1-1-1991

An analysis and implementation of linear derivation strategies

Winston M. Tabada
Edith Cowan University

AN ANALYSIS AND IMPLEMENTATION OF LINEAR DERIVATION STRATEGIES

by

Winston Membrebe Tabada

B.S.A.E., Grad. Dip. (Computer Studies), Post-Grad. Dip. (Computer Studies)

A thesis submitted in partial fulfilment of the requirements for the award of

Master of Applied Science (Computer Science)

in the School of Information Technology and Mathematics,

Faculty of Science and Technology, Edith Cowan University.

1991

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

DECLARATION BY CANDIDATE

I certify that this thesis does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any institution of higher education and that, to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature of Candidate

30 October 1990

Date

ACKNOWLEDGEMENTS

The author is sincerely grateful to his supervisor, Mr. Geoff Sutcliffe, for his guidance during the work. This work would never have been completed without the long hours he spent discussing the research with the author with such an untiring patience. His readiness to discuss and help solve even the smallest problem is very much appreciated. The author thanks Mr. John Weatherspoon for sharing his limited time in proofreading the manuscript.

Financial support has been obtained from the Australian government through the Australian International Development Assistance Bureau, and the Visayas State College of Agriculture of the Philippines. Their assistance is highly appreciated.

Special thanks is also extended to Elizabeth C. Payumo, a fellow master's student, for her encouragement and valuable suggestions. The author acknowledges her unconditional help and support throughout the research endeavor.

Last , but not the least, the author expresses his sincere appreciation to his loving wife, Tess, who has shouldered the responsibility of being both mother and father to two kids, Abraham and Paul, during the author's absence. Their patience, understanding, and assurances provided a great source of strength to the author to keep on.

ABSTRACT

An Analysis and Implementation of Linear Derivation Strategies

by

Winston Membrebe Tabada

This study examines the efficacy of six linear derivation strategies : (i) s-linear resolution, (ii) the ME procedure, (iii) t-linear resolution, (iv) SL-resolution, (v) the GC procedure, and (vi) SLM. The analysis is focused on the different restrictions and operations employed in each derivation strategy. The selection function, restrictive ancestor resolution, compulsory ancestor resolution on literals having atoms which are or become identical, compulsory merging operation, reuse of truncated literals, spreading of FALSE literals, no-tautologies restriction, no two non-B-literals having identical atoms restriction, and the use of semantic information to trim irrelevant derivations from the search tree are the major features found in these six derivation strategies. Detecting loops and minimising irrelevant derivations are the identified weak points of SLM. Two variations of SLM are suggested to rectify these problems.

The ME procedure, SL-resolution, the GC procedure, SLM and one of the suggested variations of SLM were implemented using the Arity/Prolog compiler to produce the ME-TP, SL-TP, GC-TP, SLM-TP and SLM5-TP theorem provers respectively. In addition to the original features of each derivation strategy, the following search strategies were included in the implementations : the modified consecutively bounded depth-first search, unit preference strategy, set of support strategy, pure literal elimination, tautologous clause elimination, selection function based on the computed weight of a literal, and a match

check. The extension operation used by each theorem prover was extended to include subsumed unit extension and paramodulation.

The performance of each theorem prover was determined. Experimental results were obtained using twenty four selected problems. The performance was measured in terms of the memory use and the execution time. A comparison of results between the five theorem provers using the ME-TP as the basis, was done. The results show that none of the theorem provers consistently performs better than the others. Two of the selected problems were not proved by SL-TP and one problem was not proved by SLM-TP due to memory problems. The ME-TP, GC-TP and SLM5-TP proved all the selected problems. In some problems, the ME-TP and GC-TP performed better than SLM5-TP. However, the ME-TP and GC-TP had difficulties in some problems in which SLM5-TP performed well.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION AND OVERVIEW

1.1. Rationale

Mechanising theorem proving is a major field of endeavor in Artificial Intelligence. The interest in theorem proving stems from the ability of theorem provers to emulate many tasks associated with human intellect. Tasks that require human intuition such as question answering, general problem solving, writing programs and some robotic applications can now be automated using a theorem prover (Green 1981). Theorem provers may play a vital role in the fields of mathematics and mathematical logic. In mathematical logic, one can express conveniently almost all kinds of deductive arguments, and this allows mechanical manipulation by a theorem prover. Mathematicians can now study deduction in its purest form with the aid of an automatic theorem prover. The realisation that powerful theorem proving techniques could provide a key component of many "intelligent machines" has drawn many computer scientists and mathematicians to the computer rooms to implement theorem provers.

An important milestone in automatic theorem proving research was the introduction of the resolution principle by Robinson (1965). Green (1981, p. 202) asserted that "automatic theorem proving using the resolution proof procedure represents perhaps the most powerful known method for automatically determining the validity of a statement of first-order logic". The resolution proof procedure is complete, that is, it is able to prove everything that is actually true, and can be easily mechanised. However, the cost of completeness is a combinatorially large search space. One of the disadvantages of using the resolution principle in an unrestricted manner is that it leads to many redundant and irrelevant inferences. An inference is redundant when the result of the inference can be derived in other ways while an inference is irrelevant when the result of the inference does

not lead to a derivation of the desired goal. The vast number of clauses generated before a proof is found is one of the outstanding problems in theorem proving using the resolution principle. The completeness property of the resolution principle is only of theoretical interest if the search problems are not solved. Hence, considerable research effort has gone into refinements of the resolution proof procedure. Some of these research efforts have been directed to the linearisation of the resolution proof procedure.

The resolution proof procedure repeatedly selects pairs of clauses which can be resolved and derives new clauses by the resolution rule until the empty clause is derived. The derived non-empty clauses, called intermediate clauses, are added to the original set of clauses. An unrestricted resolution proof procedure derives much of the redundancy in its search space from the resolution of intermediate clauses with other intermediate clauses, which causes the combinatorial explosion of the search space. Linear resolution minimises this redundancy by restricting the selection of pair of clauses for resolution. Linear resolution requires that one of the selected pair of clauses must be the most recently derived clause (starts with a top clause which can be any of the input clauses), and the other clause must be an input clause or an ancestor of the first parent. The most recently derived clause is referred as the *center clause* while the other parent clause is referred as the *side clause*. Linear resolution has a relatively uncomplicated structure of its search space which makes a heuristic search easy to apply. Linear resolution also suits question-answering applications better than unconstrained resolution (Brown 1974).

To restrict derivations to be linear, however, has been found to be insufficient to control the size of the search space. Much work has been done to produce a restricted linear derivation strategy. Loveland (1968) formalised two restricted linear derivation strategies : s-linear resolution and the Model Elimination (ME) procedure. Kowalski and Kuehner (1971) built on the work of Loveland by adding more restrictions and the factoring operation. These refinements were formalised as t-linear resolution and SL-resolution. Shostak (1976) refined the ME procedure by devising the C-literal mechanism

as a substitute for the lemma scheme of the ME procedure. This work is formalised as the Graph Construction (GC) procedure. Brown (1974) argued that the ability to select dynamically a literal to resolve on is critical in obtaining a refutation as well as controlling the size of the search space. The ME procedure, SL-resolution and the GC procedure have limited choices of literal to resolve upon. In view of this, Brown formulated the Selective Linear Model (SLM) linear inference system which he claimed to be superior than the other derivation strategies in terms of its flexibility in the choice of literal to resolve on. Each of these works has features distinct from the others. However, it is not known, which of these derivation strategies presents a better and more efficient theorem prover in terms of memory use and speed. To know this, a comparison of the efficacy of each derivation strategy needs to be done. In addition, there is a need to analyse their different features, and to implement and test them as theorem provers. Hence, this study has been conducted.

1.2. Objectives of the Study

This study aims to attain the following objectives :

1. To provide an insight into the features of the following linear derivation strategies, through analysis of:
   a. s-linear resolution
   b. the ME procedure
   c. t-linear resolution
   d. SL-resolution
   e. the GC procedure
   f. SLM

2. To formulate possible extensions to the SLM derivation strategy

3. To implement theorem provers based on the following linear derivation strategies:
   a. the ME procedure
   b. SL-resolution
   c. the GC procedure
   d. SLM
   e. SLM with any extensions formulated.

4. To compare the efficacy of the implemented theorem provers.

1.3. Overview of Automated Reasoning Strategies

Griffiths and Palissier (1987) describe, in brief, the basis of automatic theorem proving as follows :

> Automatic theorem proving follows research on logic and the validity of proofs, which is particularly ancient. At the end of the seventeenth century, Leibniz was already looking for an algorithm to prove or refute formulae. The modern era dates from Herbrand (1930), who gives an algorithm of this kind for formulae in first order logic. It is this logic, called predicate calculus if it contains no additional specific axioms, that is the basis of today's theorem provers. (p. 63)

Earlier works by Newell, Simon, Shaw and Gelernter in the middle and late 1950s emphasized the heuristic approach to problem solving, but soon shifted to various syntactic methods culminating in increased research on resolution type systems (Bledsoe 1981). Since the development of resolution, many refinements have improved its efficiency. The following are reviews of some of the major works on the refinements of resolution, which relate to linear derivation strategies:

1.3.1. P_1 and N_1 Resolution (Meltzer 1966, Robinson 1979)

P_1 resolution, as described by Stickel (1986, p. 86), is a restricted application of resolution which requires that one of the parent clauses must be a positive clause. N_1 resolution requires that one of the parent clauses must be a negative clause, which is the inverse of P_1. These two resolution proof procedures are closely related to the set of support strategy (described in the next section). In fact, Stickel viewed P_1 resolution as an extension of the set of support strategy. He supported this view by the following arguments :

> Using the set of support restriction, it is legitimate to designate the set of all positive clauses as the set of support. Resolution operations between input clauses will then require one parent to be a positive clause as desired. However, with just the set of support restriction, any derived clause can be resolved with any other clause and the intended restriction that one of the parent clauses to each resolution operation must be positive will not be obeyed. After each resolution operation, the resulting set of clauses is unsatisfiable provided the initial set of clauses is unsatisfiable. Thus, the set of support restriction (with the set of all positive clauses designated as the set of support) can be applied to each set of clauses resulting after performing a resolution operation and not just to the initial set of clauses, effectively imposing the desired restriction that one parent clause of each resolution operation be a positive clause. (p. 86)

N_1 resolution is closely related to the linear input resolution. Given a set of Horn clauses, a linear input derivation which starts with a negative top clause is also an N_1 derivation.

1.3.2. Hyper-resolution (Robinson 1965)

Unlike ordinary resolution that requires two clauses for each application, hyperresolution uses an arbitrary number of clauses in each inference step. Each hyperresolution operation takes a single mixed or negative clause, referred to as the *nucleus*, and a number of positive clauses, referred to as *electrons*, which correspond to the number of negative literals in the nucleus. To produce a positive clause, each negative literal of the nucleus is resolved with a literal in one of the electrons. The derived clause, referred to as the *hyperresolvent*, consists of all the positive literals of the nucleus and the unresolved on literals of the electrons (Stickel 1986, Wos et al. 1984). Hyperresolution is a generalisation of the P_1 resolution. Negative hyperresolution interchanges the roles of positive and negative clauses in hyperresolution (Wos et al. 1984, p. 168).

A successful application of hyperresolution can be viewed as applying linear resolution with the nucleus as the top clause, until the derived clause is positive. However, the entire process does not yield intermediate clauses.

