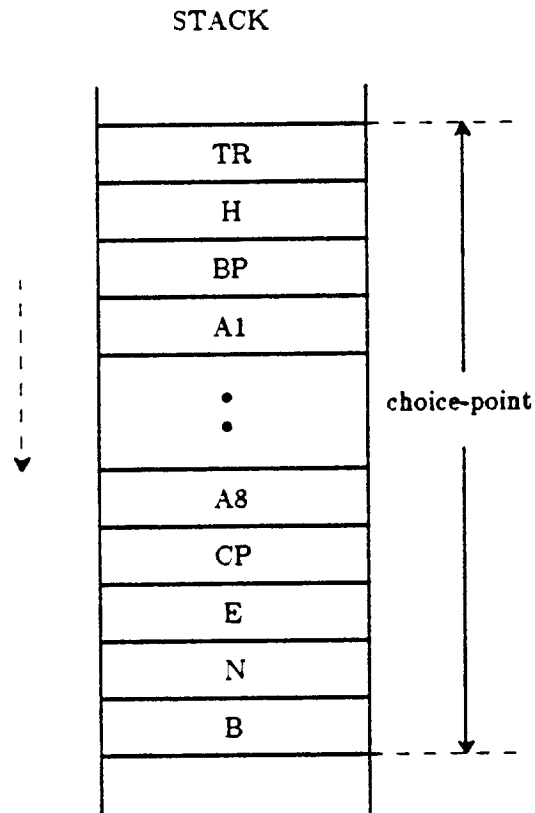


Figure 3.7 (b) A Choice-Point
(Continued)

| Register | Description |
|--------------------|--|
| P | program counter (to the code space) |
| CP | continuation pointer (to the code space) |
| E | last environment (to the stack) |
| B | last choice-point (to the stack) |
| N | size of last environment |
| H | top of the heap |
| HB | heap backtrack pointer (to the heap) |
| S | structure pointer (to the heap) |
| TR | top of the trail |
| PDL | top of the PDL (for unification between structures or lists) |
| AX _{1..3} | argument and temporary registers |

PLM instructions are categorized as follows:

- (1) unification -
Unification instructions include *get* instructions and *unify* instructions. These are the instructions that generally bind variables.
- (2) procedure call -
This category includes *put* and *unify* instructions for setting up arguments, *call* and *execute* instructions for invoking a procedure, *proceed* for a call return. *allocate* and *deallocate* instructions for allocation and deallocation of call frames (environments).
- (3) choice-point manipulation -
A set of instructions are used to manipulate choice-points. They are *try* instructions for creating choice-points, *retry* instructions for updating next untried clauses, and *trust* and *cut* instructions for throwing away choice-points. A few *switch_on_term* (clause indexing) instructions are also available for pre-selecting qualified candidate clauses.
- (4) built-in functions -
There are several built-in functions, e.g. side-effect predicate and arithmetic operators.

3.3.2.2. Implementing Intelligent Backtracking in PLM

The new data structures necessary to support semi-intelligent backtracking are shown in Figure 3.8. At compile time, a backtrack table (BT_TABLE) is constructed, as part of the compiled code, for each clause which can take advantage of semi-intelligent backtracking. At run time, a choice-point table (CP_TABLE) is maintained for each such clause. Each entry in the choice-point table is a pointer to the last choice point, if any, of the corresponding literal. Entries of BT_TABLE in Figure 8 correspond to the *color/5* clause of the map-coloring problem. No entries are provided for the first literal because it should always backtrack to the parent's choice-point.

A few new instructions are designed to support the intelligent backtracking. There are described below.

- (1) *i_allocate n, label* -

This is a new *allocate* instruction used to create an environment for a clause that can take advantage of intelligent backtracking. In this instruction, *n* is the number of literals in the clause, and *label* is the label of the BT_TABLE. When *i_allocate n, label* is executed, it creates an environment on the stack as well as a choice-point table (with *n* entries) on the heap. Pointers to the choice-point table and backtrack table are stored in the environment as shown in Figure 3.9 (a). The field LB is used to store a pointer to the last choice-point before the execution of a procedure call so that it can be determined whether this procedure call has a choice-point at the exit. It is needed for the *enter* instruction described below. A one bit flip-flop (E_FF) is set by the *i_allocate* instruction to denote that the current environment has a different structure (Figure 3.9 (a)) from an ordinary environment (Figure 3.7 (a)). It is part of the machine state. It is copied onto the stack along with the E register (in which only bits 26⁰ contain the address of an environment) (see Figure 3.9 (a)) when a call frame or a choice-point is created. It is restored when the E register is restored. Another one-bit flag (EXIT_FLAG) is also stored in the modified call frame. It is used to distinguish backtracking of type III from type II. It is set permanently by the *deallocate* instruction.

| literal# | type I | type II |
|----------|--------|---------|
| 1 | 0 | 0 |
| 2 | 1 | 1 |
| 3 | 1 | 2 |
| 4 | 2 | |
| 5 | 3 | |
| 6 | 1 | 3 |
| 7 | 6 | |
| 8 | 6 | |

BT_TABLE

| literal# | pointers to choice-points |
|----------|---------------------------|
| 1 | |
| 2 | |
| • | • |
| • | • |
| n | |

CP_TABLE

Figure 3.8 Backtrack Table (BT_TABLE) and Choice-Point Table (CP_TABLE)

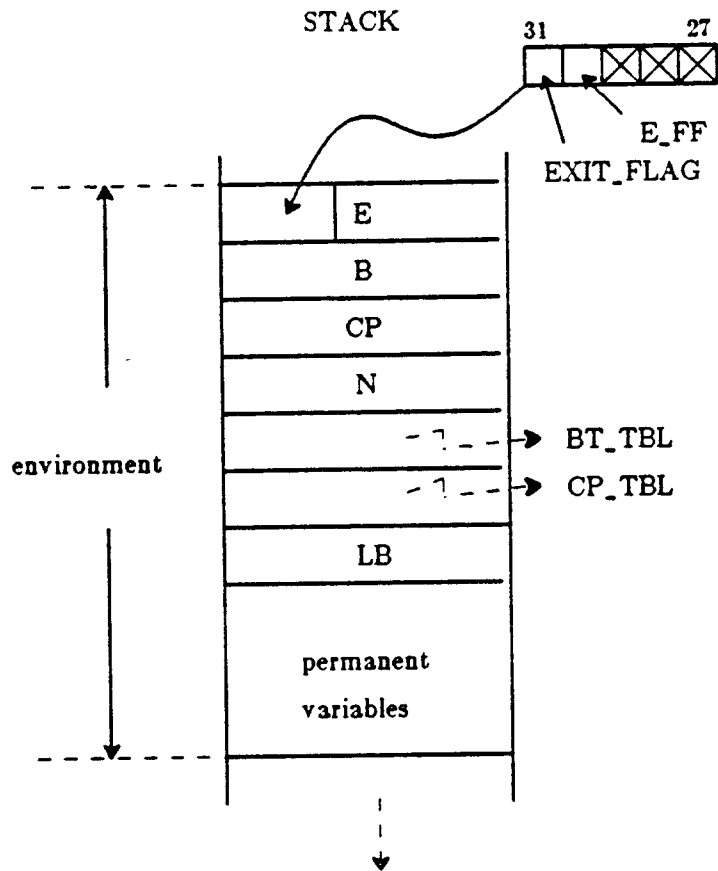


Figure 3.9 (a) A Modified Environment Frame

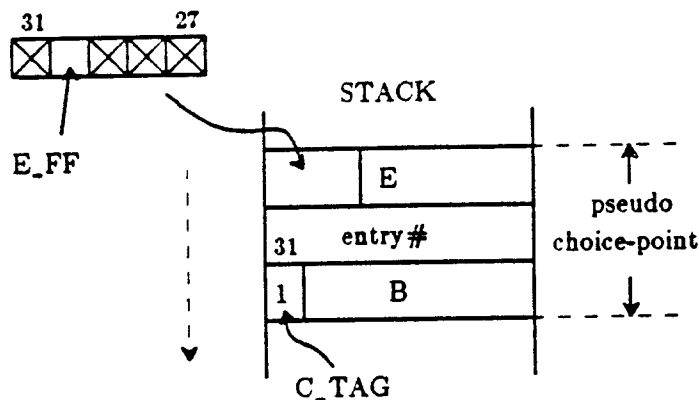


Figure 3.9 (b) A Pseudo Choice-Point
(Continued)

(2) *make m* -

A *make* instruction is used to create a *pseudo* choice-point (Figure 3.9 (b)) on the stack. It is inserted right before a procedure call to set up intelligent backtracking when the procedure call fails. A *pseudo* choice-point does not contain previous machine state as in a normal choice-point. It contains only information for the purpose of intelligent backtracking. It includes copies of registers B (in order to access the previous choice-point, *real* or *pseudo*), E (to access EXIT_FLAG and pointers to CP_TABLE and BT_TABLE), and a value *m* (the literal number of the following procedure call). At the end of the *make m* instruction, register B points to the newly created pseudo choice-point, and the *m*th entry in the CP_TABLE is initialized as UNKNOWN. A one bit tag (CTAG) is used to distinguish a pseudo choice-point (CTAG = 1) from a normal choice-point (CTAG = 0).

(3) *i_call name/arity, n* -

An *i_call* instruction is almost the same as a *call* instruction except that it also copies the value of the B register into the LB field in the environment. With the pointer to the most recently activated choice-point available in the LB field, it can be determined whether or not there are choice-points under this procedure call.

(4) *enter m* -

An *enter m* instruction is inserted right after an *i_call* in order to record the last choice-point under the procedure call into the *m*th entry of the CP_TABLE. If there are no choice-points generated by the procedure call or its children when *enter m* is executed, the *m*th entry in the CP_TABLE is tagged as NONE. It can be determined by comparing the closest *real* choice-point (traced from the B register) with the LB value in the environment.

(5) *i_cut m* -

When *cut* is a literal of a clause which can take advantage of semi-intelligent backtracking, it is replaced by *i_cut m*. An *i_cut m* instruction is similar to *cut*,

except that the pointer which is supposed to load into the B register is also recorded in the m th entry of the CP_TABLE.

In naive backtracking, the machine state kept in the last choice point, pointed to by the B register, is restored when a failure occurs. With semi-intelligent backtracking, the operations of failure handling are changed to the following:

- (1) If register B points to a real choice point (CTAG=0), then go to (3). Otherwise, continue.
- (2) Access entry# from the pseudo choice-point and pointers to CP_TABLE and BT_TABLE from the environment. Examine CP_TABLE[entry#]. Two possible cases follows:
 - (a) CP_TABLE[entry#] is tagged as UNKNOWN. It is backtracking of type I. Look up the "TYPE I" field of BT_TABLE[entry#] to get the backtrack literal.
 - (b) Otherwise, get EXIT_FLAG from the modified environment.
 - (i) If EXIT_FLAG = 1 (i.e. TYPE III backtracking), retrieve the pointer to the previous choice-point from the B field of the pseudo choice-point and keep tracing pointers until a real choice-point is found; then go to (3).
 - (ii) If EXIT_FLAG = 0 (i.e. TYPE II backtracking), look up the "TYPE II" field of BT_TABLE[entry#] to get the backtrack literal.

In TYPE I and Type II backtracking, if the backtrack literal has a NONE entry in the CP_TABLE, then keep tracing backtrack literal chains of type II until either a backtrack literal with a non-NONE entry is found or the calling literal (literal #0) is met. In the first case, the non-NONE entry is loaded into the B register. In the latter case, a copy of the B register, stored in the environment, is restored. If it points to a pseudo choice-point, then keep tracking B pointers until a real choice-point is reached.

- (3) Restore machine state from the choice-point.

In the proposed modifications, only one flip-flop (E_FF) has to be added into the PLM hardware. There are five new instructions to be implemented in microcode. The code size of a typical program increases very little (a small backtrack table and at most two more instructions per procedure call for clauses which can take advantage of semi-intelligent backtracking). At run time, a little bit more memory space is required to hold pseudo choice-points (3 32-bit words per pseudo choice point) and environments (3 32-bit additional words for such a clause). The run-time overhead for normal backtracking only involves checking the CTAG bit stored in the choice-point. To backtrack semi-intelligently involves more work but the overhead is still small. It can be seen that this is indeed a feasible scheme and can be supported in a Prolog system.

3.3.2.3. Code Generation

There are three possible forms of compiled code for a procedure call in a clause which can take advantage of the semi-intelligent backtracking:

- (1) For a literal which can not backtrack intelligently and will not be backtracked into intelligently from the other sibling literals, it is translated as a regular procedure call:

call name/arity,n

- (2) For a literal which can not backtrack intelligently but can be backtracked into intelligently from the other sibling literals, it is translated into the following code:

i_call name/arity,n
enter m

- (3) For a literal which can backtrack intelligently (whether or not it can be backtracked into intelligently), it is translated into the following code:

make m
i_call name/arity,n
enter m

The compiled code for the 'color' clause (Figure 3.2) is shown in Figure 3.10.

The code can be further optimized by considering the determinism of each literal. A simple criterion for detecting determinism of a literal is that all its arguments are bound to ground term when it is called. (Of course, there may be a lot of deterministic literals which can not be detected by this criterion.) This information is derivable from the SDDA. For example, in the map-coloring problem, the analyzer can detect that all leaf-literals are deterministic. So, there are no intelligent backtracking paths for literal #5 & #8 (because type I backtracking from literal #5 to literal #3 goes through literal #4 which is deterministic, same for literal #8). They can be translated into normal procedure calls.

The above reasoning is correct if there are no choice-points created for a deterministic literal at run-time. However, because of non-perfect hashing, it is possible that some useless choice-points are created for a deterministic literal. To backtrack through these useless choice-points may be time-consuming, e.g. in data-base application. One may want to bypass these useless choice-points. It can be easily done by generating the following code:

call name/arity,n
make m

By putting a *make* instruction after the procedure call, it sets up a TYPE I backtracking trap. With the proper entry in the backtrack table, the choice-points of this procedure call can be skipped.

The above two paragraphs describe two different ways that a deterministic (known at compile time) literal can be handled. One may be more applicable than the other in different applications. It also shows the versatility of the enhanced instruction set.

There are two ways to handle the last procedure call of a clause. The last procedure call can have an entry in the choice-point table. In this way, the last procedure call will be treated as the other procedure calls, and it can take advantage of semi-intelligent backtracking. It is translated into the following code:

```

i_allocate 8, BT_TBL
{ get instructions } /* unify with head literal */
{ put instructions } /* set up arguments for next(A,B) */
i_call next/2,5
enter 1
{ put instructions } /* set up arguments for next(A,C) */
i_call next/2,5
enter 2
{ put instructions } /* set up arguments for next(A,D) */
make 3
i_call next/2,4
enter 3
{ put instructions } /* set up arguments for next(B,C) */
make 4
i_call next/2,4
enter 4
{ put instructions } /* set up arguments for next(C,D) */
make 5
i_call next/2,4
enter 5
{ put instructions } /* set up arguments for next(B,E) */
make 6
i_call next/2,3
enter 6
{ put instructions } /* set up arguments for next(C,E) */
call next/2,2
{ put instructions } /* set up arguments for next(D,E) */
make 8
i_call next/2,0
enter 8
deallocate
proceed
BT_TBL: /* Backtrack Table */
1 1
1 2
2
3
1 3
6
6

```

Figure 3.10

```

make n
call foo/2, 0
enter n
deallocate
proceed

```

It can not, however, achieve last literal (tail recursion) optimization. The alternative is not to take advantage of the semi-intelligent backtracking for the last procedure call, i.e. backtracking from the last procedure call is handled as the type III (naive) backtracking. In this way, the last procedure call can stay in its original form (with last literal optimization):

```

deallocate
execute foo/2

```

In order to recover stack space for determinate execution up to the last literal, *deallocate* instruction have to be modified to trace through and throw away pseudo choice-points whose corresponding entries in the CP_TABLE are tagged as NONE until a real choice-point is met.

3.3.2.4. Simulations

The proposed scheme is simulated by modifying our PLM simulator [40]. The simulator generates run-time statistics which show the frequency of instruction execution and other important information. This information includes invocations of built-in micro routines, such as failures (backtracks), and dereferences, as well as memory access. The simulated run-time statistics for the map-coloring problem (in Figure 2.1) with and without the semi-intelligent backtracking scheme are shown in Figure 3.11 (a) & (b). Instructions with zero execution frequency are not shown in the figure. It can be seen that for this simple example the performance gain is about 30%, and the memory space used is about 9% more. The trade-off is definitely worthwhile. Also shown in Fig 3.11(a) are the number of intelligent backtrackings that occur and the total number of real choice-points that have been skipped by intelligent backtracking hops. In the map-coloring problem, intelligent backtracking of type I occurs once (when next(B,C) fails in forward execution), and it skips over two real choice-points created by next(A,D) and its children. Although it only hops over two real choice-points, it actually saves all the work which may involve many more choice-points that would be otherwise spawned by them.

Although the count of each machine instruction is not weighted by the machine cycles required to execute it, Figure 3.11 shows that almost all instruction executions are reduced by the same percentage by the intelligent backtracking. This is what one might have expected.