1.3.3. Linear Input and Unit resolution (Chang 1970)

Linear input resolution is a restricted resolution proof procedure which requires that for every resolution, one of the parent clauses is the most recently derived clause, and the other is an input clause. If clauses are viewed as lists (not sets) of literals, every resolution involves binary resolution and merging operation. Chang (1970, p. 703) noted that a linear input derivation involves binary resolution and merging (Andrews 1968). Linear input resolution is only complete for sets of Horn clauses. The completeness of linear input resolution for Horn clauses shows that it is unnecessary to resolve arbitrary pairs of input clauses with each other, because it is sufficient to take only those pairs of clauses that include a negative clause. A restricted form of linear input resolution, called *ordered input resolution* is the basis of PROLOG. Ordered input resolution requires that literals be resolved away in some fixed order such as strictly left to right, as used in PROLOG. In ordinary linear input resolution, the literals of the top

clause can be resolved away in any order to obtain an empty clause. If there are N literals in the top clause then there would be N! derivations of the empty clause. This inefficiency is eliminated in ordered input resolution. This technique is used in the ME procedure, SL-resolution, the GC procedure and SLM derivations.

Unit resolution is a resolution proof procedure which requires that at least one of the parent clauses is a unit clause. A unit clause is a clause that contains one literal only. Chang has shown that if a theorem has a linear input proof then it also has a unit proof. Henschen and Wos (1974, p. 590) pointed out that "many in the field of automated theorem proving have devoted much effort and interest to the advantages, properties, and implementation of techniques which give preference and often exclusion to unit inference." The advantage of this derivation strategy is that whenever a clause is resolved with a unit clause, the result has fewer literals than the parent does. This helps to focus the search toward obtaining a refutation and thereby improves efficiency. However, like linear input resolution, unit resolution is only complete for sets of Horn clauses.

1.3.4. UR-resolution (McCharen et al. 1976)

McCharen et al. (1976) formulated an inference system known as Unit-Resulting (UR) resolution, which they claimed to be an improved version of unit resolution. UR-resolution is an inference rule, similar to hyperresolution, which combines a series of binary resolution steps into one step. Where hyperresolution requires a single mixed or negative clause and a set of positive clauses, UR resolution requires a nonunit clause and a set of unit clauses to produce a unit clause or an empty clause. Unit resolution produces an intermediate clause for every successful resolution and some of these intermediate clauses may pollute the search space. UR-resolution minimises this problem by combining a series of unit resolution steps into one step, thus, yielding no intermediate clauses. However, UR-resolution is only complete for Horn clauses.

A successful UR-resolution can be viewed as applying linear input resolution with the non-unit clause as the top clause, and each step uses a unit clause as the other parent. The entire process, however, does not yield intermediate clauses.

1.3.5. Connection Graphs (Kowalski 1975)

Connection graphs were first proposed by Kowalski (1975). Other authors who have used different forms of connection graphs are Sickel (1976), Chang and Slagle (1979), and Stickel (1982) as cited by Genesereth and Nilsson (1987). In brief, Amble (1987, p. 42) describes a connection graph as an - "implementation strategy for resolution, using a network of links between resolvable clauses. Links are selected for resolution and deleted afterwards; new clauses are linked into the graph; and clauses with no links are deleted." He added that connection graphs allow any search strategy to be implemented and their application in expert systems and parallel inference machines is a promising research topic. Ramsay (1988, p. 102) conjectured that "connection graphs seem in some sense to be the last word in resolution theorem proving". However, he mentioned that connection graph resolution is unnatural. Traces of connection graph theorem provers at work are almost impossible to follow, unlike the linear type of derivation.

There is some similarity between the Connection Graph with the Graph Construction of Shostak. In Graph Construction, clauses are linked with other clauses to form the refutation graph. However, its main purpose is to be able to analyse pictorially various resolution strategies. In the case of the Connection Graph, clauses are linked to form a connection graph which is a data structure to facilitate the encoding of inference operations.

1.4. Overview of General Search Space Restrictions

Most research works on theorem provers based on the resolution principle are concerned with devising good search strategies. However, Ramsay (1988, p. 86) asserted that "there is no universal agreement as to which are the best strategies ... the choice will depend on characteristics of the kind of problem being solved as much as on the general properties of resolution." Resolution has no inherent search strategy that can be fixed independently. Experimental evidence, as described by Wos (1988), has shown that automating a resolution proof procedure without good search strategy has problems in terms of computer memory requirements and time to obtain a proof. To solve these problems, one should understand their causes. Wos (1988, pp. 21-43) describes major obstacles in the automation of theorem proving which are the main focus of most research on theorem proving. Some of them are summarised as follows (those not included are beyond the scope of this study):

1. **Clause retention.** The reasoning program (theorem prover) keeps too many deduced clauses (too many conclusions) in its database of information. Retention of so much unneeded information is harmful to the effectiveness of an automated reasoning program. When a reasoning program retains a large number of unneeded clauses, this extra information interferes markedly with the program's effectiveness by causing the program to waste too much time.

2. **Inadequate focus.** The program's reasoning is not sufficiently well directed. When a reasoning program is asked to complete some given assignment, it immediately begins drawing conclusions. Unfortunately, the program too easily gets lost, pursuing one unprofitable path after another. A reasoning program could provide even greater assistance if it could choose wisely the clauses from which to draw conclusions.

3. **Redundant information.** The reasoning program generates the same clause (or proper instances of clauses already generated) over and over again. A clause is redundant if it is a copy of a second clause, or if it is a proper instance of a second clause, that is obtainable from a second clause by substituting appropriate terms for variables. Deducing a clause of either type serves no purpose.

4. **Clause generation.** The program draws too many conclusions, many of which are redundant and many of which are irrelevant even though they are not redundant.

5. **Size of the deduction steps.** The inference rules do not take deduction steps (inference steps to obtain refutation) of the appropriate size. If the steps are too small, then the program can generate too many clauses, and, most likely, the program will retain, too many in its database. The result can be an inordinate waste of computer time to complete a given assignment. To always obtain a proof with a minimum number of inference steps, one has to use a breadth-first search strategy which requires a lot of memory and computation time. If the size of the deduction step is too large, then the program can bypass the needed information and, therefore, fail to complete the assignment.

With all the research works done on devising good search strategies, different types of search strategies have evolved that can be applied to control the search space of derivation strategies that use the resolution principle. Wos et al. (1984) categorises the different search strategies into groups : the ordering, restriction and pruning strategies. A search strategy is classified as an ordering strategy if the inference steps are directed in their choice of which clause to focus on next. When a derivation is prevented from using certain combinations of clauses, then the search strategy used is classifed as a restriction strategy. A pruning strategy is a strategy to remove redundant clauses. The following are reviews of some known search strategies which are used with linear derivation strategies:

1.4.1. Unit Preference Strategy (Wos et al. 1964)

The unit preference strategy tries to deduce clauses with as few literals as possible. The point of this strategy is that short clauses are easier to work with. A clause is a unit if it contains a single literal. The unit preference strategy, as the name implies, gives more priority to a resolution involving a unit. The goal of theorem proving is to derive the empty clause, hence, clauses with fewer literals are closer to the goal than clauses with more literals. The unit preference strategy loosens the restriction of unit resolution to allow non-unit clauses be selected, but shorter clauses are used first. The advantages of unit resolution also apply to the unit preference strategy. The unit preference strategy may have a larger search space than unit resolution, however, it is complete for general clauses.

The unit preference strategy plays an important role in the implementation of the ME procedure, SL-resolution, the GC procedure, SLM and its extension.

1.4.2. Set of Support Strategy (Wos et al. 1965)

An unsatisfiable set of clauses S can be subdivided into two subsets : a satisfiable set of clauses (S-T) and the set of support (T). Wos et al. (1965) have proven that if S is unsatisfiable, then a refutation can be derived by a sequence of resolutions in which at least one of the parent clauses of each resolution is a member of the set of support. The derived clause of each resolution is placed in the set of support. This strategy is known as the *set of support strategy*. Kowalski and Kuehner (1971, p. 232) described possible sets of support as the set of all positive clauses, the set of all negative clauses, or the set of all clauses that were derived from the negation of the conclusion of the theorem. The advantageous effect of the set of support strategy is that it avoids resolutions between parent clauses which both belong to a set of satisfiable clauses. This restriction can reduce the size of the search space significantly. Wos (1988, p. 52) claimed that the set of support strategy "is currently considered the most powerful restriction strategy available". Wos et al. (1965) have also shown that a strategy which

combines the set of support strategy and the unit preference strategy is often much more efficient than the unit-preference strategy alone, as cited by Slagle (1971).

Supposing the selected top clause C_1 of a linear derivation is a member of the set of support, then the resolvant of a resolution step between C_1 and a side clause is placed in the set of support. Since one of the parent clauses in each step of linear derivation is the previously derived clause, every derived clause is placed in the set of support. Thus, each step of a linear derivation satisfies the restriction of the set of support strategy. Hence, a linear derivation strategy is a special case of the set of support strategy. Loveland (1968, p. 161) showed that there always exists an s-linear refutation with the top clause chosen from the set of support. Restricting the selection of the top clause to be always from the set of support is advantageous because it will limit the number of search trees which need to be investigated in the course of searching for a refutation.

The SL-resolution, GC procedure and SLM explicitly require the set of support in the definition of their derivations. The ME procedure also satisfies the set of support strategy (Loveland 1969a), thus, the set of support strategy can be incorporated in the implementation of the ME procedure.

1.4.3. Purity Elimination

A literal is called a *pure literal* if its complement (subject to unification) does not appear in a set of clauses. A clause that contains a pure literal is useless for the purpose of refutation since the literal can never be resolved away. Hence, such a clause may be removed from a set of clauses. To remove such a clause is called pure-literal elimination. This strategy is used in the implementations of the ME procedure, SL-resolution, the GC procedure, SLM and its extension.

1.4.4. Elimination of Tautologies

A clause which contains a pair of exactly complementary literals is a tautology. The presence or absence of tautologies in a set of clauses has no effect on the unsatisfiability of the set. Hence, they may be removed from the set of clauses. The ME procedure, SL-resolution, the GC procedure and SLM provide restrictions that in effect prevent the use of tautologous input clauses. These restrictions are included in the implementations.

1.4.5. Subsumption

"Subsumption is the process for discarding a clause that duplicates or is less general than another clause available" (Wos et al. 1984, p. 171). A clause A subsumes a clause B if there exists a substitution θ such that $A\theta$ is a subset of B. If a clause in a set of clauses is subsumed by another clause in the set, then the set remaining after eliminating the subsumed clause is unsatisfiable if and only if the original set of clauses is unsatisfiable (Genesereth and Nilsson 1988). There are two forms of subsumption that can be employed. The elimination of a newly derived clause that is subsumed by a clause that is already present is called forward subsumption. Discarding of clauses already present that are subsumed by a newly derived clause is called backward subsumption (Stickel 1987, p. 84). The idea of subsumption is to prevent the use of subsumed clauses in the selection of pairs of clauses for resolution because they will only expand the search space unnecessarily. Subsumption is used in a different way in the *subsumed unit extension* operation (Sutcliffe 1989). Subsumed unit extension is included in the implementations of the ME procedure, SL-resolution, the GC procedure, SLM and its extension.

1.4.6. Weighting (Wos et al. 1984)

Wos et al. (1984) describe weighting as -"the process for assigning priorities to terms, clauses, and concepts". It can be used to reflect knowledge and intuition about

how a derivation should proceed. Weighting is both an ordering and a restriction strategy. A weighting strategy is used in the selection of literals to resolve on in the implementations of the ME procedure, SL-resolution, the GC procedure, SLM and its extension.