3.3.3. Applications

Intelligent backtracking in Prolog can be quite important for a lot of applications, e.g. deductive databases, CAD, and other generic searching problems which have a flavor of theorem proving. For example, in a deductive knowledge-based system, a user may be interested in knowing the first few possible solutions, which satisfy a probability threshold set by the user, of a query. This can not be solved efficiently by conventional database techniques which retrieve all the possible solutions. The depth-first search strategy with built-in backtracking allows us to get the first, or the first few, answers with a modest set of resources. It seems to a good idea. It is not, however, the whole story if it always backtracks naively when a failure occurs. (Some people argue that the depth-first search strategy in

| # Simulator Data Run | Count | % |
|----------------------|-------|-------|
| proceed | 21 | 7.95 |
| execute | 12 | 4.55 |
| call | 14 | 5.30 |
| i_call | 9 | 3.41 |
| allocate | 10 | 3.79 |
| i_allocate | 1 | 0.38 |
| deallocate | 11 | 4.17 |
| get_variable | 8 | 3.03 |
| get_constant | 50 | 18.94 |
| put_variable | 5 | 1.89 |
| put_value | 26 | 9.85 |
| try_me_else | 11 | 4.17 |
| retry_me_else | 9 | 3.41 |
| trust_me_else | 10 | 3.79 |
| try | 12 | 4.55 |
| retry | 10 | 3.79 |
| switch_on_term | 13 | 4.92 |
| switch_on_constant | 16 | 6.06 |
| make | 7 | 2.65 |
| enter | 9 | 3.41 |
| TOTAL | 264 | |
| failures | 29 | |
| type I | 1 | |
| skip(type I) | 2 | |
| type II | 0 | |
| skip(type II) | 0 | |
| unifications | 50 | |
| unify routine | 50 | |
| bindings | 7 | |
| escapes | 0 | |
| memory reads | 606 | |
| memory writes | 483 | |
| dereferences | 79 | |
| binding trails | 7 | |
| maximum trail | 6 | |
| maximum stack | 238 | |
| maximum heap | 14 | |
| maximum PDL | 0 | |

Figure 3.11 (a) Run-time statistics with semi-intelligent backtracking for the map-coloring problem (Figure 2.1).

| # Simulator Data Run | Count | % |
|----------------------|-------|-------|
| proceed | 24 | 6.58 |
| execute | 18 | 4.93 |
| call | 29 | 7.95 |
| allocate | 13 | 3.56 |
| deallocate | 13 | 3.56 |
| get_variable | 11 | 3.01 |
| get_constant | 90 | 24.66 |
| put_variable | 5 | 1.37 |
| put_value | 36 | 9.86 |
| try_me_else | 14 | 3.84 |
| retry_me_else | 18 | 4.93 |
| trust_me_else | 19 | 5.21 |
| try | 16 | 4.38 |
| retry | 15 | 4.11 |
| switch_on_term | 18 | 4.93 |
| switch_on_constant | 26 | 7.12 |
| TOTAL | 365 | |
| failures | 52 | |
| unifications | 90 | |
| unify routine | 90 | |
| bindings | 15 | |
| escapes | 0 | |
| memory reads | 931 | |
| memory writes | 585 | |
| dereferences | 134 | |
| binding trails | 15 | |
| maximum trail | 6 | |
| maximum stack | 226 | |
| maximum heap | 6 | |
| maximum PDL | 0 | |

Figure 11 (b) Run-time statistics with naive backtracking
for the map-coloring problem (Figure 2.1).
(Continued)

Prolog is too strict to be useful. This argument is a bit misleading, because it is not hard to write a meta-interpreter on top of user programs to achieve adaptive searches. What is really bad about current Prolog interpreters or machine architectures is the naive backtracking which is built into the DFS (depth-first search). More discussion on this appears in the last chapter.) In the following, it is shown that the proposed scheme can remove the inefficiency of

execution, caused by the naive backtracking, for some applications.

I. Adaptive Quadrature Integration:

Prolog is well suited for writing adaptive algorithms. An adaptive algorithm can be written in Prolog in the following way:

- (1) Each clause in a procedure describes an algorithm. Clauses are ordered such that a more refined and time-consuming algorithm always follows a cheap-and-fast one.
- (2) If a clause (algorithm) fails (e.g. it does not produce acceptable results), then the next clause (more expensive algorithm) in order will be tried. Time-consuming algorithms do not have to be tried until necessary.

One of the advantage of the quadrature integration (shown in Figure 3.12) is that the error bound can be estimated. In this example, arithmetic functions are expressed, for convenience, with arithmetic symbols and are used directly in argument expressions. Two clauses are used to describe the adaptive quadrature integration method. The first clause will return the value of the quadrature function as the result if the estimated error bound is less than the imposed error bound. The second clause partitions the integration into two independent adaptive quadrature integrations each with an error bound which is proportional to its share of integration. The partitioning is done by the subgoal 'select' which may simply chop the area in half or chop it according to the behavior of the function. Both 'integration' subgoals have to be satisfied. Although it is not shown, the 'select' procedure should prevent indefinite partitioning and cause a failure when it detects that the error bound can not be satisfied with a reasonable amount of computations. If the second 'integration' fails, by type I backtracking, it should backtrack directly to the 'select' procedure and try another way to partition. In naive backtracking, to backtrack to the first 'integration' and try to get a more accurate result if it does not solve the failure of the second 'integration'.

II. Automated Circuit Design Problem

A Prolog program has been written for automated circuit design. This program can generate circuit configurations for a given truth table. The most crucial step in the program is, given an output function, to derive a set of possible input functions and to implement the input functions one by one. It is written in Prolog as follows:

```
circuit([0,1,0,1,0,1,...],i1)./* boundary conditions */
.
.
.
circuit(Output,[Config1,Config2]):-
    derive(Output,Input1,Input2),
    circuit(Input1,Config1),
    circuit(Input2,Config2).
```

In the above, it is assumed that the basic logic component is a two-input gate. Its function is hidden by the predicate "derive". It is easy to see how unfortunate it would be if a pair of possible input functions, (Input1,Input2), generated by the predicate "derive" could be implemented by the first "circuit" predicates but not the second. Since the first "circuit" predicates can have many possible implementations, it will take quite a while to enumerate all of them before useful business (backtracking to 'derive') is accomplished. Although this problem can be solved by putting a *cut* at the end of the recursive clause, this is still too restrictive. The user may want to see several alternate

. Quadrature Integration

$$\left| \int_a^b f(x) dx - \text{quad}(f, a, b) \right| \leq \text{err}(f, a, b)$$

$$\text{trap}(f, a, b) = (f(a) + f(b)) * \frac{b-a}{2}$$

$$\text{quad}(f, a, b) = \text{trap}(f, a, \frac{a+b}{2}) + \text{trap}(f, \frac{a+b}{2}, b)$$

$$\text{err}(f, a, b) = \left| \text{quad}(f, a, b) - \text{trap}(f, a, b) \right|$$

query:

```
:- integration(a,b,f,Ans,ε,Errbd).
/* ε: given error bound */
/* Errbd: estimated error bound */
```

program:

```
integration(S,T,Fct,Y,Err1,Err2) :-
    Y = quad(Fct,S,T),
    Err2 = |Y-trap(Fct,S,T)|
    Err2 ≤ Err1.
integration(S,T,Fct,Y1+Y2,Err1,Err2+Err3) :-
    select( ...,U),
    integration(S,U,Fct,Y1,Err1 *  $\frac{U-S}{T-S}$ ,Err2),
    integration(U,T,Fct,Y2,Err1 *  $\frac{T-U}{T-S}$ ,Err3).
```

Figure 3.12 Adaptive Quadrature Integration

designs instead of just the first one.

The automatic circuit design program is listed in Appendix B. The 'ngate' procedure is used to implement the 2-input NAND function. To take advantage of the semi-intelligent backtracking, the 'ngate' procedure call is compiled into the following code:

```
i_call ngate/3,5
enter 1
```

The second 't' procedure call is compiled into the following code:

```
make 2
i_call t/3,3
enter 2
```

The run-time statistics of running this query with and without semi-intelligent backtracking are shown in Figure 3.13. With the proposed scheme, the performance gain is about four times!!!!

III Data Base Query

Consider the following query:

Q: List names of a couple in this community who are both employees of the company PHONY and share no common hobbies. This query can be written in Prolog as follows:

```
?- couple(Husband,Wife),
   employee_hobby(Husband,HobbyA),
   employee_hobby(Wife,HobbyB),
   neq(HobbyA,HobbyB).
```

In the above, the couple relation contains all couples in the community, the employee_hobby relation contains hobbies of all employees in company PHONY. Now suppose the names of a couple are retrieved from the couple relation, and suppose that the husband is an employee of the company PHONY and the wife is not. It can be seen that the first two literals can be executed successfully, but not the third. With naive backtracking, the second literal will be retried, uselessly, for several times until all the hobbies of the husband are enumerated. From the data dependency analysis, we can tell that the third literal is a successor of only the first literal. With semi-intelligent backtracking of type I, the third literal can backtrack directly to the first literal on failure.

From above examples, one can see the importance of supporting intelligent backtracking. In current Prolog compilers/interpreters, the only means to prevent exhaustive depth-first search is by *cut*. As mentioned above, *cut* is too restrictive. With the proposed scheme, sometimes it is possible to prevent redundant, exhaustive, depth-first search while maintaining the flexibility.

One may have observed that in the above examples, all the intelligent backtracking paths are of type I. In general, because type II backtracking handles only continuous backtracking between body literals of a clause. Most likely it is useful only for clauses which have many non-deterministic body literals. Furthermore, since it is based on the worst-case

| # Simulator Data | Run Count | % |
|------------------|-----------|-------|
| proceed | 166 | 1.12 |
| execute | 273 | 1.83 |
| call | 137 | 0.92 |
| i_call | 32 | 0.21 |
| allocate | 456 | 3.06 |
| i_allocate | 60 | 0.40 |
| deallocate | 84 | 0.56 |
| get_variable | 468 | 3.14 |
| get_value | 60 | 0.40 |
| get_constant | 400 | 2.69 |
| get_nil | 170 | 1.14 |
| get_structure | 390 | 2.62 |
| get_list | 2379 | 15.98 |
| put_variable | 91 | 0.61 |
| put_value | 595 | 4.00 |
| put_unsafe_value | 155 | 1.04 |
| put_constant | 486 | 3.27 |
| put_structure | 20 | 0.13 |
| put_list | 1 | 0.01 |
| unify_void | 615 | 4.13 |
| unify_variable | 804 | 5.40 |
| unify_value | 20 | 0.13 |
| unify_constant | 2172 | 14.59 |
| unify_nil | 164 | 1.10 |
| try_me_else | 143 | 0.96 |
| retry_me_else | 725 | 4.87 |
| trust_me_else | 102 | 0.69 |
| try | 239 | 1.61 |
| retry | 300 | 2.02 |
| trust | 168 | 1.13 |
| switch_on_term | 538 | 3.61 |
| unify_cdr | 1800 | 12.09 |
| escape | 546 | 3.67 |
| cut | 47 | 0.32 |
| make | 22 | 0.15 |
| enter | 56 | 0.38 |
| | | |
| TOTAL | 14884 | |
| | | |
| failures | 1295 | |
| type I | 16 | |
| skip(type I) | 35 | |
| type II | 0 | |
| skip(type II) | 0 | |

| | |
|----------------|-------|
| unifications | 3757 |
| unify routine | 3757 |
| bindings | 2378 |
| escapes | 546 |
| memory reads | 30507 |
| memory writes | 19526 |
| dereferences | 2425 |
| binding trails | 2267 |
| maximum trail | 80 |
| maximum stack | 495 |
| maximum heap | 142 |
| maximum PDL | 0 |

Figure 3.13 (a) Run-time statistics with semi-intelligent backtracking for the automatic circuit design problem.

| # Simulator Data Run | Count | % |
|----------------------|-----------|-------|
| proceed | 472 | 0.83 |
| execute | 1036 | 1.82 |
| call | 425 | 0.75 |
| allocate | 1880 | 3.31 |
| deallocate | 270 | 0.48 |
| get_variable | 1760 | 3.10 |
| get_value | 156 | 0.27 |
| get_constant | 1746 | 3.07 |
| get_nil | 634 | 1.12 |
| get_structure | 1329 | 2.34 |
| get_list | 9289 | 16.36 |
| put_variable | 235 | 0.41 |
| put_value | 2123 | 3.74 |
| put_unsafe_value | 495 | 0.87 |
| put_constant | 2046 | 3.60 |
| put_structure | 52 | 0.09 |
| put_list | 1 | 0.00 |
| unify_void | 2304 | 4.06 |
| unify_variable | 3178 | 5.60 |
| unify_value | 52 | 0.09 |
| unify_constant | 9304 | 16.39 |
| unify_nil | 704 | 1.24 |
| try_me_else | 449 | 0.79 |
| retry_me_else | 2924 | 5.15 |
| trust_me_else | 436 | 0.77 |
| try | 792 | 1.39 |
| retry | 1126 | 1.98 |
| trust | 655 | 1.15 |
| switch_on_term | 1804 | 3.18 |
| unify_cdr | 6785 | 11.95 |
| escape | 2202 | 3.88 |
| cut | 119 | 0.21 |
| TOTAL | 56783 | |
| failures | 5141 | |
| unifications | 15572 | |
| unify routine | 15572 | |
| bindings | 10033 | |
| escapes | 2202 | |
| memory reads | 116964 | |
| memory writes | 73313 | |
| dereferences | 10152 | |
| binding trails | 9618 | |
| maximum trail | 80 | |

maximum stack 484
 maximum heap 134
 maximum PDL 0

Figure 3.13 (b) Run-time statistics with naive backtracking
 for the automatic circuit design problem.
 (Continued)

consideration and is applied before the first successful exit from the clause, its occurrences are rare. Table 3.1 shows the effectiveness of type I & II backtracking for some benchmark programs. Note that for these examples, type II backtracking does no good at all. (It does not allow any real choice points to be skipped). However, it is not difficult to observe that type II backtracking can be useful for other cases. For example, in Figure 3.14, lit3 can backtrack directly to lit1 by type II backtracking (without going through choice points of lit2) after both lit4 and lit3 fails.

3.3.4. Backtrack Graph vs. Data Dependency Graph

Up to now in this chapter, it is assumed that all the backtrack paths are determined from data dependency graphs. As a matter of fact, backtrack paths are determined from *backtrack graphs* and have taken into account the *determinancy* (the word is used to describe whether or not a procedure is determinate³) of a literal. The difference between a backtrack graph and a dependency graph can be seen from the following example:

| Benchmark | | Type I | | Type II | |
|---------------------|------|-------------------|---------------------------------------|-------------------|---------------------------------------|
| | | no. of occurrence | no. of choice-points being skipped | no. of occurrence | no. of choice-points being skipped |
| exhaustive-coloring | | 27 | 18 | 0 | 0 |
| query | | 24 | 22 | 15 | 0 |
| circuit | | 16 | 35 | 0 | 0 |
| color13 | good | 4 | 3 | 0 | 0 |
| color13 | bad | 9 | 90 | 0 | 0 |

Table 3.1 A comparison between type I & II backtracking

³A calling literal is deterministic if it has at most one successful instance or fails.

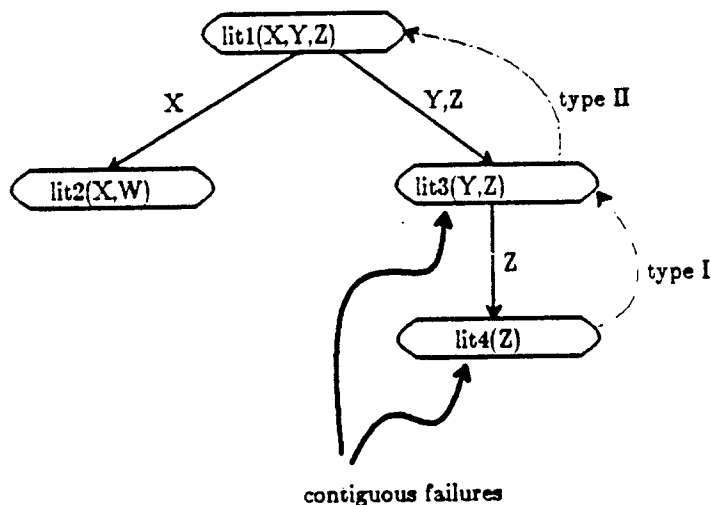


Figure 3.14 A type II backtracking

```

entry_(map(i,i,i,i,i)).
map(A,B,C,D,E) :- next(A,B), write((A,B)), det(B,C),
                 next(C,D), next(C,E), next(D,E).

```

Assume at the exit of either 'next' or 'det' both arguments are grounded, and 'det' is deterministic. The data dependency graph is shown in Figure 3.15 (a), and the backtrack graph is shown in Figure 3.15 (b). It is clear that I/O operations should be considered in determining the dependency graph. However, since an I/O operation always succeeds and is deterministic, it should be excluded from the backtrack graph which is used to determine intelligent backtracking paths.

The information about the determinacy of a literal is important for the effectiveness of the proposed intelligent backtracking scheme as can be seen from the following example.

```

a(X,Y) :- b(X), ..., c(Y), ..., d(X,Y).

```

In this example, assume at the entry of the clause, both X and Y are unbound. Assume X is bound by 'b' and Y is bound by 'c'. In the proposed scheme, if 'd' fails, it will backtrack to the closest predecessor which is 'c'. But, if 'c' is deterministic, then it would be more desirable for 'd' to backtrack directly to 'b' via type I backtracking; otherwise, the backtrack path from 'c' would be type II which is non-optimal. In the current implementation of the SDDA, the programmer can declare that a procedure is deterministic. A simple-minded algorithm could be incorporated into the analyzer to check for determinacy. But, it requires including

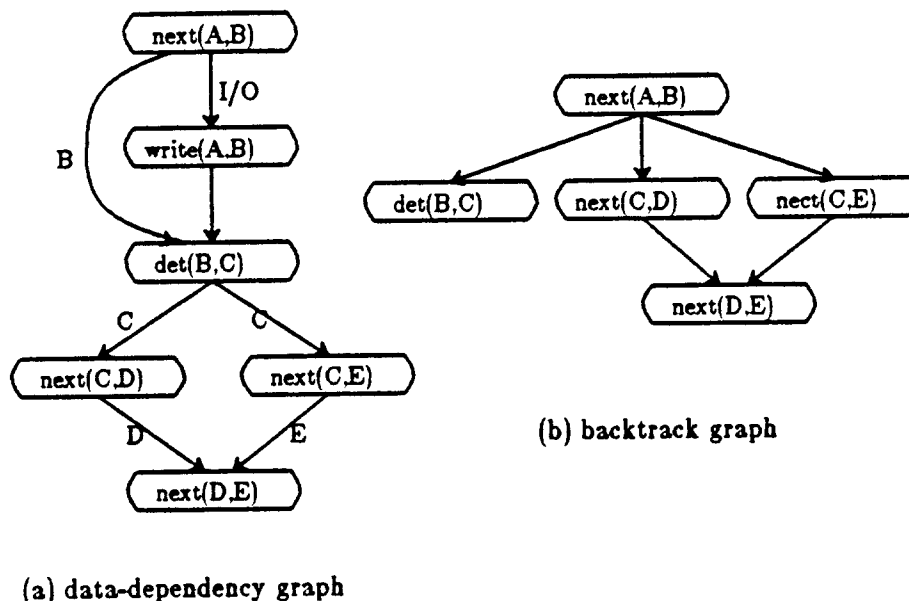


Figure 3.15 Backtrack graph vs. data dependency graph

structure matching into the analyzer to be effective.

3.4. Comparison with Other Implementations/Approaches of Intelligent Backtracking

In this section, different implementations/approaches of intelligent backtracking are examined. Advantages and disadvantages of each approach are compared.

3.4.1. Compiling Intelligent Backtracking vs. Run-Time Intelligent Backtracking

The general approach to run-time intelligent backtracking is outlined at the beginning of this chapter. Shown in Table 3.2 is a table of run-time statistics of the benchmark programs selected in the paper of Bruynooghe and Pereira [18]. Columns 2 to 4 are the run-time statistics of the benchmark programs simulated with the PLM simulator. The last column is the run-time statistics of the benchmark programs quoted directly from their paper. It shows the percentage change in performance ('-' stands for improvement, and '+' otherwise) when Bruynooghe and Pereira's (P&B's) intelligent backtracking is supported in a Prolog interpreter. Two figures are used to illustrate performance changes in the simulation of the semi-intelligent backtracking. One is the total number of PLM instructions executed, and the other is the number of failures (backtracking) that occurred. The first benchmark program is a simple data base query. The second benchmark program is the queens problem (tested for 5

| Benchmark | Naive (PLM) no. of occurrence | | Semi-Intelligent (PLM) no. of occurrence | | Semi-Int vs. Naive (PLM) % Change | | B&P's Int. vs. Naive (Interpreter) % Change | |
|-------------|----------------------------------|---------|---|---------|--------------------------------------|---------|--|--------------|
| | Total | Failure | Total | Failure | Total | Failure | | |
| query | 1493 | 289 | 1258 | 188 | -16% | -35% | -20% | |
| queens | simple | - | - | - | 0% | 0% | -36% - -77% | |
| | clever | - | - | - | - | 0% | 0% | +99% - +100% |
| map-color | bad | 2484893 | 714808 | 2272 | 371 | -99.9% | -99.9% | -99.7% |
| | good | 894 | 200 | 913 | 185 | +0.7% | -7.5% | +63% |
| binary tree | - | - | - | - | 0% | 0% | +44% | |

Table 3.2 A performance comparison between naive and intelligent backtracking

queens, ..., 8 queens). The third benchmark program is the map-coloring problem to color 13 regions. The last benchmark is a deterministic program to build an ordered binary tree.

Since B&P's intelligent backtracking involves backtracking intelligently across clause boundaries, it can improve performance for the generate-and-test (simple) algorithm used in solving the queens problem. However, by merging the generate and test parts, the resulting (clever) program has backtracking distance (in terms of choice points) which is always one, similar to naive backtracking. In this case, run-time intelligent backtracking only introduces overhead. In the map-coloring problem, with a good ordering of body literals, the effect of intelligent backtracking is not sufficient (as in the bad ordering case) to overcome the overhead. For the deterministic program which builds an ordered binary tree, there is no backtracking. So, again, intelligent backtracking only introduces overhead.

The semi-intelligent backtracking has the advantage of flexibility. This mechanism is used only when it is advantageous. For the queens problem, as well as the deterministic program which builds a binary tree, no intelligent backtracking paths⁴ are detected by the static data dependency analyzer. So, these programs are compiled into the same code as before.

The weakness of the proposed scheme, as compared with run-time intelligent backtracking, is that it can not backtrack directly across a clause boundary and it is based on a static worst-case analysis. However, since the overhead of our scheme is very small, the

⁴An intelligent backtracking path is a backtracking path which skips over literals which are not known to be deterministic at compile time.

penalties are also very small (e.g. 0.7% in the map-coloring problem with good ordering of body literals) even when the advantage of using this mechanism turns out to be limited (-7.5% in this case).

In general, the disadvantage of dynamic intelligent backtracking is high overhead. These schemes cannot judge, before run-time, whether this mechanism should be used or not.

An important observation about both forms of intelligent backtracking is that problems whose executions are order-sensitive with naive backtracking can become much less order-sensitive when equipped with (semi-)intelligent backtracking, as for example, in the map-coloring program. However, there are still some problems whose execution can not be sped up by reordering of body literals, but can be sped up by (semi-)intelligent backtracking, for example the automatic circuit design problem shown above.

3.4.2. Intelligent Backtracking Based on Annotated Programs

Another approach to achieve intelligent backtracking has been suggested by Dembinski and Maluszynski [41]. Their approach requires annotating I/O modes for all predicates in a program. It is summarized as follows:

- (1) Each argument of a predicate is declared either as an input argument or an output argument.
- (2) Based on the I/O mode declarations, data dependency graphs can be constructed at compile time. At run time, an *extended* data dependency graph is constructed for a clause instance from the compiled data dependency graph to take into account of variable bindings due to unification. From the *extended* data dependency graph, predecessors of literals and generators of variables can be determined.
- (3) A failure of a literal is assumed to be due to all its input variables. When a failure occurs, the closest predecessor is selected to backtrack. An algorithm is used to determine the *backtrack point* [41] of a literal at run-time. The *backtrack point* of a literal includes all its predecessors, and all the previous untried backtrack candidates.

It is interesting to observe how this scheme fits in between the semi-intelligent backtracking and the run-time intelligent backtracking. It is clear that this scheme still requires much more run-time overhead than the semi-intelligent backtracking, but less than the run-time intelligent backtracking. It can backtrack more intelligently than the semi-intelligent backtracking. This is because that it computes the *extended* data dependency graph at run-time (instead of using the worst-case assumption). It is not as intelligent as run-time intelligent backtracking because the faulty variable bindings are not determined in a more refined way (see (3) above). No simulation results are available to allow a more thorough comparison.

3.5. Conclusion

In this chapter, the scheme to compile intelligent backtracking based on data dependencies between literals was described. Not many modifications are required to incorporate this scheme into the PLM architecture. The overhead at run time is small. Data dependencies between literals can be determined automatically by a static data dependency analyzer, or manually indicated by programmers using pragmas. The advantages of intelligent backtracking are illustrated with examples and proved with simulation results.

In a parallel execution environment [34], the semi-intelligent backtracking as discussed in this chapter can be also applied. It is discussed in the next chapter.

CHAPTER 4

COMPILING AND-PARALLELISM FOR A PARALLEL ARCHITECTURE

4.1. Overview

In general, independent subgoals in the goal list can be executed concurrently by exploiting AND-Parallelism. AND-Parallelism has been studied by Conery [22] in his thesis. He proposed an AND/OR process model in which an AND process controls the AND-parallel execution of subgoals in a clause and an OR process works in an eager mode for solving each subgoal and generating all the possible solutions. Processes are communicated via messages. In his scheme, dependency checks among literals are done at run-time which can be quite expensive. Since data dependency graph can be generated at compile time as described in Chapter 2, there is no run-time overhead for dependency checks. In this Chapter, a scheme to compile Prolog programs for AND-Parallel execution is described. The scheme is designed for the Aquarius system [42] which is a tightly-coupled, heterogeneous multiprocessor system specialized for symbolic/numeric computation.

4.2. Why Exploit AND-Parallelism ?

AND-Parallelism exists for both deterministic and non-deterministic Prolog programs. In AND-parallel execution, independent subgoals can be executed in parallel. Resources are not wasted, in computing results which are not requested, as is possible in OR-Parallel execution.

There are several advantages in exploiting AND-Parallelism:

- (1) Parallel execution of independent subgoals.
- (2) Cooperative failure detection - Variable bindings generated by a subgoal are evaluated by all the successor subgoals. Variable bindings are rejected as soon as one of the successor subgoals fails.
- (3) Semi-intelligent backtracking.
- (4) Parallel backtracking.

The first two advantages are easy to understand. Semi-intelligent backtracking in the parallel execution environment and parallel backtracking are discussed later.

4.2.1. An Example

In this section, an example is used to describe the advantages of and issues concerned in exploiting AND-Parallelism.

Example 4.1.

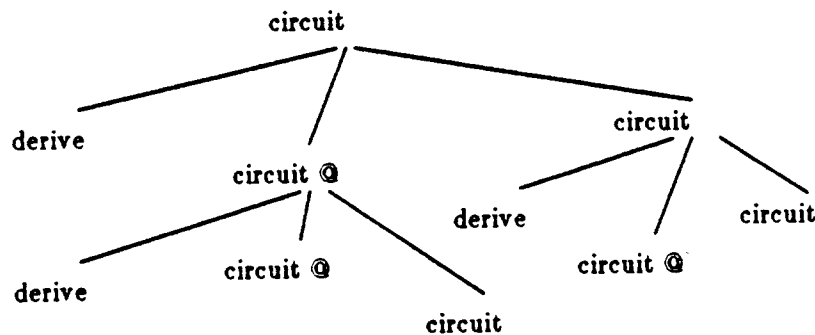

```

circuit(Output,[Config1,Config2]) :-
    derive(Output,Input1,Input2),
    circuit(Input1,Config1),
    circuit(Input2,Config2).

```

This is the clause used in Chapter 3 to illustrate the semi-intelligent backtracking for the automatic circuit design problem. After the 'derive' predicate generates input functions, (Input1,Input2), both 'circuit' subgoals can be executed concurrently. If both input functions can be implemented (i.e. the 'circuit' predicates can successfully derive configurations for input functions), then the time it takes to finish executing these two subgoals would be $\max(T_{s1}, T_{s2})$ instead of $T_{s1} + T_{s2}$ as in sequential execution (where T_{s1} (T_{s2}) denotes the time it takes to implement function Input1 (Input2)). If the function 'Input1' can be implemented and the function 'Input2' cannot, then the time it takes to reject the instance (Input1,Input2) generated by 'derive' would be $\min(T_{s1}, T_{r2})$ instead of $T_{s1} + T_{r2}$ (where T_{r2} denotes the time it takes to fail the second 'circuit' predicate). (Of course, if the function 'Input1' cannot be implemented, then resources are wasted in trying to find concurrently a configuration for the function 'Input2'.) The advantage of the semi-intelligent backtracking is the same as in the sequential execution. That is, when the second 'circuit' predicate fails, it is not necessary to backtrack to the choice-points under the first 'circuit' predicate.

In this example, the proof tree of depth two is shown in Figure 4.1, assuming the first 'circuit' predicate is always forked off and runs concurrently with the second 'circuit' predicate at the exit of the 'derive' predicate. It can be seen that four independent processes, executing



Legend :

@ - denote forked off processes

Figure 4.1 Spawning Processes in AND-Parallel Execution

'circuit' subgoals, are running concurrently. It is clear that the number of processes will be 2^n for a proof tree of depth n , assuming none of the functions of the non-leaf 'circuit' predicates correspond directly to the input signals. In reality, because the cost of maintaining a process will not be zero, there should be a way to limit the degree of multi-processing at run-time. This can be done by explicitly specifying, by the programmer, the maximum number of processes that can be generated by the program. The same approach is used in the HEP machine [43]. It can also be done by the central scheduler which monitors the usage of system resources and decides whether or not more processes can be spawned.

In general, when a clause can take advantage of the semi-intelligent backtracking, it can also take advantage of the AND-Parallel execution. But, not vice versa. That is because semi-intelligent backtracking mostly occurs in non-deterministic programs, while AND-Parallel execution can occur in both deterministic and non-deterministic programs.

4.3. An AND-Parallel Execution Environment

The Aquarius system architecture [42] is shown in Figure 4.2, where PPP stands for Prolog Processor, FPP stand for Floating Point Processor, and IOP stands for I/O Processor. Processors are connected to memory through a crossbar switch. The simplest Aquarius system which has only a single PPP is surround by dash lines. This is the PLM system discussed earlier. A full blown Aquarius system can have multiple PPPs. The Aquarius system has the following features:

- (1) Heterogeneous multiprocessor system - It has special processors, for example PPP, FPP, and IOP, to perform special functions. It is different from systems which consist of a large number of homogeneous processors with fixed interconnection, for example X-tree [44] and FAIM [45].
- (2) Shared memory model - Data can be accessed by each processor and operated upon without interruptions from other processors. The alternative is the message passing model¹. Although the message passing model is cleaner in a parallel execution environment, it has a lot more overhead and is thus inefficient for a modest number of processors.
- (3) Dedicated synchronization memory - A fast synchronization mechanism is crucial for a tightly coupled multiprocessor system. A dedicate synchronization memory system with special hardware support is proposed to achieve this goal.
- (4) Static partitioning - A problem can be partitioned into many tasks by the user during the design of the algorithm by using special languages such as Concurrent Prolog [28] or OCCAM [46]. In the Aquarius project, we propose to compile standard Prolog program and partition it into tasks. This will be accomplished by the compiler, which is similar to the approach of Paraphrase [47] and BULLDOG [48].
- (5) Dynamic scheduling with central scheduler - Tasks are generated at compile time and put into task queues of PPPs at run time, which is similar to the approach of the NYU Ultracomputer [49], although in the Ultracomputer a single global task queue is used. In Aquarius, each processor has its own process queue. The overhead of dynamic scheduling should not offset the advantages of AND-Parallel processing.

¹In the message passing model, there is no shared memory. Both data and control information are passed by messages through communication channels.

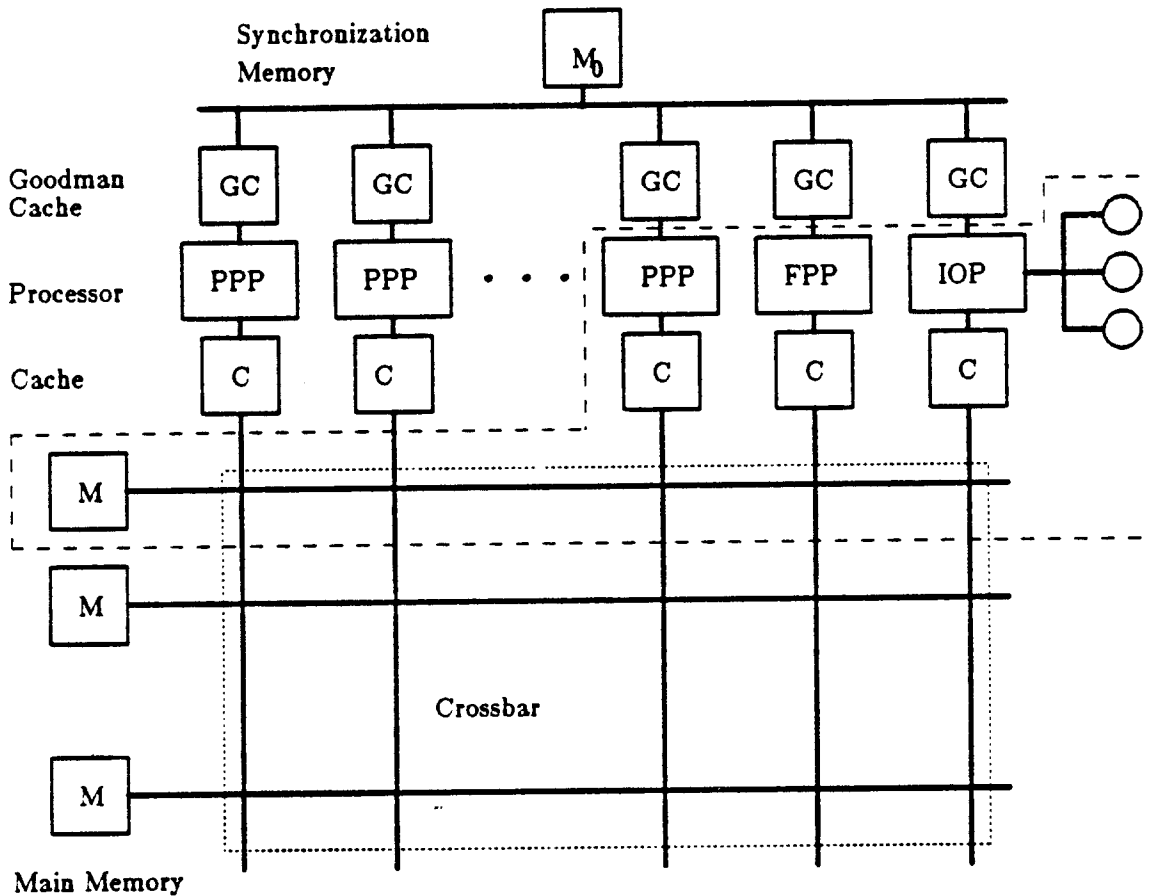


Figure 4.2 The Aquarius Architecture

The static partitioning can be done with the help of the SDDA. The data dependency graphs generated by the SDDA is passed to the compiler. It is not difficult to have a crude measure of the load of an independent task (such as the number of nodes in its AND/OR subtree), and incorporate such measures into the compiler as a criterion to partition tasks.

In AND-Parallel execution of Prolog, a forked off process is assumed to execute either one or a group of subgoals. But, to make it easier to explain, in the following, a process is assumed to execute a single independent subgoal. Forking a process can be reduced to the following actions:

- (1) Set up arguments for the procedure call.
- (2) Create a new process. (A process structure is shown in Figure 4.5).

- (3) Allocate memory space (the Stack², Heap, and Trail) for the new process.
- (4) Select a processor and put the new process into its process queue.

The first action is done by the caller (parent process), and the others are done by the central scheduler.

4.4. Memory Management

Memory Management is probably the most difficult and important issue in parallel processing of Prolog. Symbolic languages are memory hogs. Typically, an efficient garbage collector is implemented to free used but no longer referenced data objects. A garbage collector is invoked when free memory space falls below a certain level.

Typically, Prolog tends to use more memory space than other languages because of the following reasons:

- (1) Heap and trail space used by a procedure call can not be reclaimed until backtracking occurs³.
- (2) An environment frame can not be reclaimed unless all the body literals are deterministic. (If tail-recursion optimization is supported [39], then the environment frame can be recovered when all the body literals except the last one are deterministic.) An environment is needed to store variables in a clause in order to set up arguments for body literals, and to continue the execution at the exit of each body literal.