1.4.7. Consecutively Bounded Depth-first Search

Stickel (1986) pointed out that the use of an unbounded depth-first search strategy is incomplete as a search strategy for theorem proving. Breadth-first search and the A* algorithm (Nilsson 1980) are some of the suggested complete search strategies for theorem proving. However, the use of such search strategies for theorem proving have disadvantages. One of these disadvantages is the increase in memory requirements. A derivation strategy using these search strategies would have to represent and retain more than one derived clause at once. An alternative complete search strategy known as the consecutively bounded depth-first (Stickel and Tyson 1985) also known as depth-first iterative-deepening search (Korf 1985), has been suggested for automated theorem proving. Consecutively bounded depth-first search strategy involves repeatedly performing an exhaustive bounded depth-first search, with increasing depth bounds. This search strategy has minimal memory requirements and finds optimal solutions as efficiently as breadth-first or A* search, in spite of the effort spent on repeated search (Nie and Plaisted 1989). A modified version of consecutively bounded depth-first search is used in the implementations of the ME procedure, SL-resolution, the GC procedure, SLM and its extension.

1.5. Thesis Structure

The rest of this thesis is arranged as follows:

Chapter 2 investigates the inference operations and different restrictions imposed in s-linear resolution, the ME procedure, t-linear resolution, SL-resolution, the GC procedure and SL, and proposes two possible extensions to SLM;

Chapter 3 describes the data structures, self-configuration, search strategies used, and algorithms in the implementations of the ME procedure, SL-resolution, the GC procedure, SLM and SLM-5. Descriptions of each program and its user-interface are also presented;

Chapter 4 presents a comparison of experimental results from the implemented theorem provers;

Chapter 5 presents conclusions with suggestions for future work;

Appendix A contains the general algorithms of the implemented derivation strategies;

Appendix B contains the source codes of the programs with comments;

Appendic C contains the description of how to operate the theorem provers; and

Appendix D contains the example theorems used to test the theorem provers.

Chapter 2

ANALYSIS

2.1. Introduction

Robinson (1965) proved that a method of inference called *resolution* is a complete strategy for proving the unsatisfiability of a set of first order clauses. Since the introduction of the resolution principle, many researchers have attempted to mechanise theorem proving. However, although resolution has reduced the number of inference rules, the combinatorial explosion of the search space caused by the unrestricted application of resolution is a great hindrance. There is a great need to prune the search space of derivations for resolution to be practical. Hence algorithms that selectively choose resolutions, known as *refinement strategies* , have been formulated. Refinement strategies serve as guides for automated theorem provers that compute only a restricted set of all possible resolutions. Hunt (1975, p. 303-304) classifies three general classes of refinement strategies: syntactic, semantic and ancestory strategies. A syntactic strategy chooses clauses for resolution based on the structural properties of the clauses themselves, without regard to the interpretation of the atoms in the clause. These strategies are relatively easy to implement because they depend only on a structural examination of the clauses potentially involved in a resolution. However, syntactic strategies do not eliminate any redundant derivations from the search space. In contrast, semantic strategies select clauses which are known to have a certain truth value under a certain interpretation. These strategies are selective in applying resolution to clauses, which will result to the reduction of the search space. Ancestory strategies select clauses for further resolution based upon the history of the derivation of the clauses so selected. Ancestory strategies permit savings in both the number of resolutions to be considered and the amount of computer storage required for recording clauses that have been inferred (Hunt 1975, p. 308). Linear derivation strategies fall into this category.

2.1.1. Definition of Linear Resolution

Linear derivation strategies were independently formulated by Loveland (1970), Luckham (1970), and Zamov and Sharonov (1969). Kowalski and Kuehner (1971, p. 231) contend that a linear derivation strategy is a refinement of unrestricted resolution which reduces significantly the number of redundancies derivable.

In general, a *linear derivation* D from a set of clauses S is a sequence of clauses $C_1, ..., C_n$ such that each C_{i+1}, $1 \leq i \leq n-1$, is a resolvant of C_i *(center parent clause)* and B *(far parent clause)* where either (a) $B \in S$ *(input parent)*, or (b) B is some ancestor C_j of C_i, $j < i$ *(ancestor parent)*. $C_1 \in S$ is the *top clause* of D and C_n is the clause derived by D. If C_n is the null clause then D is a *linear refutation* of S. C_{i+1} is obtained in case (a) by *input resolution* while case (b) is by *ancestor resolution.*

A linear derivation strategy known as *input derivation* restricts the far parent to be always from the set of clauses S. This strategy has been proven complete for Horn-clauses, but is incomplete for non-Horn-clauses (Henchen 1976). The linear derivation strategies presented in this study allow ancestor resolution and have all been proven sound and complete for general clauses.

The following six linear derivation strategies were analysed :

1. s-linear resolution (1968)
2. Model Elimination (ME) procedure (1969)
3. t-linear resolution (1971)
4. SL-resolution (1971)
5. Graph Construction (GC) procedure (1976)
6. Selective Linear Model (SLM) (1974)

The different restrictions imposed by these linear derivation strategies were investigated. The relationships between the strategies and their relative efficacies were also examined.

2.1.2. Representation of Clauses and Input Set Manipulation

A clause is a disjunction of literals and a set of clauses S is interpreted as a single statement which is the conjunction of all its clauses. The equality axiom of symmetry, for instance, is expressed in clausal form as equal(X,Y) v ~equal(Y,X), where "v" indicates disjunction and "~" indicates negation. The linear derivation strategies presented here use two different formats for representing a clause.

The s-linear and t-linear strategies represent a clause as a *set* of literals. Thus, the clause P v ~Q v ~R is represented as {P,~Q,~R}. In this representation scheme, when resolution is applied between two clauses, the resolvant clause must contain distinct literals. To satisfy this set constraint, a merging operation must be done implicitly together with the resolution. Thus, the resolvant of {P,Q,R} and {P,Q,~R} is {P,Q} as a result of one resolution and two merging operations.

The ME, SL-resolution, GC procedures and SLM represent a clause in *chain* format. A chain is a sequence of literals, which are not necessarily distinct. There are three classes of literals that may occur in a chain : (i) B-literals, (ii) A-literals which are denoted by boxed literals or literals embedded in [] brackets, and (iii) C-literals denoted by circled literals. B-literals are derived from the literals in input clauses. The most basic chain structure is the *elementary* chain. An elementary chain is a sequence of B-literals, determined by assigning an ordering to the literals of a clause. In the ME procedure, SL-resolution and GC procedure, an A-literal is derived from the resolved upon B-literal of a center parent chain after an input resolution. A C-literal, which is only used in the GC

procedure, is the complement of a deleted A-literal. A cell is defined such that two B-literals belong to the same cell iff they are not separated by an A-literal.

The ME procedure generates *matrix* chains from an input clause. Matrix chains (which are also elementary chains) are formed by ordering the set of literals of a clause in such a way that each literal of the clause is the first literal of a chain, and the remaining literals are ordered according to some convenient rule. Hence, the clause :

$$C = P \vee \sim Q \vee R$$

will form the matrix chains :

$$m_1 = P \ \sim Q \ R$$

$$m_2 = \sim Q \ R \ P$$

$$m_3 = R \ P \ \sim Q$$

All literals in the matrix chain are classified as B-literals.

In SL-resolution, each input clause is factored. For each factor produced a chain is formed in the same manner as ME's matrix chains. For example, the clause :

$$C = \sim p(X) \vee \sim p(a)$$

has two factors :

$$C_1 = \sim p(X) \vee \sim p(a)$$

$$C_2 = \sim p(a)$$

The first factor will generate two matrix chains :

$$m_1 = \sim p(X) \ \sim p(a)$$

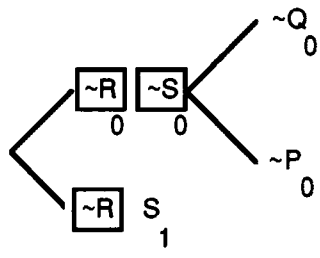
$$m_2 = \sim p(a) \ \sim p(X)$$

and the second factor will generate one matrix chain :

$$m_3 = \sim p(a)$$

In the GC procedure, each input clause is only converted to a sequence of literals to form an elementary chain.

In SLM, an input clause is converted into a sequence of B-literals to form a chain. Furthermore, each B-literal of the input chain has an assocaited truth index whose value is either 0 or 1 (representing false and true respectively), as determined by a given interpretation. A center chain of an SLM derivation is composed of a root node followed by zero or more subnodes. A subnode may also contain subnodes. To illustrate, consider the following figure of an SLM center chain :



The rootnode of the center chain is null. There are two subnodes of the root node. The upper subnode, whose contents are $[\sim R_0]\,[S_1]$, also has two subnodes. A subnode is said to be a tip node iff it has no subnodes. Subnodes $\sim Q_0$, $\sim P_0$ and $[\sim T_0]\,S_1$ are tip nodes. A branch of a center chain is a sequence of nodes from the root node to a tip node. The subscript of each literal in the center chain represents the truth index of the literal.

The line connecting the A-literal $[S_1]$ and the root node represents the depth associated with the A-literal $[S_1]$. A depth is a position within the center chain which is to the left of the associated A-literal, and must be in the same branch. Only A-literals indexed by 1 have an assocaited depth.

2.2. s-linear Resolution

Loveland (1968a) formulated a restricted linear derivation strategy called the s-linear strategy. The s-linear strategy does not only allow ancestor resolutions but also restricts the selection of ancestor clause to be used in ancestor resolution. The strategy ensures that the resolvant of an ancestor resolution always subsumes an instance of the ancestor parent clause. This restriction reduces the search space and simplifies the derivation. The strategy also eliminate redundant derivations by implementing a no-tautologies restriction.

2.2.1. Formal Definition

A derivation $C_1, ..., C_n$ from the set of clauses S is an *s-linear* derivation iff restrictions i), ii) and iii) are satisfied.

i) The sequence $C_1, ..., C_n$ is a linear derivation.

ii) C_{i+1} is obtained from C_i by applying resolution with either :

   (a) a clause from S, or

   (b) a clause C_j, $j < i$, chosen so that the resolvant C_{i+1} subsumes an instance of C_i.

iii) No tautology occurs in the derivation.

If C_n is the null clause then the derivation is an s-linear refutation.

2.2.2. Example Problem

To demonstrate how s-linear resolution works, consider the propositional problem found in Kalish and Montague (1964) as cited by Pelletier (1982). The set of clauses S = { {P, Q, R}, {P,Q, ~R}, {P,~Q,R}, {~P,Q,R}, {P,~Q,~R}, {~P,~Q,R}, {~P,Q,~R}, {~P,~Q,~R} } represents the problem. Using the set of support strategy, the clause that contains all positive literals is chosen as top clause. The search tree of the s-linear refutation of S with {P,Q,R} as the top clause is shown in Figure 1. The circled clauses are the permissible clauses, others being rejected as tautologies. The double arcs drawn below a center chain represent the different inference options. The ancestor parent clauses are highlighted to distinguish them from the input parent clauses.

2.2.3. Effects of the Restrictions

It is known that ancestor resolution is required for the completeness of linear derivation strategies. However, allowing ancestor resolution also expands the search tree due to the increased number of possible parent clauses. The s-linear derivation strategy minimises this effect by imposing the restriction (ii.b) which restricts the selection of an ancestor parent clause for ancestor resolution. This restriction reduces the number of inference steps in a refutation since every ancestor resolution produces a resolvant with fewer literals than its center parent clause. Thus, this restriction has increased the efficacy of the derivation strategy.