For a large Prolog program, running short of heap space is a potential threat that the programmer must worry about. Garbage collection (together with tail-recursion optimization) can help to relieve this problem.

In an AND-parallel execution environment, there is yet another important memory management problem besides the garbage collection problem. This is how to allocate memory space to concurrently running processes. This is related to the problem of communicating information, which includes the variable bindings and control information, between processes. The execution flow control, both forward and backward, can be done by using compilation and run-time bookkeeping (by the central scheduler). This control will be discussed later. Memory space, including the environment stack, heap, and trail, can be allocated to independent processes in chunks out of a pool. In this way, it becomes easier to deal with the extension and contraction of memory space of a process. In the AND-parallel execution environment, the environment stack may become a tree, as shown in Figure 4.3, in which stack segments used by forked-off subgoals of a clause form separated tree branches. The *virtual* environment stack viewed by a subgoal consists of stack segments from the current branch to the root. Within each stack segment, the discipline of stack usage remains the same as in sequential execution. The usage of an environment stack is different from the usage of a heap in that, when two variables are bound together, "junior" variable always points to the more "senior" one to avoid creating dangling references when stack space is reclaimed at determinate exit. So, in the AND-parallel execution environment, a method (e.g.

²The 'Stack' refers to the environment stack.

³Logic variables allow a procedure call binds variables in ancestors' environment. Heap space of the procedure can not be reclaimed at the exit of the procedure because that the procedure may bind a variable in an ancestor's environment to structured data on the heap.

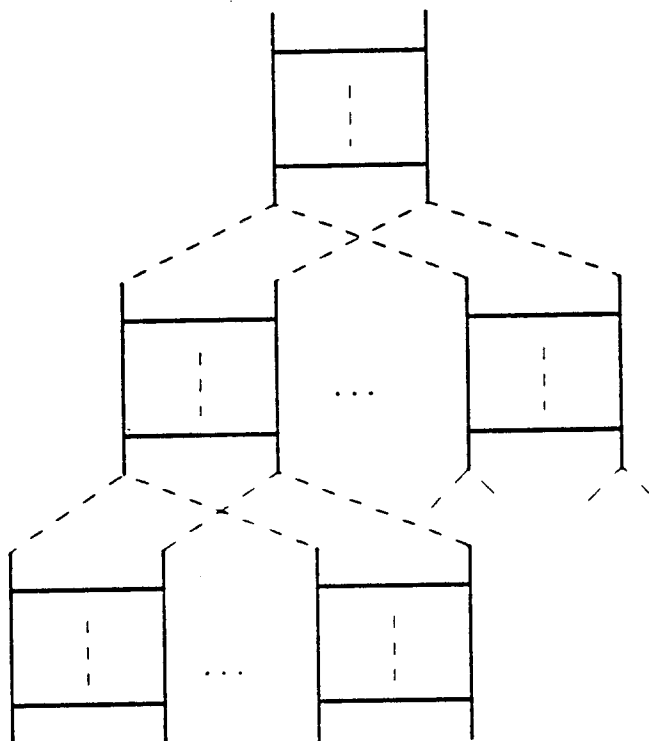


Figure 4.3 Tree-like environment stack in AND-parallel execution

in [50]⁴) must be provided to determine the seniority between two unbound variables. Since all the binding conflicts are resolved by the static data dependency analysis in exploiting AND-parallelism, there are no binding conflicts between parallel running processes as in OR-parallel execution. So, there is no need to shadow non-local variable bindings⁵.

As explained above, the memory space (except, perhaps, environment stack space) can not be reclaimed at the exit of a forked-off independent process. It seems to be desirable to reclaim the memory space (which includes environment stack, heap, and trail) of a forked-off, independent process which is known to be deterministic. To achieve that, it is necessary to trail non-local variable bindings, i.e. bindings of variables which are outside of the data space of a running process. (This can be done by trailing non-local variable bindings in a special

⁴In this method, each environment frame is assigned an invocation level, which is the depth of the proof tree. Since a subgoal can only bind variables in the current environment of the clause it belongs to, or the environment of its ancestors, the invocation level is good enough to determine the seniority between two environment frames.

⁵In OR-parallel execution, if an OR-parallel executing process binds a unbound variable in the ancestor environment or heap, it has to shadow the bindings in its local memory to make it invisible to other OR-parallel

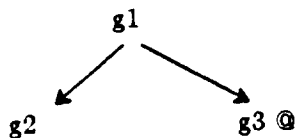
data areas.) Since non-local variable bindings are trailed, the part of the heap which is referenced by non-local variables can be found and copied back to the heap of the parent process. Trailing of non-local variable bindings must also be copied back to the trail of the parent process. Although there is overhead to do the copying, the advantage of removing a forked-off, deterministic process and retrieving all its resources can outweigh the overhead of copying. In this way, the net effect would be as if no forking had occurred. An example is shown in Figure 4.4. In this example, subgoal 'g2' and 'g3' are independent, and subgoal 'g3' is forked-off to be executed non-locally. Assume that 'g3' is deterministic. Then, at the end of executing 'g2' and 'g3', the part of the heap and trail of 'g3' which are associated with non-local variable bindings can be copied back to its parent process. Note that since 'g3' is forked-off before 'g2' is locally executed, the order of using heap (and trail) by the subgoals is preserved. (The reversal of execution order of 'g2' and 'g3' is necessary only when we want to retrieve the memory space of a forked-off, deterministic process at the end of its execution. In the rest of the chapter, it is assumed that this scheme is not supported.)

There are other proposals that try to exploit deterministic AND-parallelism [50, 28]. In general, for a forked-off process which is nondeterministic, both the data space and the process itself, which is known to the central scheduler, should be retained at its successful exit. This is because that the process may be backtracked into later on to find other successful instances.

4.5. Execution Flow Control in AND-Parallel Execution

In this section, the flow control of AND-parallel execution is outlined. Flow control in AND-parallel execution is achieved partly through properly compiled code, and partly through run-time bookkeeping by the central scheduler. The central scheduler is responsible for keeping track of process status, and routing of control messages, such as *fail* and *kill*, between processes. A process can be in either one of the two states, active or non-active. A process is in an active state when it is either on a ready queue or on one of the wait queues (waiting for I/O or for an external function to complete). A process is in a non-active state when the execution is completed. The process structure is retained for the purpose of backtracking. Non-active processes are maintained by the central scheduler. A process, active or non-active, is deleted when it is the target of a *kill* message. A non-active process becomes active and is put into one of the active queues when it is the target of a *fail* (backtracking) message. A process structure is as shown in Figure 4.5. The first entry, called the "command" field, in the process structure is used to specify whether a forward or a backward execution is requested. When a procedure call is forked off, the central scheduler creates a process structure, allocates memory space (environment stack, heap, and trail), copies argument registers and P register into the process structure (other register entries, which include pointers to the environment stack, heap, and trail, in the process structure are also set properly), marks the "command" field in the process structure as "forward", and puts the process structure into the active queue of a processor. If a non-active process is backtracked into (i.e. the target of a *fail* message), the central scheduler marks the "command" field as "backward" and puts the process into the active queue of a processor. When a processor starts executing a process, it first checks the "command" field. If the "command" field contains "forward", the registers are loaded from the process structure and execution starts. If the "command" field contains "backward", then the machine state stored in the last choice-point is restored and execution starts. During the execution, if I/O operations or external functions are encountered, the machine states can be saved into the

$p := g1, g2, g3.$



Legend: @ - denotes forked-off process

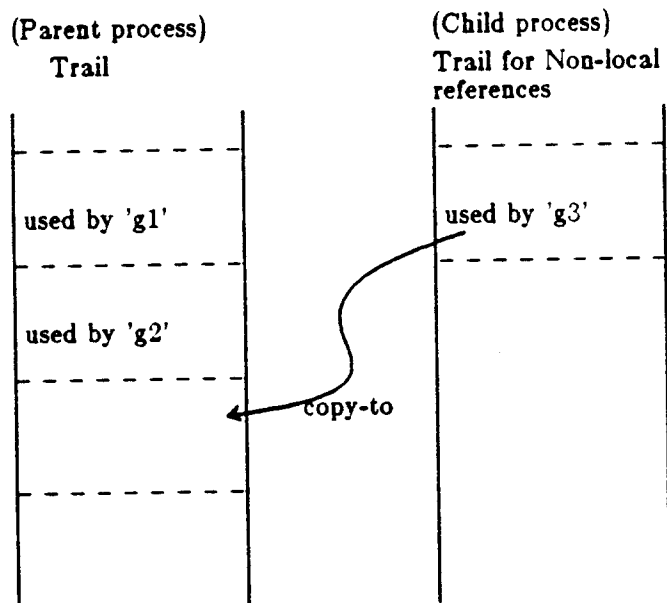


Figure 4.4 Handling a forked-off deterministic procedure call

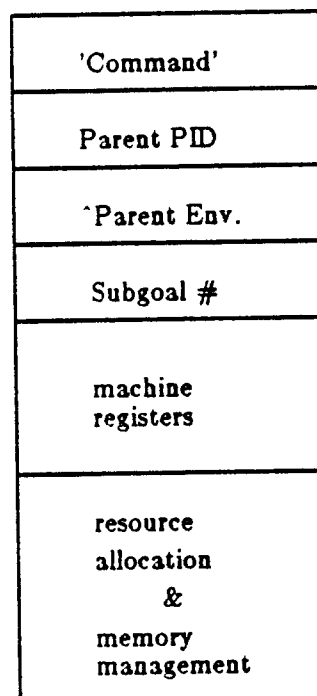


Figure 4.5 A Process Structure

process structure, and put into a wait queue. The parent process ID (Parent PID) and the pointer to the parent environment (^Parent Env.) are used to access the parent environment, from which the child process can access various data structures to achieve backtracking and synchronization. Synchronization and backtracking for the AND-parallel execution will be discussed later.

As discussed in Chapter 2, data dependency graphs (one for each clause) can be generated at compile time. In a data dependency graph, all the subgoals in the same layer can be executed concurrently in forward execution. In Chapter 3, it is shown that data dependency graphs can be used to achieve intelligent backtracking for a sequential Prolog machine. Below, it is shown how the AND-parallel execution flow control can be achieved through the static data dependency analysis and compilation.

The data structures required to support AND-parallel execution are the Join Table (JOIN_TBL), the Process Table (PROC_TBL), and the Backtrack Table (BT_TBL). The Join Table and the Backtrack Table are generated at compile time. Each entry in the Join Table contains two fields, one is the synchronization count and the other is the starting address of a piece of code to be executed when the count reaches zero. A copy of the Join Table is kept in the environment so that a clause can be reentered. The backtrack Table contains information which is needed for failure handling. The Process Table records the

process ID of a forked-off procedure call and the list of synchronization variables (SYNCVAR) that must be updated at the end of a successful execution or before a retry of this procedure call (see Figure 4.7). The Process Table is maintained at run time by the central scheduler. Through the central scheduler, a forking process (parent process) can fork off independent processes (children processes), kill a forked off process, and force backtracking on a forked off process. BT_TBL is used as before to achieve intelligent backtracking. However, it contains more information (discussed later) than in the sequential execution. PROC_TBL is needed to find the process ID of a forked off process when backtracking to the forked off process is demanded. The entry of the JOIN_TBL, which is updated by the backtracked process, has to be incremented to reflect the retry.

An example of a clause, which can take advantage of AND-parallelism, and its compiled intermediate code are shown in Figure 6. The PROC_TBL is updated whenever a procedure call or a fork instruction is executed (as shown in Figure 4.7). For a forked off procedure call, the central scheduler, which is invoked by the fork instruction, will create a new process and assign a new PID to it. Figure 4.7 shows one possible way to compile for AND-parallel execution. The advantage of this approach is that it is easy to implement. The disadvantage of this approach is that the AND-parallel execution is restricted by the sequential nature of the main stream of the compiled code. Because it is possible that the execution is stalled at one synchronization point while the subsequent subgoals are ready to be executed, e.g. in Figure 4.8 the execution may stalled because the join variable for subgoal #6 has not been reduced to zero, while subgoal #7 may be ready to be executed.

4.6. Semi-Intelligent Backtracking in the AND-Parallel Execution Environment

In AND-parallel execution environment, independent subgoals may be forked off to be executed on other processors. Special arrangements are necessary to handle backtracking to/from a forked-off process. Three kinds of process manipulating messages are sent between processes via the central scheduler to cause desired interactions. They are *fail*, *reset* and *kill* messages. *Fail* messages are sent to non-active processes which have successfully executed the subgoals to be backtracked into. *Reset* messages are sent to the right siblings processes (processes which are on the same layer of the dependency graph) of the processes which are backtracked into. *Kill* messages are sent to processes (active or non-active) which are no longer needed.

To understand the usage of the *reset* message, let us look at the sequential execution of Prolog. In the sequential execution, when a subgoal is selected to backtrack into, all the execution histories of the subsequent subgoals to the right of the backtracked subgoal are thrown away. When the backtracked subgoal has another successful instance, the subsequent subgoals are re-executed. In the AND-parallel environment, under certain circumstances (explained later), right sibling subgoals, unlike the other subsequent subgoals which are not on the same layer on the dependency graph, of the backtracked subgoal do not need to be re-executed. So, *reset* messages are sent to the right sibling subgoals instead of *kill* messages. The action taken by a subgoal upon receiving a *reset* message is described later.

When a failure occurs during forward execution, a subgoal is selected to be backtracked into (see the discussion on semi-intelligent backtracking in Chapter 3). Backtracking is achieved by restoring the machine state from the last choice-point of the backtracked subgoal. If the backtracked subgoal is executed by a different process, then a *fail* message is sent to that process. The usages of these messages are illustrated at below.

Receiving *fail* messages -

When a process receives a *fail* message, it sends *kill* messages to all its successor

Example:

```
a:- b,c,d.    /* assume c & d are independent, but depend on b */
```

Compiled code:

```
allocate PROC_TBL/1, JOIN_TBL/1, BT_TBL
    /* allocate environment; PROC_TBL, a copy of the */
    /* JOIN_TBL, and pointers to BT_TBL are kept in the */
    /* the environment. */

{ get instructions }    /* unify with head */

{ put instructions }   /* set up arguments for b */
call b,1,0             /* this call corresponds to literal #1; */

{ put instructions }   /* set up arguments for c */
fork c,2,1            /* this call corresponds to literal #2; */
                    /* on success, it decrements the count */
                    /* field of entry #1 of JOIN_TBL and */
                    /* goes into a non-active state. */

{ put instructions }   /* set up arguments for d */
call d,3,1            /* this call corresponds to literal #3; */
                    /* on success, it decrements the count */
                    /* field of entry #1 of JOIN_TBL. */

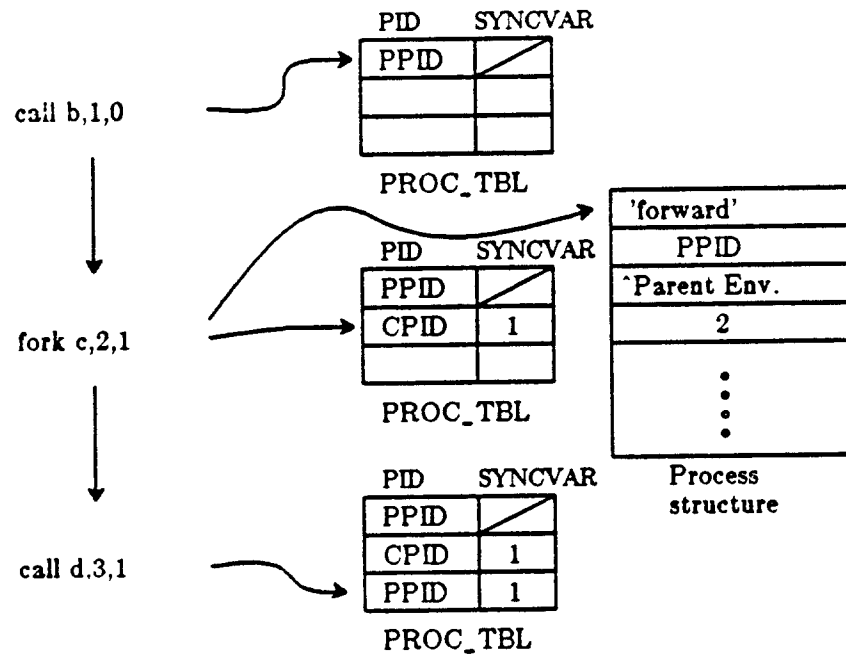
wait 1                /* wait until entry #1 of JOIN_TBL is 0 */

Label1:
    proceed           /* success */

JOIN_TBL:
    2,Label1

BT_TBL:               /* for failure handling */
.
.
.
```

Figure 6 An Example



Note:

PPID - Parent PID
 CPID - Child PID

Figure 4.7 Forking-off an independent procedure call

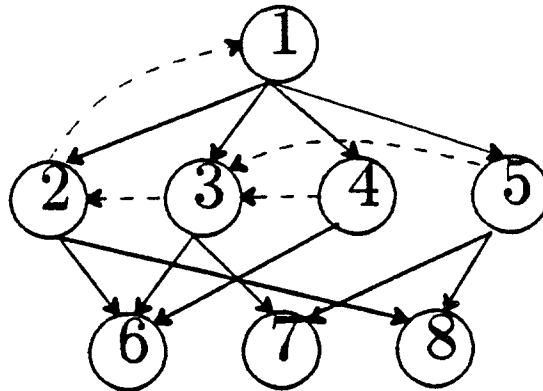


Figure 4.8 A data dependency graph and its type II backtracking paths

subgoals (one layer down in the dependency graph) which are executed by different processes; it also sends *reset* messages to *backward dependent* sibling subgoals which are executed by different processes. As defined in Chapter 3, two subgoals are *backward independent* if one is not on the semi-intelligent backtracking paths of the other. In Figure 4.8, node 4 and 5 are *backward independent* nodes for both type I & II backtracking. For type III backtracking, all nodes on the same layer of a dependency graph are assumed to be *backward dependent*. The reset list for the data dependency graph in Figure 4.8 is shown at below:

Type I & II:

```

[5] ==> ∅
[4] ==> ∅
[3] ==> [4,5]
[2] ==> [3,4,5]
[1] ==> ∅

```

Type III:

```

[8] ==> ∅
[7] ==> [8]
[6] ==> [7,8]
[5] ==> ∅
[4] ==> [5]
[3] ==> [4,5]
[2] ==> [3,4,5]
[1] ==> ∅

```

The reset list contains information about which subgoal will be sent a *reset* message during backtracking. For example, if subgoal #3 is backtracked into via type I

backtracking path, then reset messages are sent to subgoal #4 & #5 if they are executed by different processes. The reset lists for type I and type II backtracking (although in Figure 4.8 they are the same) may be different because in type I backtracking *backward dependency* is determined by considering the subgraph from the root to one layer below the backtracked node, while in type II backtracking the whole graph has to be considered. The Backtrack Table (BT_TBL) mentioned in the previous section contains lists of backtrack nodes as well as reset nodes for each type of backtracking.