As shown in Figure 1, there are twelve possible resolvants that can be derived from the top clause {P,Q,R}. With the imposition of restriction (iii) , the permissible resolvants are reduced to three. This is a significant reduction of the search space. A refutation can of course be derived even if tautologous resolvants are allowed. However,

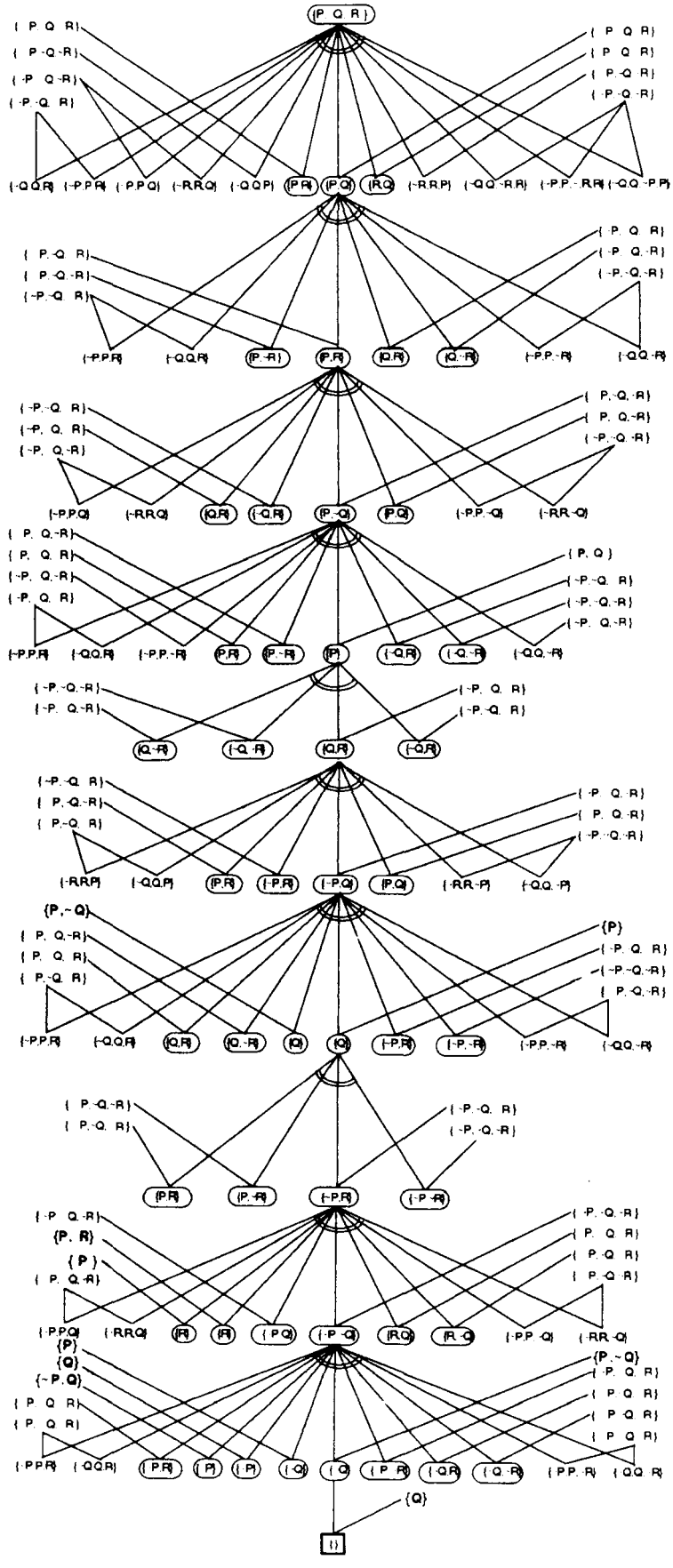


Figure 1. The search tree for S = {{P, Q, R}, {P,Q, ~R}, {P,~Q,R}, {~P,Q,R}, {P,~Q,~R}, {~P,~Q,R}, {~P,Q,~R}, {~P,~Q,~R}} using s-linear resolution.

derivations containing tautologous clauses will tend to be longer and in the extreme are redundant. Loveland (1968a, p. 156) has proven that a minimal refutation of S contains no tautologies. Allowing tautologies, though, will lessen the constraints in selecting a top clause from the set of clauses. Take, for instance, the set of clauses S = { {P,Q}, {~P,~Q}, {P}, {Q} }. With the "no-tautologies" restriction, no s-linear refutation exists when {P,Q} is chosen as the top clause although a refutation exists if tautologies are allowed.

2.3. Model Elimination Procedure

In 1968, Loveland introduced a new proof procedure which is a Herbrand-type procedure, and uses the matching technique employed by Prawitz (1960) in his mechanical theorem proving. This new procedure was labelled Model Elimination (Loveland, 1968b, p. 236). The procedure seeks the truth-functionally contradictory clause associated with the Herbrand procedures by developing clauses which contradict the assigned truth values over its atomic components (Loveland 1968b, p. 236). A TRUE value is assigned to every resolved-upon literal of the center clause. A number of these truth-assigned literals represent a model. If this model contradicts an instance of one of the input clauses, then this model will be eliminated by removing the literals that define the model. (See section 2.3.3 for details.) Hence the name, "Model Elimination".

Loveland realised that the original Model Elimination procedure he developed was too complex from a data-handling viewpoint, and somewhat cumbersome. In view of this, he made some modifications to the procedure that led to the formulation of a simplified version of the Model Elimination procedure (Loveland 1969b). This new version of the Model Elimination procedure extends the original version by the production of new clauses, called *lemmas* , during the derivations. These lemmas may be added to the original set of matrix chains. It is this simplified version of Model Elimination that this study discusses.

2.3.1. Formal Definition

For a given set of clauses S, a set of matrix chains M is derived. A derivation $C_1,..., C_n$ is said to be a Model Elimination procedure derivation if restrictions i), ii) and iii) are satisfied.

i) $C_1, ..., C_n$ is a linear derivation and $C_1 \in M$.

ii) Each C_i is a preadmissible chain. A chain is preadmissible if :

a) two B-literals are complementary they must be separated by an A-literal,

b) a B-literal is identical to an A-literal, the B-literal must precede the A-literal in the chain, and

c) no two A-literals have identical atoms.

An admissible chain is a preadmissible chain whose last literal is a B-literal. An empty chain is an admissible chain.

iii) C_{i+1} is obtained from C_i by :

a) extension

b) reduction, or

c) contraction

C_{i+1} is obtained from C_i by **extension** with an elementary chain B iff :

a) C_i is an admissible chain.

b) C_i and B share no variables. This can be achieved by a simultaneous replacement of all n variables of C_i by the variables $x_1, ..., x_n$ and of all m variables of B by the variables $y_1, ..., y_m$.

c) the last literal L of C_i and the complement of the first literal K of B are unifiable with the most general unifier mgu θ. That is $L\theta = \sim K\theta$.

d) θ is applied to the result of concatenating C_i and B minus K,

e) $L\theta$ is designated an A-literal with *scope* 0. Every other literal in C_{i+1} has the same classification as the literal from which it descends in C_i or B. The scope is a non-negative integer associated with A-literals, and is used as an aid in the production of lemmas.

C_{i+1} is obtained from C_i by **reduction** iff :

a) C_i is an admissible chain.

b) there exists an A-literal ~K which preceeds a B-literal L where L and K are unifiable with mgu θ.

c) $L\theta$ is deleted from $C_i\theta$, $C_{i+1}=(C_i - L)\theta$. All literals of C_{i+1} have their parent classification except $\sim K\theta$. Let m be the scope of ~K in C_i. If the number of A-literals n between ~K and L is greater than m, then the scope of $\sim K\theta$ in C_{i+1} is n, otherwise the scope is m.

C_{i+1} is obtained from C_i by **contraction** iff:

a) C_i is a pre-admissible chain and not an admissible chain.

b) all A-literals beyond the last B-literal are deleted. A-literals are removed one at a time. As an A-literal L is removed, a lemma is formed consisting of the complement of L plus the complements of any preceeding A-literal K of C_i such that the number of A-literals (strictly) between K and L is less than the scope of K. The lemma is added to M unless subsumed by a member of M. (Subsumption was only mentioned on Loveland's (1969a, p. 75) second paper on the ME procedure.) Each A-literal L in C_{i+1} has the same scope as its parent A-literal in C_i unless the scope of an A-literal L exceeds the number of

A-literals n beyond A-literal L in C_{i+1}. If this occurs, the scope is reduced to n.

Loveland has proven that each lemma produced by the ME procedure is a clause deducible by the resolution procedure from the same input clauses. Robinson's theory of resolution as cited by Loveland (1969a, p. 77), has established that the set produced by adding a resolvant clause to the original input set is unsatisfiable iff the original is unsatisfiable. Hence, the soundness of the ME procedure is unaffected by the addition of lemmas to the initial set of matrix chains.

2.3.2. Example Problem

Consider the example problem described in s-linear resolution. The given set of clauses S is transformed into an initial set of matrix chains M_0 as follows:

Input Clauses		Matrix Chains		
P v Q v R	→	PQR,	QRP,	RPQ
P v Q v ~R	→	PQ~R,	Q~RP,	~RPQ
P v ~Q v R	→	P~QR,	~QRP,	RP~Q
~P v Q v R	→	~PQR,	QR~P,	R~PQ
P v ~Q v ~R	→	P~Q~R,	~Q~RP,	~RP~Q
~Pv~QvR	→	~P~QR,	~QR~P,	R~P~Q
~P v Q v ~R	→	~PQ~R,	Q~R~P,	~R~PQ
~P v ~Q v ~R	→	~P~Q~R,	~Q~R~P,	~R~P~Q
P v Q v R	→	PQR,	QRP,	RPQ

Figure 2 shows the search tree for the given set of clauses using the ME procedure. Those chains that are not circled are non-preadmissible and are therefore pruned from the search tree (they did not meet the last two restrictions of

preadmissibility). The lemmas used as input chains in extension are highlighted to distinguish them from the original input chain. The scope associated with each A-literal is shown above the boxed A-literal.

2.3.3. Elimination of Models

In the definition of the extension operation, restriction (c) specifies that the last literal of the center chain is to be resolved-upon with the first literal of the input chain. This process links the center chain with the input chain by their complementary literals. A truth value of TRUE is then assigned to the resolved-upon literal which becomes an A-literal. The sequence of A-literals in the center chain represents a "model" (partial interpretation). If the interpretation contradicts an instance of one of the matrix chains then this interpretation cannot satisfy the given set of input clauses, and is hence not a model. Interpretations that are not models are implicitly eliminated. If the given set of input clauses is unsatisfiable then all interpretations developed in this strategy will be eliminated by the time an empty center chain is obtained. Take, for example, the fifth center chain of Figure 2. This center chain was obtained after three extension and two reduction operations. The three A-literals R, Q and P represent a partial interpretation. Examining the set of matrix chains, the interpretation that R, Q and P are all true contradicts the chain ~P~Q~R (i.e., the clause ~P v ~Q v ~R is false) which was the input chain used at the last extension operation. This implies that this partial interpretation is "not a model" and should be eliminated.