Receiving *reset* messages -

When a process receives a *reset* message, it ignores the message if it has only generated one solution, for example a process which executes a deterministic subgoal. If it has generated more than one solutions, then the process will restart from its initial state, that is the state when the process was first created. In the latter case, the synchronization variable has to be updated to reflect the re-execution. If re-execution is required, then it sends *kill* messages to all the successor subgoals which are executed by different processes.

Receiving *kill* messages -

When a process receive a *kill* message, it sends *kill* messages to all the successor subgoals which are executed in different processes, and then terminates itself (returns all its resources back to the central scheduler).

4.6.1. Parallel Backtracking in AND-Parallel Execution Environment

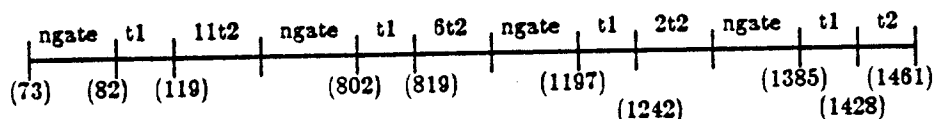
Backtracking is a way to sequentially exploit alternative solution paths. However, in AND-parallel execution environment, there can be several forward as well as backward execution activities going on concurrently. The simplest example is that, during the execution of two independent subgoals, both subgoals can be in backward states at the same time. In this case, two backtrack paths are in two separated proof trees (or graphs) and they can not interfere with each other. The more complicated cases are that two backtrack paths are derived from the same graph, that is, two backtrack paths between subgoals in the same clause. In this case, two backtracking activities can proceed concurrently if one does not interfere with the other (for example, one is not on the reset list of the other). The semantics of the control messages mentioned above is designed exactly to achieve this.

4.7. Performance Improvement of The AND-Parallel Execution - An Example

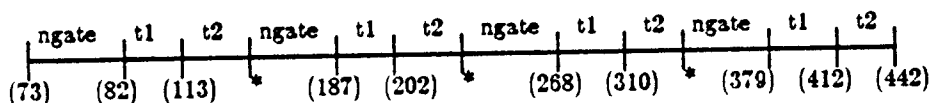
The run-time profile of the automatic circuit design program (see Appendix B) with a query, $t(2,X,[0,0,1,1,0,1,0,1])$, for designing a multiplexer is examined. The improvement of performance due to semi-intelligent backtracking and parallel execution, as well as the number of messages sent between two concurrently running processes are shown in Figure 4.9. In Figure 4.9, the sequence and numbers of times that procedure 'ngate' and 't' (in the clause which describes the NAND gate, and when the 'Depth' is equal to 2^6) are called or retried are shown, where 't1' denotes the first 't' predicate and 't2' denotes the second 't' predicate in the clause ('1t2' means that the second 't2' predicate is retried or called 11 times). The numbers shown in the parentheses are the total numbers of inferences at the exit of procedure calls. In this figure, it is assumed (for simplification) that the independent procedure call 't' is forked-off only when at the entry of the clause the 'Depth' is equal to 2. With sequential execution,

^aAll the descendant procedure calls, with 'Depth' less than 2, which include more invocations of 't' and 'ngate' are not shown.

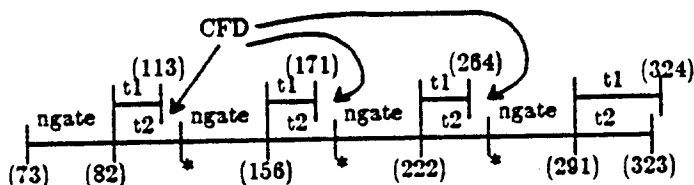
Sequential Execution:



Sequential Execution with
Semi-intelligent Backtracking :



AND-Parallel Execution :



no. of fork instructions : 4
no. of 'fail' messages : 0
no. of 'reset' messages : 0
no. of 'kill' messages : 0

Note:

- (1) CFD : Co-operative failure detection
- (2) The numbers in parentheses are the total no. of inferences at the exit of procedure calls.
- (3) * : denotes the occurrence of a failure.

Figure 4.9 Performance Improvement of the AND-Parallel Execution

to get the first answer requires 1461 inferences. If the semi-intelligent backtracking is exploited, the number of inferences is reduced to 442. With a limit AND-parallel execution, as assumed above, the time it takes to get the first answer is amount to performing 324 sequential inferences. Not much performance gain is shown for the AND-parallel execution because this is a simple example and the maximum number of concurrency is limited to 2. Although the co-operative failure detection (CFD) has potential to reduce the total number of

inferences⁷, in this example it does not. This is because that in the three CFD phases, 't1' has successful instances and takes shorter time to find those instances than the failures of 't2'. There are no messages being passed because the forked-off 't1' has successful instances. Note that, in this simple example, the overhead of executing the 'fork' instructions has not been taken into account. Future simulations must include the overhead of executing 'fork' instructions and passing messages.

4.8. Conclusion

In this chapter, a scheme to exploit the AND-parallel execution in Prolog was proposed. It involves using the SDDA at compile time to detect independent subgoals, and forking-off independent subgoals at run time. Data is accessed through a globally shared memory, while the execution flow controls are achieved by sending messages between concurrently running processes via a central scheduler. It is shown that concurrency for both forward and backward execution as well as the semi-intelligent backtracking can be exploited.

Only one example of AND-parallel execution has been presented here. Clearly much further analysis and simulation is needed to fully evaluate the cost effectiveness of this approach. Nevertheless, it can be seen that the SDDA methodology presented here does have great potential for increasing the speed of Prolog execution through the use of AND-parallelism.

⁷This is because that short failure sequences, in the parallel case, can terminate long chains before they (uselessly) complete.

CHAPTER 5

CONCLUSION

5.1. Conclusions

Prolog, an implementation of a logic programming language, has caught the attention of the computer science community because of its declarative semantics, logic foundation, and efficient implementation. The efficiency of current Prolog systems mainly comes from its simple control strategy which is a left-to-right and top-to-bottom control strategy with built-in naive backtracking. However, because of the naive backtracking, a lot of redundant effort may be wasted for problems which have a flavor of theorem proving. Two ways are proposed in this thesis to improve the performance of Prolog:

- (a) Semi-intelligent backtracking,
- (b) AND-parallel execution.

Both rely on a static data dependency analysis done at compile-time.

Use of compile-time analysis to improve the run-time performance is the main emphasis in this thesis. AND-parallel execution [22] and intelligent backtracking [18,21] have been studied by other researchers. Their approaches are based on either a run-time bookkeeping or a dependency analysis. Although a run-time analysis is more effective than a static (hence worst-case) analysis, it incurs a lot of overhead at run-time and is thus inefficient. For a static analysis, although the compilation costs more, the price only needs to be paid once. Static analysis has been used in the past for the other HLLs to generate more optimized code [51], to partition a program into concurrent subtasks [47, 48], and to translate HLL programs into data flow graphs for a data-flow architecture [52]. For Prolog, the static analysis has

been used to derive the invocation modes to generate more efficient code [35], and for query optimization [53]. In this thesis, it was shown that static analysis can be applied to Prolog to achieve both intelligent backtracking and AND-parallel execution.

The static data dependency analysis (SDDA) is based on a worst-case analysis of run-time variable bindings of Prolog programs. This analysis is easier for Prolog than the other high level languages because of its local variable scoping, concise syntax, and single-assignment variable binding. On the other hand, the analysis is harder for Prolog because of the untyped logical variables, and the non-deterministic execution. A methodology is proposed in this thesis to perform the SDDA. It was shown that the cost of doing the SDDA can be comparable to the cost of compilation (see Table 2.1). For programs without many nested recursions or many activation modes for the same procedure, the cost of doing the SDDA can be 30% to 40% of the cost of compilation. For programs with many nested recursions or many activation modes for the same procedure, the cost of doing the SDDA can be 80% to 300% of the cost of compilation. The SDDA is easily incorporated into the compiler. The output of the SDDA is a collection of data dependency graphs, one for each clause of a Prolog program. A data dependency graph describes the dependency between body literals of a clause. From the data dependency graphs, the compiler can generate codes to achieve both the semi-intelligent backtracking and AND-parallel execution.

Semi-intelligent backtracking can improve the performance of Prolog for problems which have intelligent backtracking paths within a clause boundary. The scheme proposed in this thesis is to derive intelligent backtracking paths (within a clause boundary) from the data dependency graphs generated from the SDDA. At run-time, when a subgoal fails, a backtrack literal can be determined from the pre-computed backtracking paths and the literal can be retried. It was shown that this scheme can be implemented by modifying the compiler and the architecture. The performance improvement (see Table 3.2) can be substantial (up to

1200:1 in execution time) for some programs and the performance is more robust with respect to the ordering of body literals. However, it was observed that semi-intelligent backtracking is useless for certain problems, e.g. the queens problem and the D-algorithm (shown in Appendix B). In these problems, the positions of queens or the logic values of nodes are gradually determined across several clauses or iterations. To achieve intelligent backtracking in these problems require recording the binding history as in the run-time intelligent backtracking [18], which is known to be very expensive. A future research topic is to investigate an efficient scheme to achieve intelligent backtracking which is not restricted by the clause boundary.

While semi-intelligent backtracking is useful only for non-deterministic programs, AND-parallel execution is useful for both deterministic and non-deterministic programs. A scheme to achieve AND-parallel execution was described in this thesis. In this scheme, an independent procedure call can be forked-off and run concurrently in a different processor. Data is accessed through a globally shared memory, while the execution-flow controls are achieved by sending messages between concurrently running processes via a central scheduler. It was shown that both parallel execution and semi-intelligent backtracking can be exploited to improve the performance. Future research directions for the parallel execution are: looking into partition algorithm for subtasking, incorporating the OR-parallel execution, run-time load balancing, and efficient memory management which includes memory allocation, deallocation, and parallel garbage collection.

The static data dependency analysis described in this thesis is used to generate dependency graphs for Prolog programs to achieve both intelligent backtracking and AND-parallel execution. It can also, however, serve for other purposes with minor modifications. Since the SDDA can find the worst-case bindings of variables, it can be used to derive the instantiation mode (as in automatic mode generation [35]) for generating more efficient code.

From the variable binding analysis of the SDDA, the subgoals can be reordered (similar to Conery's ordering algorithm [22]) for more efficient sequential execution. Since the SDDA knows whether or not two terms are coupled (as described in Chapter 2), it can be used to decide whether or not parallel unification can be exploited between two literals. A future research goal is to apply the SDDA to these areas and blend all these possible applications together into the compiler to generate optimized code for more efficient execution of Prolog programs.

BIBLIOGRAPHY

1. A. Colmerauer and et. al., *Etude et Realization d'un System Prolog*, Groupe de Recherche en Intelligence Artificielle, Univ. d'Aix-Marseille, Luminy (1979).
2. J. W. Lloyd, "Foundations of Logic Programming," Technical Report , University of Melbourne (July 1982 (revised March 1984)).
3. J. A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle," *JACM* 12(1) pp. 23-41 (Jan. 1965).
4. R. A. Kowalski, "Algorithm = Logic + Control," *CACM* 22(7) pp. 424-436 (July 1979).
5. J. F. Baldwin and B. W. Pilsworth, "An Inferential Fuzzy Logic Knowledge Base," *Workshop on Logic Programming for Intelligent Systems*, pp. 1-69 (Aug. 1981).
6. I. Bratko, "Knowledge-based Problem-solving in AL3," *Machine Intelligence 10* 10 pp. 73-100 (Jan. 1982).
7. Sanjai Narain, "MYCIN in a Logic Programming Environment," *Digest of Papers of COMPCON Spring '84*, pp. 192-197 (Feb, 1984).
8. M. Dincbas, "A Knowledge-based Expert System for Automatic Analysis in CAD," *Information Processing 80*, pp. 705-710 (Jan. 1980).
9. H. G. Barrow, "Proving the Correctness of Digital Hardware Designs," *Proc. A.A.A.I.*, pp. 17-33 (August 1983).
10. V. Dahl, "On Database Systems Development Through Logic," *ACM Transaction on Database system* 7(1) pp. 102-123 (March. 1982).
11. D. H. D. Warren, "Logic Programming and Compiler Writing," *Software - Practice and Experiene* 10(2) pp. 97-126 (Feb. 1980).
12. D. H. D. Warren, L. M. Pereira, and F. Pereira, "PROLOG - The Language and Its Implementation Compared with LISP," *ACM SIGPLAN Notices* 12(8) pp. 109-115 (August 1977).
13. T. Moto-oka , "Challenge for Knowledge Information Processing Systems," *Proc. of International Conference on Fifth Generation Computer Systems*, pp. 1-86 (Oct. 1981).
14. P. Van Roy, "A Prolog Compiler for the PLM," Masters Thesis, University of California, Berkeley (August 21, 1984).
15. T. P. Dobry, Y. N. Patt, and A. M. Despain, "Design Decisions Influencing the Microarchitecture for a Prolog Machine," *MICRO 17 Proceedings*, (Oct. 1984).
16. D. H. D. Warren, "Implementing Prolog - Compiling Predicate Logic Programs," Research Reports 39 & 40 , Dept. of A. I., Edinburgh Univ. (1977).

-
17. B. Fagin, "Issues in Caching Prolog Goals," Report No. UCB/CSD 84/204, Computer Science Division, University of California, Berkeley (Nov. 1984).
 18. M. Bruynooghe and L. M. Pereira, *Revision of top-down logical reasoning through intelligent backtracking*, Departamento de Informatica, Universidade Nova de Lisboa, Portugal, and Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium (1981).
 19. L. M. Pereira and A. Porto, "An Interpreter of Logic Programs Using Selective Backtracking," Report 3/80, Dept. de Informatica, Universidade Nova de Lisboa (July 1980).
 20. R. Sprugnoli, "Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets," *CACM*, pp. 841-850 (Nov. 1977).
 21. P. Cox and T. Pietrzykowski, "Deduction Plans: A Basis for Intelligent Backtracking," *IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-3*, pp. 52-65 (1981).
 22. J. S. Conery, "The AND/OR Model for Parallel Interpretation of Logic Program," PhD thesis, Dept. of Information and Computer Science, Univ. of California, Irvine (1983).
 23. G. Lindstrom and P. Panangden, "Stream-Based Execution of Logic Programming," *International Symposium on Logic Programming*, pp. 2-11 (Feb. 1984).
 24. A. Ciepielewski and S. Haridi, "Control of Activities in the OR-parallel Token Machine," *International Symposium on Logic Programming*, pp. 49-57 (Feb. 1984).
 25. P. A. Subrahmanyam and J.-H. You, "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming," *Proc. 1984 International Symposium on Logic Programming*, pp. 144-153 (Feb. 1984).
 26. C. Dwork, P. C. Kanellakis, and J. C. Mitchell, "On the Sequential Nature of Unification," *The Journal of Logic Programming* 1(1) pp. 35-50 (June 1984).
 27. M. J. Wise, "A Parallel Prolog: The Construction of a Data Driven Model," *Conf. Record of the Symposium on LISP and Functional Programming*, pp. 56-66 (August 1982).
 28. E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," Research Report, The Weizmann Institute of Science (Feb. 1983).
 29. K. L. Clark and S. Gregory, "PARLOG: A Parallel Logic Programming Language," Research Reports DOC 83/5, Imperial College (March 1983).
 30. T. P. Dobry, J.-H. Chang, A. M. Despain, and Y. N. Patt, "Extending a Prolog Machine for Parallel Execution," *In preparation*, ().
 31. P. Henderson, "," in *Functional Programming*, Prentice-Hall (1980).
 32. J. B. Dennis and K.-S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," *Proc. of the 1979 International Conf. on Parallel Processing*, pp. 35-45 (August 1977).
 33. W. Citrin, "Parallel Unification Scheduling in Prolog," Internal Memo, Computer Science Division, University of California, Berkeley (March 1985).

-
34. J.-H. Chang, A. M. Despain, and D. DeGroot, "AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis," *Digest of Papers of COMPCON Spring '85*, pp. 218-225 (Feb. 1985).
 35. C. S. Mellish, "The Automatic Generation of Mode Declarations for Prolog Programs," DAI Research Paper 163, Dept. of Artificial Intelligence, Univ. of Edinburgh (August 1981).
 36. D. DeGroot and J.-H. Chang, "A Comparison of Two AND-Parallel Execution Models," *Proc. of AFCET INFORMATIQUE, Conference on Hardware and Software Components and Architectures for the 5th Generation*, pp. 271-280 (March 1985).
 37. K. L. Clark and S.-A. Tarnlund, "A First-Order Theory of Data and Programs," *Information Processing 77*, pp. 939-944 (1977).
 38. J.-H. Chang and A. M. Despain, "Semi-Intelligent Backtracking of Prolog Based on A Static Data Dependency Analysis," *To be published in Logic Programming Conference*, ().
 39. D. H. D. Warren, "An Abstract PROLOG Instruction Set," Technical Note 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI, Menlo Park, C. A. (Oct. 1983).
 40. T. P. Dobry, A. M. Despain, and Y. N. Patt, "Performance Studies of a Prolog Machine Architecture," *Conf. Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 180-190 (June 1985).
 41. P. Dembinski and J. Maluszynski, "AND-Parallelism with Intelligent Backtracking for Annotated Logic Programs," *Proc. of 1985 Symposium on Logic Programming*, pp. 29-38 (July 1985).
 42. A. M. Despain and Y. N. Patt, "Aquarius - A High Performance Computing System for Symbolic/Numeric Applications," *Digest of Papers of COMPCON Spring '85*, pp. 376-382 (Feb. 1985).
 43. B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Society of Photo-Optical Instrumentation Engineers, Real-time Signal Processing IV 298* pp. 241-248 (Aug. 1981).
 44. A. M. Despain and D. A. Patterson, "X-TREE: A Tree Structure Multi-processor Computer Architecture," *Conf. Proc, Fifth Annual Symposium on Computer Architecture*, pp. 144-151 (Sep. 1978).
 45. A. L. Davis and S. V. Robison, "The FAIM-1 Symbolic Multiprocessing System," *Digest of Papers of COMPCON Spring '85*, pp. 370-375 (Feb. 1985).
 46. P. Wilson, "OCCAM Architecture Eases System Design - Part I," *Computer Design*, pp. 107-115 (Nov. 1983).
 47. D. A. Padua, D. J. Kuck, and D. L. Lawrie, "High Speed Multiprocessor and Compilation Techniques," *IEEE Trans. Computers C-29* pp. 763-776 (Sep. 1980).
 48. J. A. Fisher and J. J. O'Donnell, "VLIW Machines: Multiprocessors We Can Actually Program," *COMPCON, Spring '84*, pp. 299-305 ().
 49. A. Gottlieb, R. Grishman, C. P. Krushkal, K. P. McAuliffe, L. Rudolph, and M. Snir,

APPENDIX A

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%           ::: Static Data Dependency Analyzer :::
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-([lib,var,main,back]). % consult files 'lib', 'var', 'main', and 'back'

sdda_ :-
    repeat, nl,
    write('          - A Static Data Dependency Analysis -'), nl, nl,
    write('Input file ? '), read(File), % get the name of input file
    write('Graph file ? '), read(Graph), % get the name of the output graph file
    write('Mode file ? '), read(Mode), % get the name of the output mode file
    sdda_(File,Graph,Mode), nl, % perform a SDDA on the input file
    (var(Mode) -> true; tell(Mode)),
    Cpu is cputime,
    write('CPU time= '), write(Cpu), nl,
    Heap is heapused,
    write('HEAP used = '), write(Heap), nl,
    tell(user),
    write('Other input files ? (y/n) '),
    read(Con),
    (Con==y ->
        deconsult_(File),
        close(File),
        (var(Graph) -> true; close(Graph)),
        (var(Mode) -> true; close(Mode)),
        fail;
    true),
    !.

```

"The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer,"
IEEE Trans. Computers C-32 pp. 175-189 (Feb., 1983).

50. R. A. Overbeek, J. Gabriel, T. Lindholm, and E. L. Lusk, "Prolog on Multiprocessors,"
 To be published, Argonne National Laboratory ().
51. Aho and Ullman, "," in *Principles of Compiler Design*, Addison Wesley (1977).
52. W. B. Ackerman, "Data Flow Languages," *Computer*, pp. 15-25 (Feb. 1982).
53. D. H. D. Warren, "Efficient Processing of Interactive Relational Database Queries
 Expressed in Logic," *Proc. of 7th International Conf. on VLDB*, pp. 272-281 (1981).


```

true),
abolish(pred_,4), fail.
sdda(_.,Mode) :- % write out all modes, for debugging purpose
(\+var(Mode) ->
(write('Write all modes ... '),
tell(Mode),
mode_(F,N,Act,Exit),
write(F), tab(6), write(N), tab(6),
write(Act), tab(6), write(Exit), nl,
fail;
tell(user),
write('into file ""'), write(Mode), write('""'), nl);
true),
abolish(mode_,4), abolish(act_,3), abolish(side_,2),
write('Done!!').

% procDep_(F,N,Act,Exit,SideEff,GraphOn,Determ) generates the Exit mode for an
% activation (F,N,Act), and data dependency graphs if GraphOn=1.
% F/N : functor/arity
% Act : activation mode
% SideEff : if there are side-effect predicates or I/O within the procedure F/N
% SideEff=1, else SideEff=0.
% Determ : F/N is deterministic if Determ=1

procDep_(F,N,Act,_,_,GraphOn,_) :-
assertz(notDone_(F,N,Act)), % begin processing (F,N,Act)
functor(H,F,N),
clause(H,Gs,Ref), % get a candidate clause
(recur_(F,N,Act,Ref) -> % if there is a cycle for (F,N,Act,Ref)
fail;
clDep_(H,Gs,F,N,Act,Exit,SideEff,Ref,GraphOn),
updateSide_(F,N,SideEff),
updateExit_(F,N,Act,Exit) ),
fail. % other candidate clauses of F/N ?

procDep_(F,N,Act,Exit,SideEff,GraphOn,Determ) :-
(det_(F,N,Act) -> Determ=1; allGnd_(Act,Determ) ),
mode_(F,N,Act,Exit), % get final Exit mode
(side_(F,N) -> SideEff=1; SideEff=0), % get SideEff flag
retract(notDone_(F,N,Act)). % end processing (F,N,Act)

% clDep_(H,Gs,F,N,Act,Exit,SideEff,Ref,GraphOn) generates the Exit mode
% for a clause (H :- Gs.) with activation mode Act, and the data dependency
% graph if GraphOn=1.

clDep_(H,Gs,F,N,Act,Exit,SideEff,Ref,GraphOn) :-
H=..[F|Args],
argVarL_(Args,AVL), % AVL is an arg-var list
varL_(AVL,Act,VarL), % VarL is the initial var-st list
(Gs==true -> NL=VarL, PredL=[], SideEff=0, PBs=[];
gsDep_(Gs,VarL,NL,1,PredL,PBs,0,SideEff,0,_,Dets,Ref) ),

```

```

findMode_(AVL,NL,Exit,_,_), % find the Exit mode
elimDet_(PBs,Dets,PredB),
((GraphOn=1, act_(F,N,Act)) ->
  assertz(pred_(F,N,PredL,PredB));
true).

% gsDep_(Gs,VarL,NL,Count,PredL,PBs,Sin,Sout,Cin,Cout,Dets,Ref)
% generates a new var-st list NL and the predecessor list PredL for a
% a subgoals list Gs
%
% VarL : var-st list before executing G
% NL : new var-st list after executing G
% Count : literal count
% Ref : reference to the clause Gs is in
% PredL : the predecessor list
% PBs : the backtrack list
% Sin : the literal number of the last side-effect predicate before Gs
% Sout : the literal number of the last side-effect predicate after Gs
% Cin : the literal number of the last control predicate before Gs
% Cout : the literal number of the last control predicate after Gs
% Dets : deterministic literal list in Gs

gsDep_(G,_,_,_,_,_,_,_,_,_,_,_) :-
  var(G), !, % a meta variable
  write('Can not handle meta variable in SDDA'), nl,
  abort.
gsDep_('(G,Gs),VarL,NL,Count,[P1|P2],PBL,Sin,Sout,Cin,Cout,Dets,Ref) :-
  !,
  gsDep_(G,VarL,L1,Count,[P1],PB,Sin,S1,Cin,C1,L2,Ref),
  NC is Count+1,
  gsDep_(Gs,L1,NL,NC,P2,PBs,S1,Sout,C1,Cout,DetL,Ref),
  (PB=[L3] -> PBL=[L3|PBs]; PBL=PBs),
  (L2=[Det] -> Dets=[Det|DetL]; Dets=DetL).
gsDep_(';(_),_,_,_,_,_,_,_,_,_,_) :- !,
  write('Disjunction has not been implemented yet'), nl,
  abort.
gsDep_(G,VarL,NL,C,[[C|PredL]],PBL,Sin,Sout,Cin,Cout,Det,Ref) :-
  functor(G,F,N),
  G=..[F|Args],
  (io_(F,N) ->
    Sout=C,
    Cout=Cin,
    Determ=1,
    PB=[],
    NL=VarL;
  control_(F,N) ->
    Sout=Sin,
    Cout=C,
    Determ=0,
    ((F=fail, N=0) -> true; PB={0}),

```

```

NL=VarL;
argVarL_(Args,ArgVarL),
findMode_(ArgVarL,VarL,Act,P1,Couple), % find the activation mode
(builtIn_(F,N,Act,Exit,SEf,Determ) -> true; % built-in predicate
notDone_(F,N,Act) -> % if there is a cycle
    asserta(recur_(F,N,Act,Ref)),
    procDep_(F,N,Act,Exit,SEf,0,Determ),
    retract(recur_(F,N,Act,Ref));
mode_(F,N,Act,Exit) -> % if mode_(F,N,Act,Exit) is available
    (side_(F,N) -> SEf=1; SEf=0),
    (det_(F,N,Act) -> Determ=1; allGnd_(Act,Determ) );
act_(F,N,A1) -> % if F/N has been examined
    worstMode_(Act,A1,A2),
    (A2=A1 -> % worst-case activation unchanged
        procDep_(F,N,Act,Exit,SEf,0,Determ);
        retract(act_(F,N,A1)), % update act_(F,N,A)
        assertz(act_(F,N,A2)),
        (pred_(F,N,_,_), retract(pred_(F,N,_,_)), fail; true),
        % rm graphs
        (A2=Act -> % Act is the new worst-case act.
            procDep_(F,N,Act,Exit,SEf,1,Determ);
            assertz(reGen_(F,N,A2)),
            procDep_(F,N,Act,Exit,SEf,0,Determ) ));
    assertz(act_(F,N,Act)), % new F/N
    procDep_(F,N,Act,Exit,SEf,1,Determ) ),
varLX_(ArgVarL,Exit,L1),
updateVarL_(VarL,C,L1,NL,Couple),
Cout=Cin,
(SEf=1 -> Sout=C; Sout=Sin) ),
(Sin>Cin ->
    ((Sout=C; Cout=C) -> % if there are special predicates
        T2 is C-1,
        (T2=Sin -> PredL=[Sin];
        T3 is Sin+1,
        numGen_(T2,T3,PredL) ),
        (var(PB) ->
            (var(P1) -> PB=[Cin|PredL]; % handle fail
            P1=[] -> PB=[Cin];
            Cin=0 -> PB=P1;
            PB=[Cin|P1] );
            true);
        (P1=[]; P1=[0]) -> PredL=[Sin], PB=[Cin];
        filter_(Sin,P1,P2),
        PredL=[Sin|P2],
        (Cin=0 -> PB=P1; filter_(Cin,P1,P3), PredL=[Cin|P3] ));
    (Sout=C; Cout=C) -> % if there are i/o or side-effect
        T2 is C-1,
        (T2=Cin -> PredL=[Cin];
        T3 is Cin+1,
        numGen_(T2,T3,PredL) ),

```

```

        (var(PB) ->
            (var(P1) -> PB==PredL;          % handle fail
             P1=[] -> PB=[Cin];
             Cin=0 -> PB=P1;
             PB=[Cin | P1] );
            true);
    (P1=[]; P1=[0]) -> PredL=[Cin], PB=[Cin];
    (Cin=0 -> PredL=P1; filter_(Cin,P1,P2), PredL=[Cin | P2]),
    PB==PredL ),
    (PB=[] -> PBL=[]; PBL=[[C | PB]]),
    (Determin=1 -> Det=[C]; Det=[]).

% filter_(N,L1,L2) eliminates any literal number in L1 which is smaller
% than or equal to N.

filter_(.,[],[]).
filter_(N,[M|L1],L2) :-
    (N<M -> L2=[M|L3];
     L2=L3),
    filter_(N,L1,L3).

% elimDet_(PBs,Dets,PredB) get the new backtrack graph by considering
% deterministic literals

elimDet_([],.,[]).
elimDet_([PB|PBs],Dets,[PB|PredB]) :-
    Dets=[] -> PredB=PBs;
    elimD_(PBs,Dets,[PB],PredB).

elimD_([],.,.,[]).
elimD_([N|L]|PBs],Dets,Bs,[L1|Ls]) :-
    backG_(L,Dets,Bs,PB1),
    L1=[N|PB1],
    elimD_(PBs,Dets,[L1|Bs],Ls).

backG_([],.,.,[]).
backG_([X|L],DetL,PBs,[Y|L1]) :-
    (in_(X,DetL) -> bG_(X,PBs,Y); Y=X),
    backG_(L,DetL,PBs,L1).

bG_(X,PBs,Y) :-
    getLs_(X,PBs,L),
    largest_(L,Y).

worstMode_([],[],[]).
worstMode_([M1|L1],[M2|L2],[M3|L3]) :-
    (M1@>M2 -> M3=M1; M3=M2),
    worstMode_(L1,L2,L3).

updateExit_(F,N,Act,Exit) :-

```

```

retract(mode_(F,_,Act,E1)) ->
    worstMode_(E1,Exit,E2);
    assertz(mode_(F,N,Act,E2));
assertz(mode_(F,_,Act,Exit)).

updateSide_(F,N,SideEff) :-
    SideEff=0 -> true;
    side_(F,N) -> true;
    assertz(side_(F,N)).

% some sample built-in predicates

builtIn_(retract,1,Act,Act,1,1) :- !.           % A more effective (and expensive)
                                                % way to handle 'assert' and 'retract'
                                                % is as described in the thesis

builtIn_(assert,1,Act,Act,1,1) :- !.
builtIn_(is,2,_,[g,g],0,1) :- !.
builtIn_(>=,2,_,[g,g],0,1) :- !.
builtIn_(=<,2,_,[g,g],0,1) :- !.
builtIn_(<,2,_,[g,g],0,1) :- !.
builtIn_(>,2,_,[g,g],0,1) :- !.
builtIn_(var,1,Act,Act,0,1) :- !.
builtIn_(=,2,Act,Act,0,1) :- !.
builtIn_(==,2,[g,_],[g,g],0,1) :- !.
builtIn_(==,2,[_g],[g,g],0,1) :- !.
builtIn_(==,2,Act,Act,0,1) :- !.
builtIn_(=\,2,[g,_],[g,g],0,1) :- !.
builtIn_(=\,2,[_g],[g,g],0,1) :- !.
builtIn_(=\,2,Act,Act,0,1) :- !.

control_(!,0).
control_(fail,0).
control_(repeat,0).

io_(read,1).
io_(write,1).
io_(nl,0).

allGnd_([],1).
allGnd_([X|L],Y) :-
    X=g ->    allGnd_(L,Y);
    Y=0.

numGen_(N,M,L) :-
    N=M -> L=[N],
    M1 is M+1,
    numGen_(N,M1,L1),
    L=[M|L1].

% get a keyed list
getLs_(X,[],[]).

```

```

getLs_(X,[[Y|L]|L1],L2) :-
    X=Y -> L2=L;
    getLs_(X,L1,L2).

%%%
%
%           ::: var :::
%
%%%

% argVarL_(Args,ArgVarL) generates ArgVarL for Args.
% ArgVarL : an arg-var list
% Args : an arg list
% e.g. arg-var list of [f(X,Y),3,[X|Z]] is [[X,Y],[X,Z]]

argVarL_([],[]).
argVarL_([Arg|Args],[AV|AVs]) :-
    argV_(Arg,AV),
    argVarL_(Args,AVs).

argV_(Arg,L) :-
    atomic(Arg) -> L=[];
    var(Arg) -> L=[Arg];
    Arg=..[_|Args], getVar_(Args,L).

% getVar_(Args,Vars) generates a var-list(Vars) for a given arg-list(Args).

getVar_([],[]).
getVar_([Arg|Args],L) :-
    getV_(Arg,L1),
    getVar_(Args,L2),
    concat_(L1,L2,L).

getV_(Arg,L) :-
    atomic(Arg) -> L=[];
    var(Arg) -> L=[Arg];
    Arg=..[_|Args],
    getVar_(Args,L).

% varL_(AVs,Act,L) generates var-st list L with given AVs and Act.
% AVs : a arg-var list
% Act : the activation mode of the literal whose arg-var list is AVs
% There are three possible modes for each variable (name) in a clause:
%     g - denotes a variable is grounded
%     i - denotes a variable is not grounded and is an independent variable
%     s - denotes a variable is coupled with at least another variable
% e.g. the var-st list of [[X,Y],[X,Z]] with an activation mode
%     [i,g] is [[X,g,0],[Y,i,0],[Z,g,0]] where '0' denotes that
%     the generator of these variable is the calling literal.

varL_(AVs,Act,L) :-

```

```

varList_(AVs,Act,[],L1),
dupElim_(L1,L2),
postVarL_(L2,L).

varList_([],[],L,L).
varList_([AV|AVs],[Mode|Modes],L1,L2) :-
  AV=[] -> varList_(AVs,Modes,L1,L2);
  (Mode=r -> % can be either 'i' or 's'
   (AV=[H] -> L3=[[H,i]]; spread_(AV,s,L3));
   spread_(AV,Mode,L3)),
  concat_(L3,L1,L4),
  varList_(AVs,Modes,L4,L2).

spread_([],_,[]).
spread_([Var|Vars],Mode,[[Var,Mode]|L]) :- spread_(Vars,Mode,L).

dupElim_([],[]).
dupElim_([[V,M]|VMs],[[V,NM,0]|FVMs]) :- % calling literal is the gen.
                                           % of all the variables initially
  eliminate_(M,NM,V,VMs,NVMs),
  dupElim_(NVMs,FVMs).
eliminate_(M,M,_,[],[]).
eliminate_(M,NM,V1,[[V2,M2]|L1],L2) :-
  V1\==V2 -> eliminate_(M,NM,V1,L1,L3), L2=[[V2,M2]|L3];
  (M=g; M2=g) -> NM=g, eliminate_(g,g,V1,L1,L2);
  (M@>=M2) -> eliminate_(M,NM,V1,L1,L2);
  eliminate_(M2,NM,V1,L1,L2).

% convert a single 's' to 'i'
postVarL_([],[]).
postVarL_([VM|VMs],L) :-
  VM=[V,s|T] -> (findS_(VMs) -> L=[VM|VMs]; L=[[V,i|T]|VMs]);
  postVarL_(VMs,NVMs),
  L=[VM|NVMs].
findS_(_) :- fail.
findS_([VM|L]) :-
  VM=[_,s|_] -> true;
  findS_(L).