2.3.4. Creation of Lemmas

The process of eliminating an interpretation is done by the contraction operation. During the contraction operation, new chains called lemmas are formed, and may be added to the original set of matrix chains. The rationale of this scheme is that if

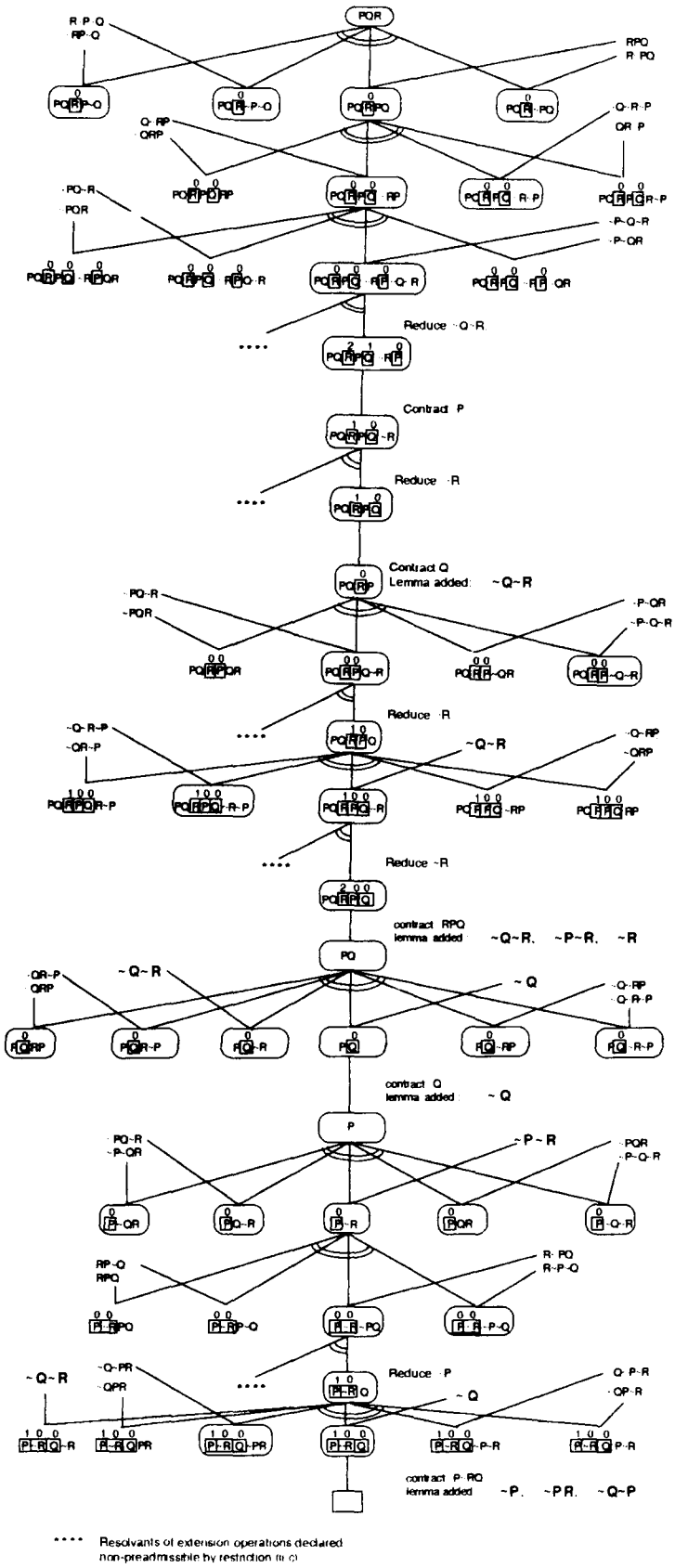


Figure 2. The search tree for S = {PQR, PQ~R, P~QR, ~PQR, P~Q~R, ~P~QR, ~PQ~R, ~P~Q~R} using the ME procedure.

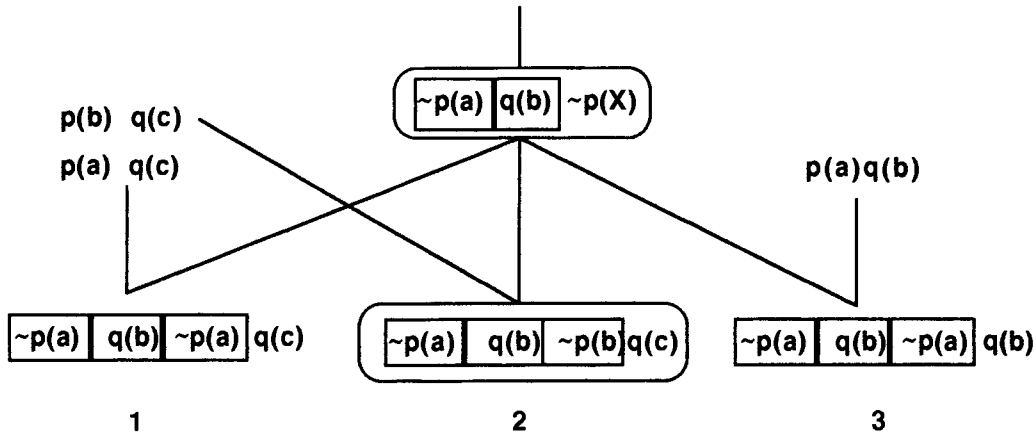
the interpretation formed by the sequence of A-literals contradicts an instance of one of the matrix chains, then the chain obtained by combining the complement of each A-literal of the center chain (as bounded by the scope associated to each A-literal) can be interpreted as true. In Figure 2, when the contraction operation is applied to the fifth center chain the lemma ~P~Q~R is produced but cannot be added to the set of matrix chains because it is subsumed by one of the matrix chains. At the seventh center chain, the contraction operation produced the lemma ~Q~R, which is added to the set of matrix chains. There are two possible effects in the addition of lemmas to the original set of matrix chains. Firstly, it may reduce the number of inference steps required for a refutation. Most of the lemmas added to the original set of matrix chains have lesser number of literals than the original matrix chains. When these lemmas are used as input chains, the extension operation will produce a resolvant with a lesser number of literals. The lesser the number of literals in the center chain, the lesser also the number of inference steps required to obtain a refutation. The second effect is that the addition of lemmas expands the search space as there are more possible input chains. In Figure 2, the extension operation produces four resolvants when the B-literal Q of the second center chain was resolved. In the thirteenth and the eighteenth center chains the operation produces seven resolvants each as a consequence of the addition of lemmas to the set of matrix chains.

2.3.5 Effects of the Restrictions

The pruning effect of the derivation strategy can be attributed to the two restrictions imposed : (a) resolving only the last literal of the center chain (restriction c of the extension operation) and (b) the preadmissibility restrictions. In the s-linear derivation strategy, all the literals of the center clause are available for resolution. This scheme produces many resolvants at each step and therefore creates a large search tree. The ME procedure avoids this problem by resolving only on the last literal of a center chain which significantly reduces the size of search tree. In the preadmissibility restrictions, restriction

(a) prunes resolvants obtained from extensions with a tautologous input chain. In this way, tautologous input chains can never be used as input parent during the extension operation. Loveland (1969a, p. 362) has pointed out that tautological input clause is not needed to prove the unsatisfiability of a set of clauses. Although, the ME procedure prevents tautologous input chains, tautologous center chains may exist in the derivation. The fourth center chain of Figure 2, which is the only preadmissible resolvant of the previous center, contains tautologous literals Q and ~Q. This is because the ME procedure does not implement the merging operation unlike with the s-linear resolution which has an implicit merging operation.

Restrictions (b) and (c) trim the search tree by ensuring that truth values are consistently and non-redundantly assigned to literals of the center chain. Restriction (b) eliminates resolvants that contain B-literals identical to any of the preceeding A-literals to avoid resolving a literal more than once and thereby catches loops. It is redundant to assign a truth value to a literal which had been previously assigned and still existing in the center chain. Stickel (1984, p. 215) justifies this restriction that - "it is unnecessary to attempt to solve a goal (B-literal) while in the process of attempting to solve that same goal". In Figure 2, the application of the extension operations on the third center chain produces two resolvant chains which are non-preadmissible. They are non-preadmissible because a B-literal R is identical to the first A-literal. Restriction (c) has two pruning effects. Firstly, it prunes resolvants which restriction (b) cannot detect. Loops that are detected later due to instantiation are therefore prevented by restriction (c). Take, for instance, the following derivation:



Resolvant 1 is pruned from the search tree by restriction (c) because of the redundancy of assigning truth value to the literal ~p(a) twice. Secondly, restriction (c) prunes resolvants that have inconsistent assignment of truth values which forces the application of the reduction operation. This implies that reduction of literals having identical atoms is compulsory in the ME procedure. This is an improvement on s-linear resolution because a compulsory ancestor resolution reduces the size of the search tree. Looking at the non-preadmissible resolvants obtained by applying the extension operation to the fourth center of Figure 2, the existence of A-literals R and ~R is logically inconsistent because R will be interpreted as true and false at the same time. Since A-literals R and ~R cannot be allowed to exist at the same time, the extension operation will fail after exhausting all possible input chains and thereby forcing the reduction operation to be performed.

2.4. t-linear Resolution

Kowalski and Kuehner (1971) formalised the t-linear and SL-resolution derivation strategies. They claimed that these two strategies are both refinements of the s-linear derivation strategy. The formulation of t-linear resolution is intended to clarify the definition of SL-resolution, simplify the comparison with other linear resolution systems, and is only defined for ground derivations. A ground derivation is a derivation consisting of ground clauses. A ground clause is a clause of which no literal contains a variable.

2.4.1 Formal Definition

Let $C_1, ..., C_n$ be a ground linear derivation from a set of input clauses S. A literal L in C_i is said to be a *descendant* of a literal L in ancestor clause C_j iff L occurs in every intermediate clause C_k, $j \leq k \leq i$. C_j is an *A-ancestor* of C_i iff :

i) $j < i$,

ii) C_{j+1} has an input parent, and

iii) all literals in C_j, except for the cancelled literal K in obtaining C_{j+1}, have descendants in C_i, that is $(C_j - \{K\}) \cong C_i$.

The cancelled literal K is called the *A-literal* of C_i from the A-ancestor C_j. To illustrate these, consider the sequence of clauses `({P,Q,R}, {P,Q}, {P,R}, {P,~Q}, {P}, {Q,R}, {~P,Q}, {Q})` in Figure 3. The A-ancestors of `{Q}` are `{P}` and `{Q,R}` with A-literals P and R respectively. `{P,~Q}` and `{~P,Q}` do not qualify as A-ancestors because their successors are not derived by input resolution. `{P,Q}` does not qualify because Q does not appear in every intermediate clause between `{P,Q}` and `{Q}`.

A linear derivation is said to be t-linear if restrictions i), ii) and iii) are satisfied.

i) If C_{i+1} is obtained by ancestor resolution, then it is obtained by resolution with an A-ancestor of C_i.

ii) If C_i contains a literal complementary to one of its A-literals, then C_{i+1} is obtained by ancestor resolution.

iii) A-literals of C_i from distinct A-ancestors have distinct atoms.

Kowalski and Kuehner pointed out that the t-linear derivation strategy is compatible with the no-tautologies restriction.

2.4.2 Example Problems

Figure 3 shows the search tree for the example problem introduced in s-linear resolution, using the t-linear derivation strategy with the no-tautologies restriction. Figure 4 illustrates the search tree for a given set of clauses, S = { {~P,~R}, {~Q,~R}, {Q,R}, {P}, {R} }, using the t-linear derivation strategy. As in Figure 1, permissible clauses are circled to distinguish them from the tautologous clauses which are pruned from the search tree. Ancestor parent clauses (A-ancestors) are highlighted to differentiate them from the input parent clauses.

2.4.3. Effects of the Restrictions

Restriction (i) makes sure that the resolvant clause C_{i+1} always subsumes the center clause C_i. During an ancestor resolution of the t-linear derivation strategy, an A-literal from an A-ancestor clause is being used to resolve with one of the literals of the center clause. Since all the other literals of an A-ancestor have descendants in the center clause , then it follows that the resolvant of an ancestor resolution always subsumes the center clause. Take, for instance, the ancestor resolution of the fourth center chain of Figure 3. The center clause {P,~Q} has an A-literal Q from the A-ancestor {P,Q}. The resolvant {P}, after applying resolution and an implicit merging operation, subsumes the center clause {P,~Q}. Obviously, this restriction has similarity with restriction (ii b) of the s-linear derivation strategy. However, restriction (i) of t-linear resolution is more effecient than restriction (ii b) of s-linear resolution since the former can immediately select an appropriate ancestor clause for ancestor resolution. In s-linear's ancestor resolution, the resolvant is generated first, and then tested to find out if it subsumes its center parent clause, which is a time consuming process. In t-linear resolution, only A-ancestors are tried in ancestor resolution which makes the subsumption test unnecessary. This restriction also produces a narrower search tree because only A-ancestors are

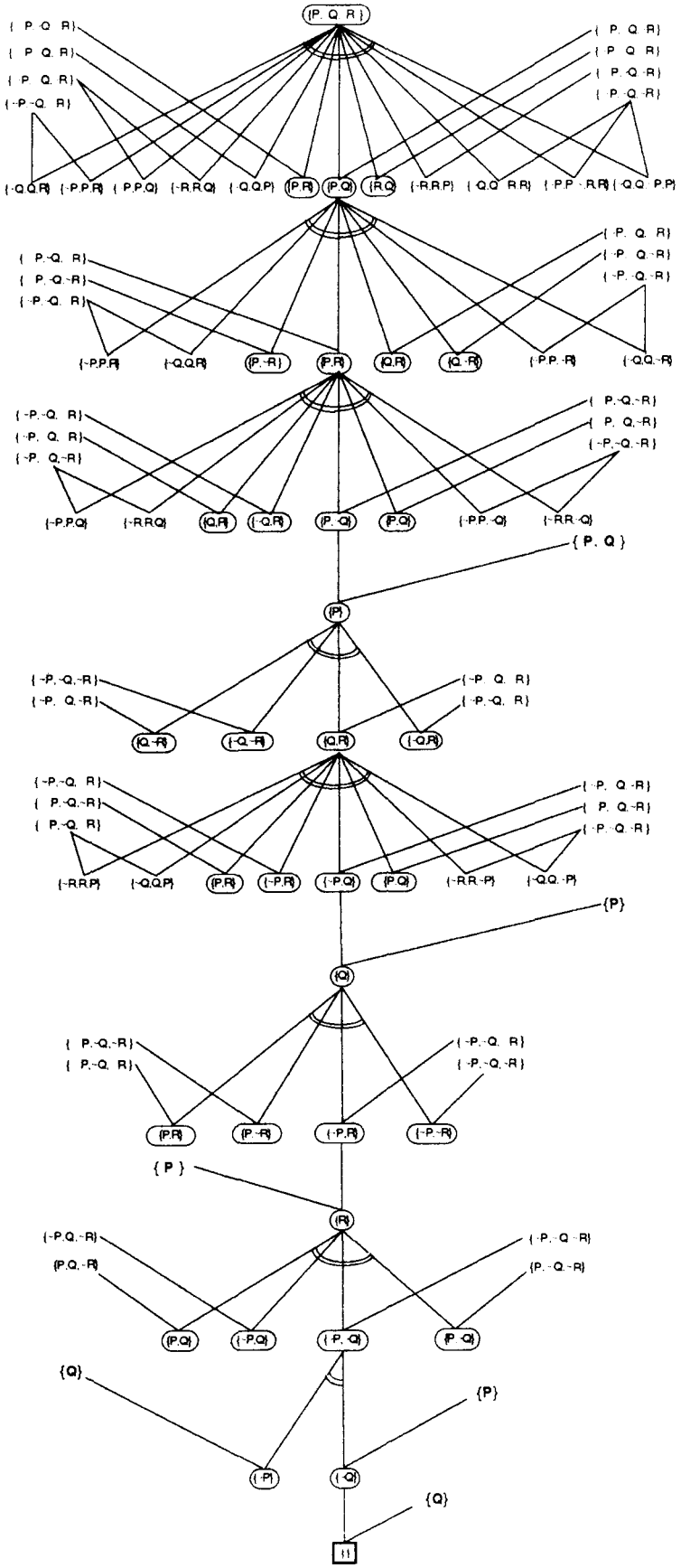


Figure 3. The search tree for S = { {P, Q, R}, {P,Q, ~R}, {P,~Q,R}, {~P,Q,R}, {P,~Q,~R}, {~P,~Q,R}, {~P,Q,~R}, {~P,~Q,~R} } using t-linear resolution.

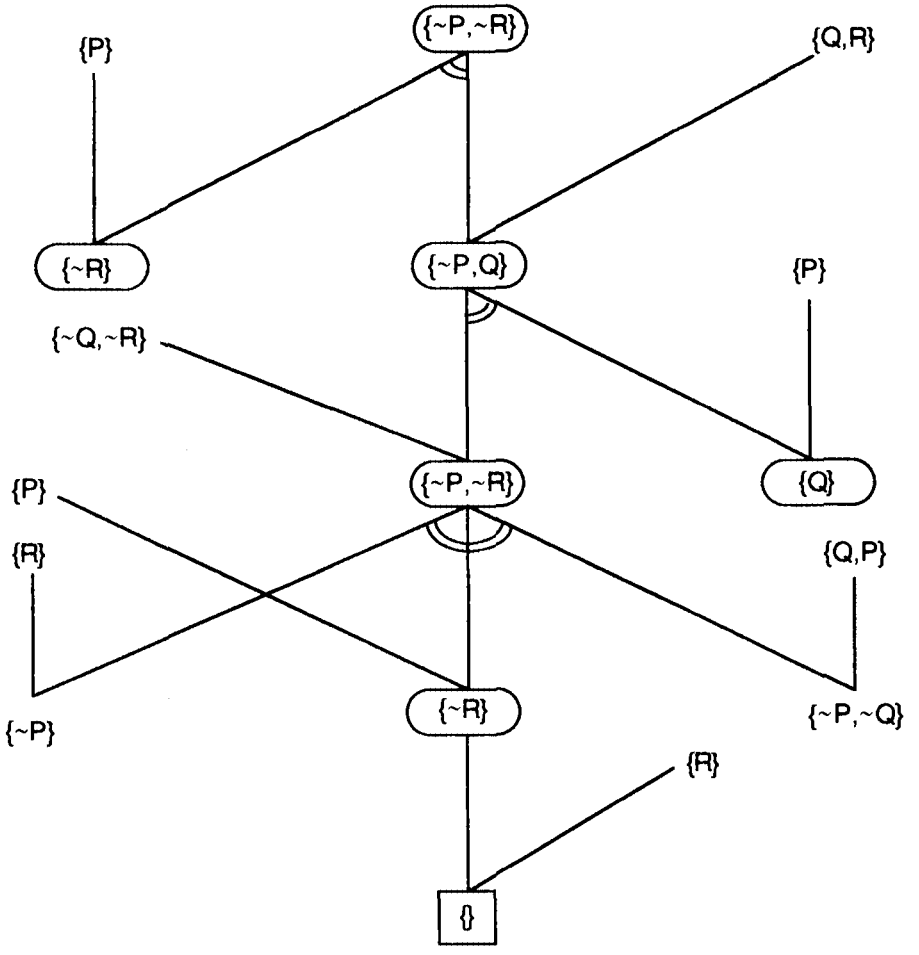


Figure 4. The search tree for S = {{~P, ~R}, {~R, ~Q}, {Q,R}, {R}, {P}} using t-linear resolution.

considered for ancestor resolution. In Figure 3, the center clause {~P,Q} is only ancestor resolved with its A-ancestor {P} to obtain a new center clause {Q}. In s-linear, there are two ancestor clauses considered for ancestor resolution with the same center clause, the ancestor clauses {P,Q} and {P} (see Figure 1).

Restriction (ii) makes the t-linear derivation strategy a refinement of the s-linear derivation strategy. In s-linear derivation strategy, ancestor and input resolution are both applied whenever possible, as observed in Figure 1. In t-linear resolution, an input resolution is disregarded whenever ancestor resolution is possible. This restriction is a preemptive version of the third preadmissibility restriction of the ME procedure. This implies that if a B-literal can be resolved on with an A-literal, it is immediately removed rather than applying an extension operation which will be failed eventually by the third preadmissibility restriction. The practical effect of this restriction is the significant pruning of the search tree. However, compulsory ancestor resolution does not always produce a refutation as short as a minimal s-linear refutation, since it may prevent an input resolution which derives a resolvant clause that can be easily refuted. Take, for instance, the sequence of clauses starting from clause {~P,R} in Figures 1 and 3. Figure 1 has the derivation ..., {~P,R}, {~P~Q}, {~Q}, {} while Figure 3 has ..., {~P,R}, {R}, {~P~Q}, {~Q}, {}. Figure 3 has a longer derivation than Figure 1 as a result of the compulsory ancestor resolution of the center clause {~P,R} with its A-ancestor {P}.

Restriction (iii) prevents loops in derivations. Figure 4 illustrates an example how restriction (iii) prevents loops. When the clause {Q,R} is chosen as input parent in applying resolution to the the third center clause, the resolution produces a redundant resolvant. The resolvant {~P,Q} has A-literal ~R from the first and the third center clause which make the resolvant redundant. Obviously, this resolvant is a repetition of the second center clause. This will cause loops in the derivation if this is allowed. However, it is possible that some of the resolvants trimmed by restriction (iii) are not redundant. The trimmed resolvant {~P} is not redundant because a refutation can still be obtained.

It is obvious that restrictions of t-linear resolution are difficult to implement in the set representation of clauses. There is a need to change the representation in order to associate A-literals and A-ancestors of a clause. This leads to the formulation of SL-resolution.

2.5. SL-resolution

SL-resolution combines all the restrictions of the t-linear derivation strategy with the no-tautologies restriction. It has also a *selection function* that calls for a single literal to be selected from the most recently introduced literals of the center parent chain for use in input resolution. Ringwood (1988, p. 6) defines a selection function as a function from a set of atoms to atoms such that an image atom is an element of the preimage set. Kowalski and Kuehner (1971 : p. 233) implement two versions of SL-resolution, denored as SL(1) and SL(2), with different selection functions. SL(1) has a selection function which chooses and resolves upon the alphabetically least atom, while SL(2) chooses the alphabetically greatest atom. SL-resolution uses the chain format for the representation of clauses. In this format, restrictions (i) and (ii) of the t-linear derivation strategy can be simply implemented by immediately deleting any of the most recently introduced literals which are complementary to an associated A-literal. SL-resolution does not only allow ancestor resolution but also includes factoring as suboperation of the reduction operation. Factoring corresponds to the implicit merging operation of s-linear and t-linear derivation strategies. The other operations used are the extension operation, which is equivalent to input resolution, and the truncation operation, which removes A-literals from the center chain.

2.5.1. Formal Definition

Given a set of clauses,S, which is converted into a set of chains, M, a set of support, S_O, which is also converted to another set of chains M_T, and a selection function Ω, a sequence of chains $C_1, ..., C_n$ is an SL-derivation if restrictions i), ii) and iii) are satisfied.

i) $C_1, ..., C_n$ is a linear derivation and $C_1 \in M_T$.

ii) Each C_{i+1} is obtained from C_i by

   a) extension, or

   b) reduction, or

   c) truncation

iii) Unless C_{i+1} is obtained from C_i by reduction, then no two literals occuring at distinct positions in C_i have the same atom (*admissibility restriction*).