% varLX_(AVs,Exit,L) generates var-mode list L given AVs and Exit.
% AVs : a arg-var list
% Exit : the exit mode of the listeral whose arg-var list is AVs
% e.g. [[X,Y],[Z]] with exit mode [g,i] has a var-mode list
%      [[X,g],[Y,g],[Z,i]]

varLX_(AVs,Exit,L) :-
  varList_(AVs,Exit,[],L1),
  dupElimX_(L1,L2),
  postVarL_(L2,L).

```

```

dupElimX_([],[]).
dupElimX_([[V,M]|VMs],[[V,NM]|FVMs]) :-
    eliminate_(M,NM,V,VMs,NVMs),
    dupElimX_(NVMs,FVMs).

% findMode_(AVs,VarL,Act,PredL,Couple) derives Act, PredL, and Couple from
% given AVs and VarL.
%
% Act : activation mode
% PredL : predecessor list for the current literal
% Couple : a flag to denote that the literal uses a var in the coupled group
% AVs : arg-var list of the current literal
% VarL : the current var-st list

findMode_(AVs,VarL,Act,PredL,Couple) :-
    elimGnd_(AVs,VarL,AVMs,L,Couple),% AVMs only contains var. of mode i & s
    dupEl_(L,PredL),
    checkCouple_(AVMs,Act).

elimGnd_([],_,[],[],0).
elimGnd_([AV|AVs],L,[AVM|AVMs],L1,C) :-
    elimG_(AV,L,AVM,L2,C1),
    elimGnd_(AVs,L,AVMs,L3,C2),
    (C1=0 -> C=C2; C=1),
    concat_(L2,L3,L1).

elimG_([],_,[],[],0).
elimG_([V|Vs],L,L1,[Pred|L2],C) :-
    eG_(V,L,VM,Pred,C1),
    elimG_(Vs,L,VMs,L2,C2),
    (C1=0 -> C=C2; C=1),
    (VM=[] -> L1=VMs; L1=[VM|VMs]).

eG_(V,[],[V,i],0,0). % encounter a new variable
eG_(V,[[W,M,P]|L],VM,Gen,C) :-
    V==W -> (M=g -> VM=[]; VM=[V,M]),
    (M=s -> C=1;C=0),
    Gen=P;
    eG_(V,L,VM,Gen,C).

dupEl_([],[]).
dupEl_([X|L1],L2) :-
    (in_(X,L1) -> L2=L3;
    L2=[X|L3]),
    dupEl_(L1,L3).

checkCouple_([],[]).
checkCouple_([AVM|AVMs],[M|Ms]) :-
    (AVM=[] -> M1=g;
    getMode_(AVM,AVMs,M1,Ms)),

```



```

(var(M) ->
  ((AVM=[_], M1=r) -> M=i; M=M1);
 true),
checkCouple_(AVMs,Ms).

getMode_([],_,M,_) :- var(M) -> M=r; true.
getMode_([[V,M] | VMs],AVMs,M1,Ms) :-
  findV_(V,M,AVMs,M1,Ms),
  getMode_(VMs,AVMs,M1,Ms).

findV_(_,_,[],_,[]).
findV_(V,M,[AVM | AVMs],M1,[M2 | Ms]) :-
  rV_(V,M,AVM,M1,M2),
  findV_(V,M,AVMs,M1,Ms).

rV_(_,_,[],_,_).
rV_(V,M,[[W,M3] | VMs],M1,M2) :-
  M=s -> ((V==W; M3=s) -> M1=s, M2=s; true);
  % M=i
  V==W -> M1=s, M2=s;
  rV_(V,M,VMs,M1,M2).

% updateVarL_(VarL,N,ArgVL,NVL,Couple) generates a new var-st list NVL
% with given N, VarL, argVL, Couple.
% VarL : the old var-st list
% N : a literal number
% ArgVL : the var-mode list contributed by the literal at its exit
% Couple : a flag which denotes that the literal uses variables from the
% coupled group.

updateVarL_(VarL,N,ArgVL,NVL,Couple) :-
  updateVL_(VarL,N,ArgVL,L,C),
  ((C=0, Couple=0) -> L1=L; coupleGen_(N,L,L1)),
  postVarL_(L1,NVL).

updateVL_(L,_,[],L,0).
updateVL_(VarL,N,[[V,M] | L],NVL,C) :-
  updateGen_(VarL,N,V,M,L1,C1),
  updateVL_(L1,N,L,NVL,C2),
  (C1=0 -> C=C2; C=1).

updateGen_([],N,V,M,[[V,M,N]],0). % a new variable
updateGen_([VM | L],N,V,M,L1,C) :-
  VM=[W,M1,N1],
  (W==V -> (M1@>M -> C=0, (M=g -> L1=[[V,g,N] | L]; L1=[VM | L]);
  M1@<M -> (M1=g -> C=0, L1=[VM | L]; % g -> s or g -> i
  C=1, L1=[[V,s,N] | L]); % i -> s
  C=0, L1=[VM | L]);
  updateGen_(L,N,V,M,L2,C),
  L1=[VM | L2]).

```

```

coupleGen_(_,[],[]).
coupleGen_(N,[VM|L],NL) :-
    (VM=[W,s,N1] -> NL=[[W,s,N]|L1]; NL=[VM|L1]),
    coupleGen_(N,L,L1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%           ;;; back ;;;
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% findBack_(PredB,BackL) generates the backtrack list from the predecessor
% list.

findBack_(PredB,BackL) :-
    PredB=[] -> BackL=[];
    PredB=[X] -> BackL=PredB;
    getPred_(PredB,L1), % get a list of literals which are predecessors
    noLeaf_(L1,NotLeaf), % find non-leaf literals
    reach_(PredB,ReachL), % construct the reachable list
    reverse_(PredB,L2),
    reverse_(ReachL,L3),
    findB_(L2,BackL,[],NotLeaf,[],L3).

noLeaf_([],[]).
noLeaf_(X|L,L1) :-
    in_(X,L) -> noLeaf_(L,L1);
    noLeaf_(L,L2), L1=[X|L2].

getPred_([],[]).
getPred_([[N|Pred]|PredB],L) :-
    getPred_(PredB,L1),
    concat_(Pred,L1,L).

findB_([],L,L,-,-).
findB_([[N|Pred]|PredB],L1,L2,NotLeaf,BackFroms,Reachs) :-
    largest_(Pred,M),
    (in_(N,NotLeaf) -> % if N is not a leaf literal
        backFLit_(N,L2,L),
        backFSet_(L,BackFroms,L3), % L3 is the backfrom set of N
        reachable_(N,L3,K,Reachs,Rs),
        L4=[[N,M,K]|L2];
        L4=[[N,M]|L2], L3=[], Rs=Reachs), % if N is a leaf literal
    findB_(PredB,L1,L4,NotLeaf,[[N|L3]|BackFroms],Rs).

backFLit_(_,[],[]).
backFLit_(N,[[M,B1|L]|Bs],L1) :-
    (B1=N -> (L=[] -> L1=[M|L2]; % M is a leaf literal and M -> N (type I)
        L1=L2);
        L=[N] -> L1=[M|L2]; % M is not a leaf literal and M -> N (type II)

```

```

    L1=L2),
    backFLit_(N,Bs,L2).

backFSet_([],_,[]).
backFSet_([X|L],BFs,BF) :-
    getLs_(X,BFs,L1),
    backFSet_(L,BFs,L2),
    concat_([X|L1],L2,BF).

reachable_(_,_,0,[],[]).
reachable_(M,BF,K,[[N|R]|Rs],NRs) :-
    M=N -> reachable_(M,BF,K,Rs,NRs);
    (intersect_(BF,R) -> K=N, NRs=Rs;
     reachable_(M,BF,K,Rs,NRs)).

intersect_(_,[]) :- fail.
intersect_(BF,[X|L]) :-
    in_(X,BF) -> true;
    intersect_(BF,L).

reach_(L1,L2) :-
    sucL_(L1,L3),
    getSuc_(L3,L4),
    reverse_(L4,L5),
    reachL_(L5,[],L2).

sucL_([],[]).
sucL_([[X|L4]|L],L1) :-
    element_(X,L4,L2),
    sucL_(L,L3),
    concat_(L2,L3,L1).

element_(X,[],[]).
element_(X,[Y|L],[[Y,X]|L1]) :-
    element_(X,L,L1).

getSuc_([],[]).
getSuc_([[X,Y]|L],L1) :-
    getS_(X,L,L3,L4),
    getSuc_(L3,L5),
    (X=0 -> L1=L5; L1=[[X,Y|L4]|L5]).

getS_(_,[],[],[]).
getS_(X,[M|L],L1,L2) :-
    M=[Y,N],
    getS_(X,L,L3,L4),
    (X=Y -> L1=L3,L2=[N|L4];
     L1=[M|L3],L2=L4).

reachL_([],L,L).

```

```

reachL_([[N|L]|L1],L2,L3) :-
    getR_(L,L2,L4),
    reachL_(L1,[[N|L4]|L2],L3).

getR_([],_,[]).
getR_([X|L],L1,L2) :-
    getLs_(X,L1,L3),
    getR_(L,L1,L4),
    concat_([X|L3],L4,L2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%           ::: lib :::
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% define new operators

:- op(700,xfx,[\=]).
X\=Y :- X=Y, !, fail.
X\=Y.

% concatenation of two lists

concat_([],L,L).
concat_([X|L1],L2,[X|L3]) :- concat_(L1,L2,L3).

% reverse a list

reverse_([],[]).
reverse_([X|L1],L3) :-
    reverse_(L1,L2),
    concat_(L2,[X],L3).

% check for membership

in_(X,[Y|L]) :-
    X==Y -> true;
    in_(X,L).

% deconsult_(File) cleans up all the predicates used in the File.

deconsult_(File) :-
    see(File),
    repeat,
    read(C1),
    (C1\=end_of_file -> retract(C1), fail; true),
    !, see(user).

```

```
% find the largest natural number in a list
```

```
largest_(L,X) :- closest_(L,0,X).
```

```
closest_([],X,X).
```

```
closest_([X|L],Y,Z) :-
```

```
    X > Y -> closest_(L,X,Z);
```

```
    closest_(L,Y,Z).
```

APPENDIX B

Circuit (Automatic Circuit Design):

```

entry_(t(g,i,g)).
% Input signals
t(_, 0 , [0,1,0,1,0,1,0,1]).
t(_, 1 , [0,0,1,1,0,0,1,1]).
t(_, 2 , [0,0,0,0,1,1,1,1]).
t(_i0 , [1,0,1,0,1,0,1,0]).
t(_i1 , [1,1,0,0,1,1,0,0]).
t(_i2 , [1,1,1,1,0,0,0,0]).

% Inverters
t(Depth, [i,Z], Table) :-
    Depth > 0,
    D is Depth -1,
    sint(Table, Itable),
    t(D, Z, Itable).

% Main NAND gate clause.
t(Depth, [n,Y,Z], Table) :-
    Depth > 0,
    D is Depth -1,
    ngate(Table, A, B),
    t(D,Y,A),
    t(D,Z,B).

% Inverter signal transformation.
sint([],[]).
sint([X,..T1],[,..T2]) :- var(X), sint(T1, T2),!.
sint([0,..T1],[1,..T2]) :- sint(T1, T2).
sint([1,..T1],[0,..T2]) :- sint(T1, T2).

% Optimized gate signal transformation.
ngate([], [], []).
ngate([X,..T0], [,..T1], [,..T2]) :- var(X), !, ngate(T0, T1, T2).
ngate([0,..T0], [1,..T1], [1,..T2]) :- ngate(T0, T1, T2).
ngate([1,..T0], [,..T1], [0,..T2]) :- tgate(T0, T1, T2).
tgate([], [], []).
tgate([X,..T0], [,..T1], [,..T2]) :- var(X), !, tgate(T0, T1, T2).
tgate([0,..T0], [1,..T1], [1,..T2]) :- tgate(T0, T1, T2).
tgate([1,..T0], [,..T1], [0,..T2]) :- tgate(T0, T1, T2).
tgate([1,..T0], [0,..T1], [,..T2]) :- tgate(T0, T1, T2).

```

Quicksort:

```

entry_(quicksort(g,i)).
quicksort(Unsorted,Sorted) :- qsort(Unsorted,Sorted,[]).
qsort([X|Unsorted],Sorted,Rest) :-
    partition(Unsorted,X,Smaller,Larger),
    qsort(Smaller,Sorted,[X|Sorted1]),
    qsort(Larger,Sorted1,Rest).
qsort([],L,L).
partition([],_,[],[]).
partition([X|Xs],A,Smaller,[X|Larger]) :-
    A < X, partition(Xs,A,Smaller,Larger).
partition([X|Xs],A,[X|Smaller],Larger) :-
    A >= X, partition(Xs,A,Smaller,Larger).

```

Population query:

```

entry_(query(i,i,i,i)).
query(C1,D1,C2,D2) :-
    density(C1,D1),
    density(C2,D2),
    D1 > D2,
    T1 is 20*D1,
    T2 is 21*D2,
    T1 < T2.

density(C,D) :- pop(C,P), area(C,A), D is (P*100)/A.

pop(china, 8250). area(china, 3380).
pop(india, 5863). area(india, 1139).
pop(ussr, 2521). area(ussr, 8708).
pop(usa, 2119). area(usa, 3609).
pop(indonesia, 1276). area(indonesia, 570).
pop(japan, 1097). area(japan, 148).
pop(brazil, 1042). area(brazil, 3288).
pop(bangladesh, 750). area(bangladesh, 55).
pop(pakistan, 682). area(pakistan, 311).
pop(w_germany, 620). area(w_germany, 96).
pop(nigeria, 613). area(nigeria, 373).
pop(mexico, 581). area(mexico, 764).
pop(uk, 559). area(uk, 86).
pop(italy, 554). area(italy, 116).
pop(france, 525). area(france, 213).
pop(phillipines, 415). area(phillipines, 90).
pop(thailand, 410). area(thailand, 200).
pop(turkey, 383). area(turkey, 296).
pop(egypt, 364). area(egypt, 386).
pop(spain, 352). area(spain, 190).

```

```

pop(poland, 337). area(poland, 121).
pop(s_korea, 335). area(s_korea, 37).
pop(iran, 320). area(iran, 628).
pop(ethiopia, 272). area(ethiopia, 350).
pop(argentina, 251). area(argentina, 1080).

```

Serialize:

```

entry_(serialize(g,i)).
serialize(L,R) :-
    pairlists(L,R,A),
    arrange(A,T),
    numbered(T,1,N).

pairlists([X|L], [Y|R], [pair(X,Y)|A]) :- pairlists(L,R,A).
pairlists([], [], []).

arrange([X|L], tree(T1, X, T2)) :-
    split(L, X, L1, L2),
    arrange(L1, T1),
    arrange(L2, T2).
arrange([], void).

split([X|L], X, L1, L2) :- !, split(L, X, L1, L2).
split([X|L], Y, [X|L1], L2) :- before(X,Y), !, split(L,Y,L1,L2).
split([X|L], Y, L1, [X|L2]) :- before(Y,X), !, split(L,Y,L1,L2).
split([], _, [], []).

before(pair(X1,Y1), pair(X2,Y2)) :- X1 < X2.

numbered(tree(T1, pair(X,N1), T2), N0, N) :-
    numbered(T1, N0, N1),
    N2 is N1+1,
    numbered(T2,N2,N).
numbered(void,N,N).

```

Determinate concat:

```

entry_(concat(g,g,i)).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).

```


Mu-math:

```

entry_(prove(g)).
prove(X) :- prove(X,1).
prove(X,N) :- write('trying depth '), write(N), nl, theorem(X,1,N).
prove(X,N) :- N1 is N + 1, prove(X,N1).

theorem([m,i],Depth,MaxDepth) :- write('[m,i] - axiom'),nl.
theorem(Y,Depth,MaxDepth) :- Depth =< MaxDepth, concat(_,[i,u],Y),
    concat(X,[u],Y), NDepth is Depth + 1, theorem(X,NDepth,MaxDepth),
    write(Y), write(' from '),write(X), write(' by rule I'), nl.
theorem([m|Double],Depth,MaxDepth) :- Depth =< MaxDepth, concat(X,X,Double),
    NDepth is Depth + 1, theorem([m|X],NDepth,MaxDepth),
    write([m|Double]), write(' from '), write([m|X]),
    write(' by rule II'), nl.
theorem(Y,Depth,MaxDepth) :- Depth =< MaxDepth, replace_u(Y,X),
    NDepth is Depth + 1, theorem(X,NDepth,MaxDepth), write(Y),
    write(' from '), write(X), write(' by rule III'), nl.
theorem(Y,Depth,MaxDepth) :- Depth =< MaxDepth, add_uu(Y,X),
    NDepth is Depth + 1, theorem(X,NDepth,MaxDepth), write(Y),
    write(' from '), write(X), write(' by rule IV'), nl.

replace_u([u|Rest],[i,i,i|Rest]).
replace_u([X|Rest],[X|NewRest]) :- replace_u(Rest,NewRest).

add_uu(X,[u,u|X]).
add_uu([X|Rest],[X|NewRest]) :- add_uu(Rest,NewRest).

concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).

```

Queens (clever):

```

entry_(queens(i)).

queens(Config) :- solution(c(0,[]),Config).
solution(c(5,Config),Config) :- !.
solution(c(M,Config),Conf) :- expand(c(M,Config),c(M1,Conf1)),
    solution(c(M1,Conf1),Conf).

expand(c(M,Q),c(M1,[p(M1,K)|Q])) :- M1 is M+1, column(K), noattack(p(M1,K),Q).

column(1).
column(2).
column(3).
column(4).
column(5).

```

```

noattack(P,[]).
noattack(P,[Q|L]) :- noattack(P,L), ok(P,Q).

ok(p(R1,C),p(R2,C)) :- !, fail.
ok(p(R1,K1),p(R2,K2)) :- Difr is R2-R1, abs(Difr,Abs), Dife is K2-K1,
                           abs(Dife,Abs), !, fail.

ok(P,Q).

abs(N,N) :- N>0, !.
abs(N,M) :- M is 0-N.

```

Queens (simple):