C_{i+1} is obtained from C_i by **extension** with a chain B iff :

a) The rightmost literal L in $\Omega(C_i)$ is a B-literal.

b) C_i and B share no variables.

c) The selected B-literal L in $\Omega(C_i)$ and the complement of the leftmost literal K in B are unifiable with mgu θ.

d) θ is applied to the result of concatenating $\Omega(C_i)$ and B minus K in that order. The literal $L\theta$ in C_{i+1} descending from the rightmost literal of $\Omega(C_i)$ becomes an A-literal in C_{i+1}. The rest of the literals in C_{i+1} have the same classification as in C_i or B.

C_{i+1} is obtained from C_i by **reduction** iff :

a) The rightmost literal in C_i is a B-literal.

b) C_i is not obtained from C_{i-1} by truncation.

c) The rightmost cell of C_i contains a B-literal L and either

1) C_i contains a B-literal K, which is not in the rightmost cell of C_i (basic factoring), or

2) C_i contains an A-literal ~K, which is not the rightmost A-literal of C_i (ancestor resolution).

d) L and K are unifiable with mgu θ.

e) $L\theta$ is deleted from $C_i\theta$. The classification of every literal in C_{i+1} is the same as it was in C_i.

C_{i+1} is obtained from C_i by **truncation** iff :

a) The rightmost literal in C_i is an A-literal.

b) All rightmost A-literals are deleted from C_i. The classification of every literal in C_{i+1} is the same as it was in C_i.

2.5.2. Example Problems

Figure 5 shows the search tree for the previous example using SL-resolution. To illutrate the incompleteness of compulsory factoring, a search tree of the set of clauses `S= { ~r, r~q(b)~m, m~p(X)~q(X), q(a), q(b), p(a) }` is presented in Figure 6. The search tree shown in Figure 7 depicts the effect of the admissibility restriction in enforcing reduction to reducible literals which have identical atoms or atoms which become identical later in the derivation. The admissible resolvants are circled to distinguish them from the inadmissible resolvants. These resolvants are declared non-admissible because they contain literals having the same atoms and are not reduceable.

2.5.3. Effects of the Restrictions

It is clear that the search tree of SL-resolution (Figure 5) is not as complex as the search trees produced by the derivation strategies discussed earlier. This pruning effect can be attributed mainly to two of the restrictions: (i) the restriction which only selects a single literal from each center parent chain for extension operation (selection function Ω), and (ii) the admissibility restriction.

The selection function of SL-resolution has similar effect to the ME procedure scheme of resolving only the last literal of the center parent chain. It reduces the size of the search tree significantly. However, the two schemes are different. The ME procedure method of selecting a literal to be resolved on can be categorised as a selection rule. "A selection rule is a decision procedure for choosing an atom to resolve on each resolution step and as such it may depend on the history of the derivation." (Ringwood 1988, p. 6). This differs from a selection function because it can select a literal based on a desired ordering which may not depend on the history of the derivation. However, the compatibility of a more liberal employment of selection function can be established for the ME procedure (Kowalski and Kuehner, 1971, p. 240).

The admissibility restriction encompasses the three t-linear restrictions and the no-tautologies restriction. It ensures compulsory merging operation, compulsory ancestor resolution on literals having identical atoms, and no tautologous resolvants. The importance of having no-tautologies has already been discussed in s-linear resolution. The no-tautologies restriction has contributed much in narrowing the search tree. Kowalski and Kuehner (1971, p. 238) had mentioned that if a literal can be removed by reduction then this is done before any extension operations are performed. This is misleading because this can be interpreted that an extension operation cannot be performed when the chain is reducible. Figure 6 illustrates an example where a compulsory reduction causes incompleteness of the derivation strategy. When the literal ~q(X) of the third center chain is removed by factoring with ~q(b), the resulting fourth

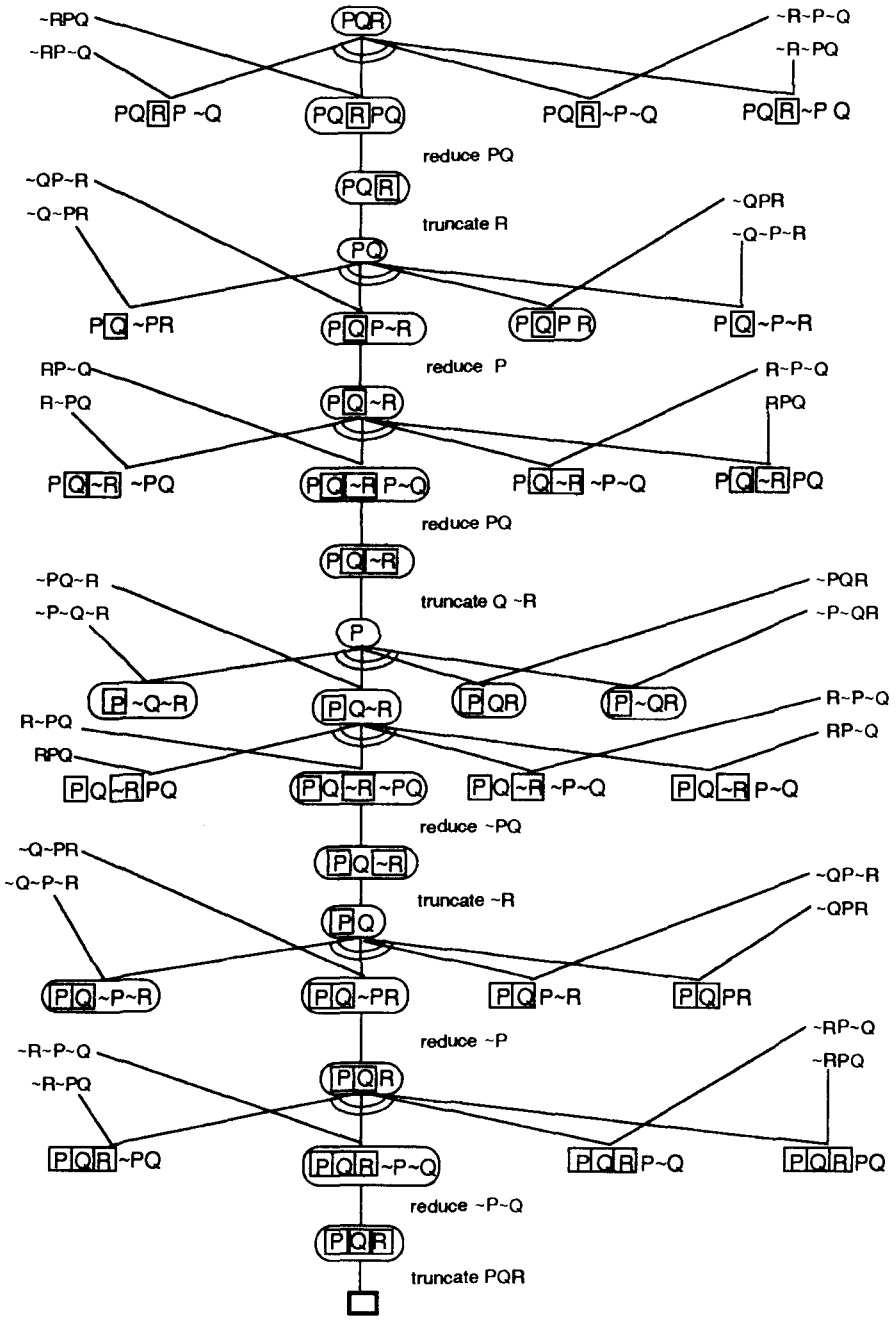


Figure 5. The search tree for S = {PQR, PQ~R, P~QR, ~PQR, P~Q~R, ~P~QR, ~PQ~R, ~P~Q~R} using SL-resolution.

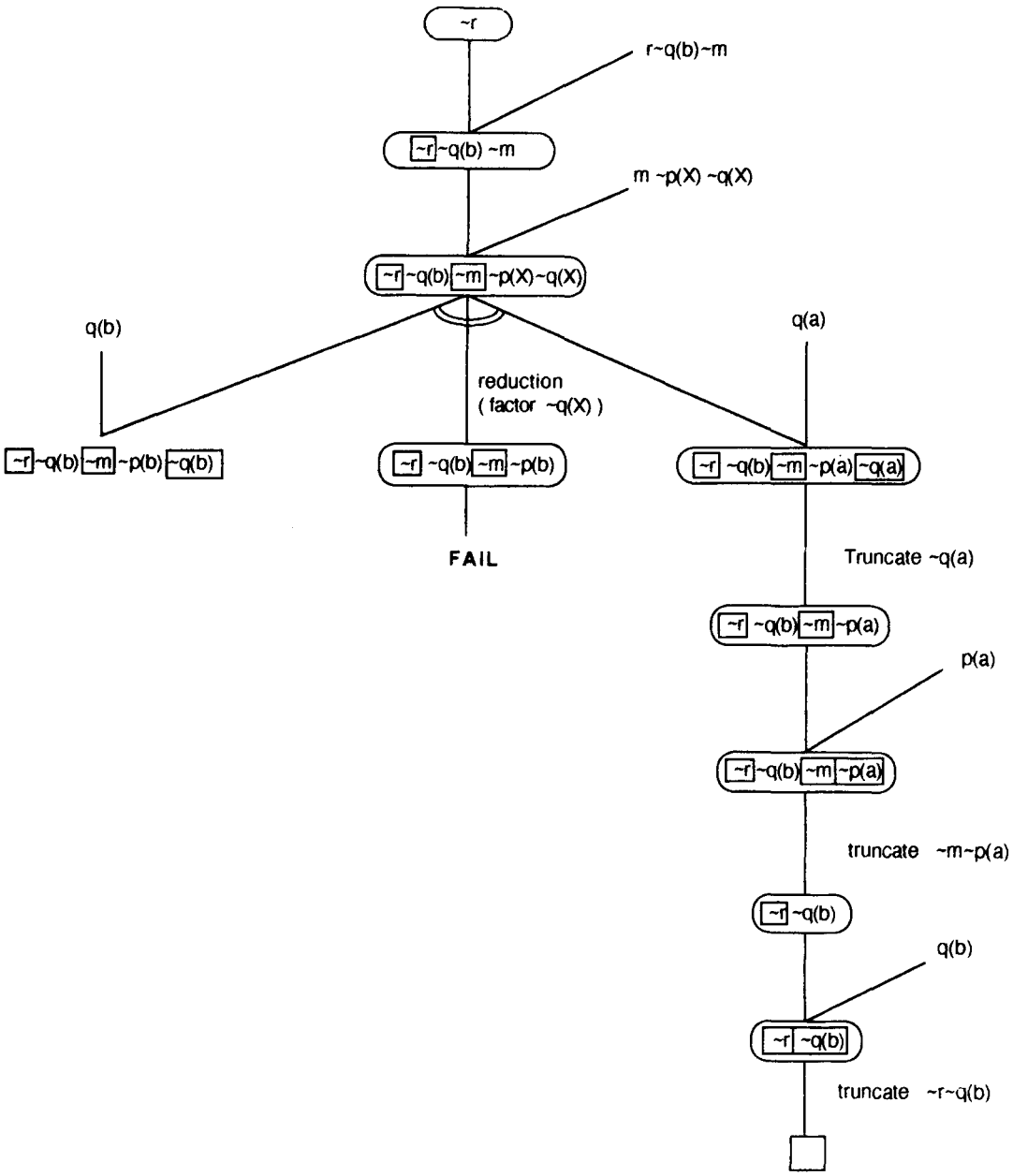


Figure 6. The search tree for S ={~r, r~q(b)~m, m~p(X)~q(X), q(a), q(b), p(a)} using SL-resolution.