```

entry_(queens(g,i)).
queens(L,Config) :- perm(L,P), pair(L,P,Config), safe([],Config).

perm([],[]).
perm([X|Y],[U|V]) :- delete(U,[X|Y],W), perm(W,V).

delete(X,[X|Y],Y).
delete(U,[X|Y],[X|V]) :- delete(U,Y,V).

pair([],[],[]).
pair([X|Y],[U|V],[p(X,U)|W]) :- pair(Y,V,W).

safe(Left,[]).
safe(Left,[Q|R]) :- test(Left,Q), safe([Q|Left],R).

test([],Q).
test([R|S],Q) :- test(S,Q), notondiagonal(R,Q).

notondiagonal(p(C1,R1),p(C2,R2)) :- C is C1-C2, R is R1-R2, C==R,
                                     NR is R2-R1, C==NR.

```

Query:

```

main :- student(Stud,Course1), course(Course1,Day1,Room),
        professor(Prof,Course1),student(Stud,Course2),
        course(Course2,Day2,Room),professor(Prof,Course2),
        not(Course1==Course2),
        write(Stud),
        write(Prof),
        write(Room),
        write(Course1),
        write(Course2).

```

```

student(robert,prolog).
student(john,music).
student(john,prolog).
student(john,surf).
student(mary,science).
student(mary,art).
student(mary,physics).
professor(luis,prolog).
professor(luis,surf).
professor(maurice,prolog).
professor(eureka,music).
professor(eureka,art).
professor(eureka,science).
professor(eureka,physics).
course(prolog,monday,room1).
course(prolog,friday,room1).
course(surf,sunday,beach).
course(math,tuesday,room1).
course(math,friday,room2).
course(science,thursday,room1).
course(science,friday,room2).
course(physics,thursday,room3).
course(physics,saturday,room2).

```

Exhaustive-Coloring:

```

main:-
  map(green,B,C,D,E),
  fail.
map(A,B,C,D,E):-
  next(A,B), next(A,C), next(A,D),
  next(B,C), next(C,D), next(B,E),
  next(C,E), next(D,E).
next1(green,red).
next1(green,yellow).
next1(green,blue).
next1(red,blue).
next1(red,yellow).
next1(blue,yellow).
next2(X,Y):-
  next1(Y,X).
next(X,Y):-
  ( next1(X,Y);
  next2(X,Y)).

```

Color13 (good):

```

main:- map(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13),
        write(R1), write(R2), write(R3),
        write(R4), write(R5), write(R6),
        write(R7), write(R8), write(R9),
        write(R10), write(R11), write(R12),
        write(R13).
map(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13):-
        next(R1,R13),next(R1,R2),next(R2,R13),next(R2,R4),next(R4,R10),
        next(R6,R10),next(R8,R13),next(R6,R13),next(R2,R3),next(R3,R4),
        next(R3,R13),next(R3,R5),next(R5,R6),next(R5,R13),next(R4,R5),
        next(R5,R10),next(R1,R7),next(R7,R13),next(R2,R7),next(R4,R7),
        next(R7,R8),next(R4,R9),next(R9,R10),next(R8,R9),next(R9,R13),
        next(R6,R11),next(R10,R11),next(R11,R13),next(R9,R12),next(R11,R12),
        next(R12,R13).
next(blue,yellow).
next(blue,red).
next(blue,green).
next(yellow,blue).
next(yellow,red).
next(yellow,green).
next(red,blue).
next(red,yellow).
next(red,green).
next(green,blue).
next(green,yellow).
next(green,red).

```

Color13 (bad):

```

main:- color(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13),
        write(R1), write(R2), write(R3),
        write(R4), write(R5), write(R6),
        write(R7), write(R8), write(R9),
        write(R10), write(R11), write(R12),
        write(R13).
color(R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13):-
        next(R1,R2),next(R2,R3),next(R3,R4),next(R4,R5),next(R5,R6),
        next(R6,R11),next(R11,R12),next(R12,R13),next(R9,R13),next(R9,R10),
        next(R4,R10),next(R4,R7),next(R7,R8),next(R2,R7),next(R6,R10),
        next(R2,R13),next(R6,R13),next(R2,R4),next(R8,R13),next(R4,R9),
        next(R3,R5),next(R8,R9),next(R1,R13),next(R3,R13),next(R5,R13),
        next(R7,R13),next(R11,R13),next(R9,R12),next(R5,R10),next(R10,R11),
        next(R1,R7).
next(blue,yellow).
next(blue,red).
next(blue,green).

```

```

next(yellow,blue).
next(yellow,red).
next(yellow,green).
next(red,blue).
next(red,yellow).
next(red,green).
next(green,blue).
next(green,yellow).
next(green,red).

```

D-Algorithm in Circuit Testing:

```

test(Ckt):-
    consult(Ckt),           % circuit description is in file Ckt
    repeat,
    print('Give the fault node and fault type: (node, type)'), nl,
    print('e.g. (5,d) or (3,nd)'), nl,
    read((FaultNode,Fault)),
    pattern(TC,FaultNode,Fault,Out),
    print('The fault (',
    print((FaultNode,Fault)),
    print(') can be detected at output node '),
    print(Out), nl,
    print(' with the following test cube:'), nl,
    print(TC), nl, nl, nl,
    fail.

% D-Algorithm
pattern(TC1,FaultNode,Fault,Out):-
    assign(TC1,TC),           % assign test-cube
    ( gate(N,Type,NumIn,In,FaultNode) -> % if success, then output s-a-f
      ( pdcf(Type,NumIn,Fault,PDCF),      % (***) choose a pdcf
        value(N,PDCF,Fault,TC),
        implication(In,TC) );
      set_in(FaultNode,Fault,TC1) ),    % else, it's input s-a-f
    drive(FaultNode,Out,TC),           % Out is the output node to detect s-a-f
    line_just([Out|L]-L,TC,FaultNode), % line justification
    !.                                  % only one test pattern is generated

implication(L,TC):-          % L contains list of lines
    fimp(L,TC),              % forward implication
    bimp(L,TC).              % backward implication

fimp([],_).
fimp([X|L],TC):-
    out(X),
    fimp(L,TC).
fimp([X|L],TC):-

```

```

connect_to(X,L1),                                % X is a input to gates in L1
consider(L1,TC),
fimp(L,TC).

consider([],_).
consider([X|L],TC):-
  value(X,L1,O,TC),
  ( (var(O),allknown(L1)) ->          % if inputs known and output unknown
    ( gate(X,Type,NumIn,_,Y),
      pc(Type,NumIn,O,L1),
      value(X,L1,O,TC),
      fimp([Y],TC) );
    true ),
  consider(L,TC).

allknown([]).
allknown([X|L]):- atomic(X), allknown(L).

bimp([],_).
bimp([X|L],TC):-
  in(X),
  bimp(L,TC).
bimp([X|L],TC):-
  gate(N,Type,NumIn,L1,X),
  value(N,L2,O,TC),
  ( allknown(L2) -> true;
    ( no_choice(pc(Type,NumIn,O,L2)) -> % if input-pattern is unique
      implication(L1,TC);
      true ) ),
  bimp(L,TC).

no_choice(X):-
  count(X,N),
  N == 1,
  X.
count(X,N):-
  assert(count(0)), X,
  retract(count(I)),
  J is I+1,
  assert(count(J)),
  fail.
count(X,N):- retract(count(N)).

set_in(N,F,[_ | L]):-
  N==1,
  M is N-1,
  set_in(M,F,L).

set_in(N,F,[F | _]):-
  N==1.

```

```

drive(Node,Out,TC):-
    ( out(Node) -> Out==Node;      % if Node is an output node; else
      ( connect_to(Node,L),      % Node is a input to gates in L
        in_list(X,L),           % (***) choose a gate to D-Propagate
        gate(X,Type,NumIn,L1,Y),
        value(X,L2,O,TC),
        pdc(Type,NumIn,O,L2),
        diff(Node,L1,L3),      % L3 is the list of other inputs
        implication(L3,TC),
        fimp([Y],TC),
        drive(Y,Out,TC) ) ).

in_list(X,[]):- fail.
in_list(X,[X|L]).
in_list(X,[_|L]):- in_list(X,L).

diff(X,[],[]).
diff(X,[X|L],L).
diff(X,[Y|L],[Y|L1]):- X==Y, diff(X,L,L1).

line_just([],[],_).
line_just([X|L]-L1,TC,N):-
    in(X),
    line_just(L-L1,TC,N).
line_just([X|L]-L1,TC,N):-
    gate(M,Type,NumIn,L2,X),
    value(M,L3,O,TC),
    ( (O==d;O==nd) ->
      ( X==N -> true;
        pdc(Type,NumIn,O,L3) );
      pc(Type,NumIn,O,L3) ), % (***) choose a pc
    diff_list(L2,L1-L4),      % convert L3 to a difference list
    line_just(L-L4,TC,N).

diff_list([],L-L).
diff_list([X|L],[X|L1]-L2):- diff_list(L,L1-L2).

value(Gate,In,Out,[[Gate,In,Out|_|_]).
value(Gate,In,Out,[[X,_,_|_|L]):- Gate==X, value(Gate,In,Out,L).

% pdcf (primitive D-cube of a logic fault) of a 2-input AND gate
pdcf(and,2,d,[1,1]).      % detect s-a-0
pdcf(and,2,nd,[0,_|_]).  % detect s-a-1
pdcf(and,2,nd,[_|,0]).    % detect s-a-1

% pc (primitive cube) of a 2-input AND gate
pc(and,2,1,[1,1]).
pc(and,2,0,[0,_|_]).
pc(and,2,0,[_|,0]).

```

% pdc (propagation D-cube) of a 2-input AND gate

```
pdc(and,2,d,[1,d]).
pdc(and,2,d,[d,1]).
pdc(and,2,d,[d,d]).
pdc(and,2,nd,[nd,1]).
pdc(and,2,nd,[1,nd]).
pdc(and,2,nd,[nd,nd]).
```

% pdcf (primitive D-cube of a logic fault) of a 2-input OR gate

```
pdcf(or,2,d,[1,_]).           % detect s-a-0
pdcf(or,2,d,[_,1]).           % detect s-a-0
pdcf(or,2,nd,[0,0]).          % detect s-a-1
```

% pc (primitive cube) of a 2-input OR gate

```
pc(or,2,1,[1,_]).
pc(or,2,1,[_,1]).
pc(or,2,0,[0,0]).
```

% pdc (propagation D-cube) of a 2-input OR gate

```
pdc(or,2,d,[0,d]).
pdc(or,2,d,[d,0]).
pdc(or,2,d,[d,d]).
pdc(or,2,nd,[0,nd]).
pdc(or,2,nd,[nd,0]).
pdc(or,2,nd,[nd,nd]).
```

% pdcf (primitive D-cube of a logic fault) of a 2-input NAND gate

```
pdcf(nand,2,nd,[1,1]).        % detect s-a-1
pdcf(nand,2,d,[0,_]).         % detect s-a-0
pdcf(nand,2,d,[_,0]).         % detect s-a-0
```

% pc (primitive cube) of a 2-input NAND gate

```
pc(nand,2,0,[1,1]).
pc(nand,2,1,[_,0]).
pc(nand,2,1,[0,_]).
```

% pdc (propagation D-cube) of a 2-input NAND gate

```
pdc(nand,2,nd,[1,d]).
pdc(nand,2,nd,[d,1]).
pdc(nand,2,nd,[d,d]).
pdc(nand,2,d,[1,nd]).
pdc(nand,2,d,[nd,1]).
pdc(nand,2,d,[nd,nd]).
```

% pdcf (primitive D-cube of a logic fault) of a 2-input NOR gate

```
pdcf(nor,2,nd,[1,_]).         % detect s-a-1
pdcf(nor,2,nd,[_,1]).         % detect s-a-1
pdcf(nor,2,d,[0,0]).          % detect s-a-0
```

% pc (primitive cube) of a 2-input NOR gate


```
pc(nor,2,0,[1,_]).
pc(nor,2,0,[_,1]).
pc(nor,2,1,[0,0]).
```

```
% pdc (propagation D-cube) of a 2-input NOR gate
```

```
pdc(nor,2,nd,[0,d]).
pdc(nor,2,nd,[d,0]).
pdc(nor,2,nd,[d,d]).
pdc(nor,2,d,[0,nd]).
pdc(nor,2,d,[nd,0]).
pdc(nor,2,d,[nd,nd]).
```

```
% pdcf (primitive D-cube of a logic fault) of an INVERTER
```

```
pdcf(inv,1,d,[0]).          % detect s-a-0
pdcf(inv,1,nd,[1]).        % detect s-a-1
```

```
% pc (primitive cube) of an INVERTER
```

```
pc(inv,1,1,[0]).
pc(inv,1,0,[1]).
```

```
% pdc (propagation D-cube) of an INVERTER
```

```
pdc(inv,1,d,[nd]).
pdc(inv,1,nd,[d]).
```

```
% pdcf (primitive D-cube of a logic fault) of a 3-input NAND gate
```

```
pdcf(nand,3,nd,[1,1,1]).    % detect s-a-1
pdcf(nand,3,d,[0,_,_]).    % detect s-a-0
pdcf(nand,3,d,[_,0,_]).    % detect s-a-0
pdcf(nand,3,d,[_,_,0]).    % detect s-a-0
```

```
% pc (primitive cube) of a 3-input NAND gate
```

```
pc(nand,3,0,[1,1,1]).
pc(nand,3,1,[0,_,_]).
pc(nand,3,1,[_,0,_]).
pc(nand,3,1,[_,_,0]).
```

```
% pdc (propagation D-cube) of a 3-input NAND gate
```

```
pdc(nand,3,nd,[1,1,d]).
pdc(nand,3,nd,[1,d,1]).
pdc(nand,3,nd,[d,1,1]).
pdc(nand,3,nd,[1,d,d]).
pdc(nand,3,nd,[d,d,1]).
pdc(nand,3,nd,[d,1,d]).
pdc(nand,3,nd,[d,d,d]).
pdc(nand,3,d,[1,1,nd]).
pdc(nand,3,d,[1,nd,1]).
pdc(nand,3,d,[nd,1,1]).
pdc(nand,3,d,[nd,nd,1]).
pdc(nand,3,d,[1,nd,nd]).
pdc(nand,3,d,[nd,1,nd]).
```

```

pdc(nand,3,d,[nd,nd,nd]).

% pdcf (primitive D-cube of a logic fault) of a 4-input NAND gate
pdcf(nand,4,nd,[1,1,1,1]).      % detect s-a-1
pdcf(nand,4,d,[0,-,-,-]).      % detect s-a-0
pdcf(nand,4,d,-,0,-,-).        % detect s-a-0
pdcf(nand,4,d,-,-,0,-).        % detect s-a-0
pdcf(nand,4,d,-,-,-,0).        % detect s-a-0

% pc (primitive cube) of a 4-input NAND gate
pc(nand,4,0,[1,1,1,1]).
pc(nand,4,1,[0,-,-,-]).
pc(nand,4,1,-,0,-,-).
pc(nand,4,1,-,-,0,-).
pc(nand,4,1,-,-,-,0).

% pdc (propagation D-cube) of a 4-input NAND gate
pdc(nand,4,nd,[1,1,1,d]).
pdc(nand,4,nd,[1,1,d,1]).
pdc(nand,4,nd,[1,d,1,1]).
pdc(nand,4,nd,[d,1,1,1]).
pdc(nand,4,nd,[1,1,d,d]).
pdc(nand,4,nd,[1,d,d,1]).
pdc(nand,4,nd,[d,d,1,1]).
pdc(nand,4,nd,[d,1,d,1]).
pdc(nand,4,nd,[d,1,1,d]).
pdc(nand,4,nd,[1,d,1,d]).
pdc(nand,4,nd,[d,d,d,1]).
pdc(nand,4,nd,[d,d,1,d]).
pdc(nand,4,nd,[d,1,d,d]).
pdc(nand,4,nd,[1,d,d,d]).
pdc(nand,4,nd,[d,d,d,d]).
pdc(nand,4,d,[1,1,1,nd]).
pdc(nand,4,d,[1,1,nd,1]).
pdc(nand,4,d,[1,nd,1,1]).
pdc(nand,4,d,[nd,1,1,1]).
pdc(nand,4,d,[1,1,nd,nd]).
pdc(nand,4,d,[1,nd,nd,1]).
pdc(nand,4,d,[nd,nd,1,1]).
pdc(nand,4,d,[nd,1,nd,1]).
pdc(nand,4,d,[nd,1,1,nd]).
pdc(nand,4,d,[1,nd,1,nd]).
pdc(nand,4,d,[nd,nd,nd,1]).
pdc(nand,4,d,[nd,nd,1,nd]).
pdc(nand,4,d,[nd,1,nd,nd]).
pdc(nand,4,d,[1,nd,nd,nd]).
pdc(nand,4,d,[nd,nd,nd,nd]).

```

A sample circuit description:

```

%% all the following predicates are circuit dependent
%% circuit description
assign([A,B,C,D,E,F,G,H,I,J,K,L], [[1,[H,I,J,K],L],[2,[B,E],H],[3,[A,F],I],
    [4,[D,F],J],[5,[C,G],K],[6,[A],E],[7,[D],G],[8,[B,C],F]]].

% describe each gate: gate(gate#, type, num_of_input, input_nodes, output_node)
gate(1,nand,4,[8,9,10,11],12).
gate(2,nand,2,[2,5],8).
gate(3,nand,2,[1,6],9).
gate(4,nand,2,[4,6],10).
gate(5,nand,2,[3,7],11).
gate(6,inv,1,[1],5).
gate(7,inv,1,[4],7).
gate(8,and,2,[2,3],6).

% describe connection: connect_to(node,gate_list)
connect_to(1,[3,6]).
connect_to(2,[2,8]).
connect_to(3,[5,8]).
connect_to(4,[4,7]).
connect_to(5,[2]).
connect_to(6,[3,4]).
connect_to(7,[5]).
connect_to(8,[1]).
connect_to(9,[1]).
connect_to(10,[1]).
connect_to(11,[1]).

% input nodes
in(1).
in(2).
in(3).
in(4).

% output nodes
out(12).

```