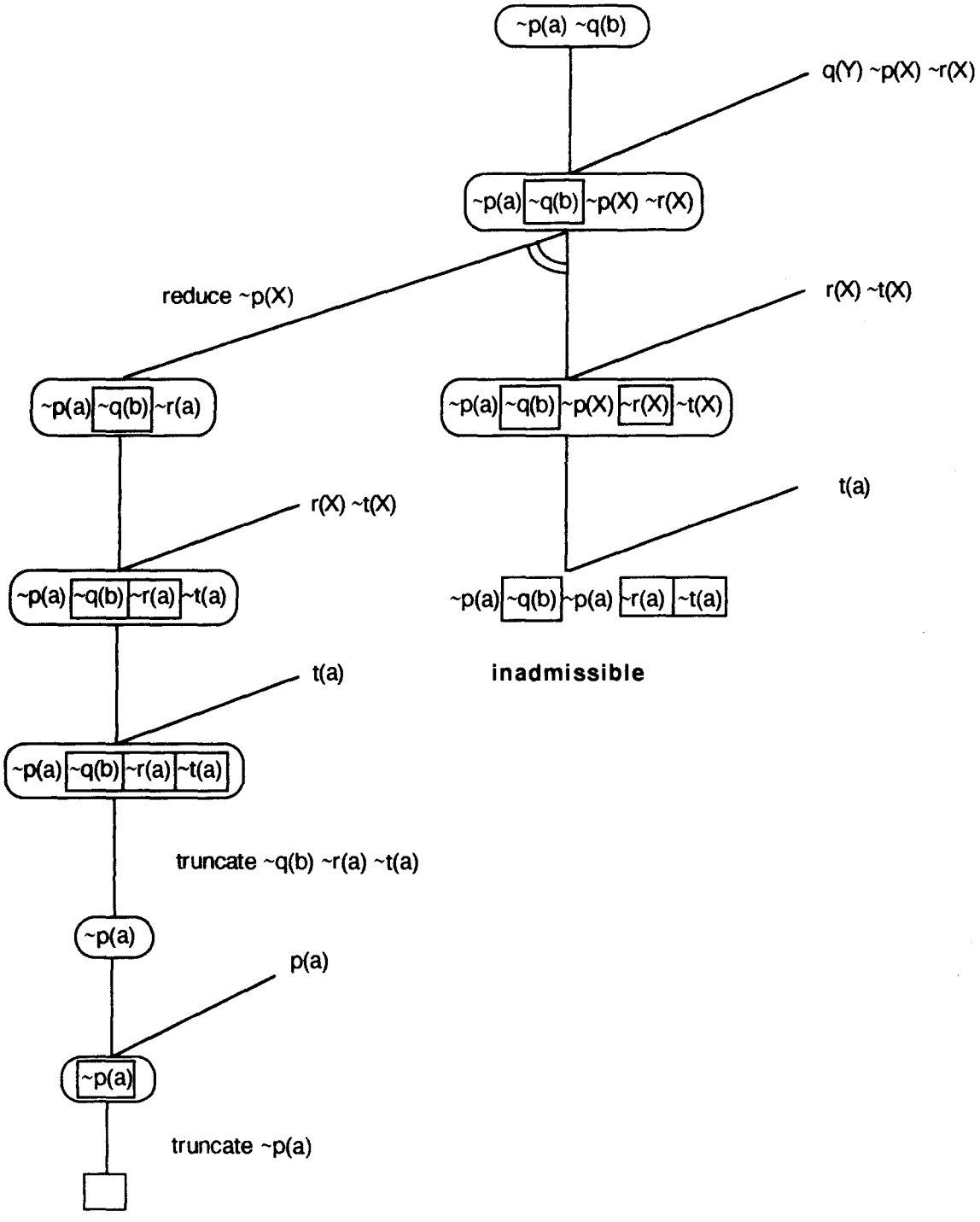


Figure 7. The search tree for S ={~p(a)~q(b), q(Y)~p(X)~r(X), r(X)~t(X), t(a), p(a)} using SL-resolution.

center chain is unresolvable and thus, refutation cannot be obtained if reduction is compulsory. The admissibility restriction only ensures that a reduction operation is performed when two reducible literals have identical atoms or atoms which become identical later in the derivation. Figure 7 illustrates this point. When an extension operation is applied on the second center chain, after two more extension operations a resolvant having two identical B-literals is obtained. This resolvant is then rejected by the admissibility restriction and the derivation backtracks to the second center chain to enforce the reduction operation.

Based on the definition of the admissibility restriction, a chain which is non-reduceable is declared non-admissible if it contains :

i) two B-literals having identical atoms,

ii) a B-literal and an A-literal having identical atoms, or

iii) two A-literals having identical atoms.

Restriction (i) enforces factoring and applies the no-tautologies restriction. This restriction is more strict than the first restriction of the ME procedure's preadmissibility restrictions. SL-resolution does not only eliminate tautologous resolvants derived from tautologous input chains but at all instances. Factoring shortens and simplifies the derivation by maintaining only distinct literals in the center chain. Restriction (ii) is an extended version of the second preadmissibility restriction of the ME procedure. The ME procedure catches redundant resolvants only when an A-literal preceeds an identical B-literal. SL-resolution rejects resolvants that contain A- and B-literals with identical atoms (regardless of their order in the chain), unless the next operation is reduction. The purpose of this restriction is to enforce factoring in the case of identical B- and A-literals, to enforce ancestor resolution in the case of complementary A- and B-literals and to prevent tautologies in the case of complementary B- and A-literals. In the ME procedure, both reduction and extension operations can be applied to a center chain [P][~R] ~PQ (literals enclosed by [] are A-literals). This means that there will be a

derivation obtained by reducing first the B-literal ~P and then an extension operation to resolve on the B-literal Q, and another derivation by applying first an extension operation to Q and reduce the B-literal ~P later. In SL-resolution, only the reduction operation can be applied because restriction (ii) will ensure that an extension operation cannot be applied due to the presence of identical atom P. Restriction (iii) is exactly the same to the third preadmissibility restriction of the ME procedure.

In the definition of the reduction operation, restriction (b) requires that the parent chain to be reduced is not obtained from a truncation operation. This is because the admissibility restriction will ensure that the chain to be truncated is already cleared of any B-literals that could be removed by reduction. Hence, a chain obtained after truncation is already unreduceable.

Restriction (c) of the reduction operation is required to deal with redundancies introduced in the conversion of clauses to matrix chains. Take, for instance, the input clause

```
p(X) v ~f(X) v ~f(a)
```

which is converted into 5 matrix chains :

```
p(X) ~f(X) ~f(a)
~f(X) ~f(a) p(X)
~f(a) p(X) ~f(X)
p(a) ~f(a)
~f(a) p(a)
```

When an extension operation is applied to a center parent chain, `~p(a)`, two of the above chains will qualify to be input parent chains. The two resolvants obtained are :

$\boxed{\sim p(a)}$ ~f(a) ~f(a)

$\boxed{\sim p(a)}$ ~f(a)

Obviously, the first resolvant is redundant since after a reduction operation (without restriction (c)) the chain will become similar to the second resolvant.

A second case is the clause `~p(a) v ~p(X)` which will generate three matrix chains during conversion :

```
~p(a)  ~p(X)
~p(X)  ~p(a)
~p(a)
```

An extension operation of the center parent chain `p(a)` will produce three resolvants. The resolvant obtained from the second matrix chain is redundant, since after a reduction operation (without restriction (c)) the chain will become the same as the other resolvant. Hence, to avoid this redundancy, restriction (c) is imposed on the reduction operation.

The inclusion of factoring into the SL-resolution has a drawback at the first-order level. Take, for instance, the clause that represents the transitivity of equality

```
equal(X,Y) v ~equal(X,U) v ~equal(U,Y)
```

which will produce two factors :

```
equal(X,Y) v ~equal(X,U) v ~equal(U,Y)
equal(Y,Y) v ~equal(Y,Y)
```

The second factor has lost the meaning of the original clause. In fact, the second factor can never be used at all because of restriction (c) imposed in the reduction operation. This means that keeping the second factor is just a waste of memory space.

2.6. Graph Construction

The added selection function of the SL-resolution and the selection rule used in the ME procedure have reduced enormously the search space of linear derivations. However, Shostak (1976, p. 59) found out that the added selectivity of these two derivation strategies may produce longer derivation as a result of "repeated computation". The added selectivity of SL-resolution and the ME procedure limits the selection of a literal within the most recently introduced literals of the chain. It is not difficult to show that selecting a literal outside from the most recently introduced literals may obtain a simpler refutation. Shostak proved this by presenting a simplified version of t-linear resolution called *st-linear* resolution. The st-linear resolution is basically the same as the t-linear resolution except for the absence of the third restriction of t-linear resolution in st-linear resolution. Shostak used st-linear resolution to show that unnecessary repeated refutation of the same literal can be avoided. Unnecessary repeated refutation means a repeated use of a certain set of input clauses to refute the same literal. To illustrate this point, consider Figure 11 (p. 56). The input chains ~L~P, PRN and ~R~L were used in that order by SL-resolution twice to refute the literal L. st-linear resolution avoids this unnecessary repeated refutation by allowing the selection of literal to be resolved on from any of the literals of the center clause. Looking at Figure 10 (p. 55), st-linear resolution is capable of selecting the literal Q instead of L from the third center clause, resulting in a simpler refutation. The ME procedure and SL-resolution cannot select the literal Q from the third center chain because of the constraints of the added selectivity. Allowing flexible selection of the literal to be resolved on, however, produces a larger search space. In view of this, Shostak devised a new derivation strategy, called the Graph Construction (GC) procedure. The GC procedure alleviates the unnecessary repeated refutation problem and at the same time minimises the size of the search space by retaining the added selectivity

of SL-resolution. The GC procedure has an additional mechanism that converts truncated A-literals into C-literals. A C-literal is inserted at a specific position called the C-point, which is associated to the truncated A-literal. Any B-literal whose complement is unifiable with any preceeding C-literal is deleted from the center chain (C-reduction). In this scheme, unnecessary repeated refutation of a B-literal is avoided.

2.6.1. Formal Definition

Let E and E_o be the sets of elementary chains derived from a given set of input clauses S and set of support S_o respectively. With a given selection function Ω, a sequence of chains $C_1, ..., C_n$ is a GC procedure derivation if restrictions i), ii), iii), and iv) are satisfied.

i) $C_1, ..., C_n$ is a linear derivation and $C_1 \in E_o$.

ii) Each C_{i+1} is obtained from C_i either by

   a) extension,

   b) reduction or

   c) truncation

iii) C_{i+1} must be obtained by reduction if it is possible (compulsory reduction).

iv) C_i must not contain two non-B-literals having identical atoms.

C_{i+1} is obtained by **extension** with input chain B from E iff :

a) The rightmost literal L in $\Omega(C_i)$ is a B-literal.

b) The selected B-literal L in $\Omega(C_i)$ and the complement of a literal K in B are unifiable with mgu θ.

c) θ is applied to the result of concatenating $\Omega(C_i)$ and B minus K, in that order. The literal $L\theta$ in C_{i+1} descending from the rightmost literal of $\Omega(C_i)$ becomes an A-literal in C_{i+1}. The C-point associated with the A-literal L is

set to the left of the leftmost literal of C_{i+1}. The rest of the literals in C_{i+1} have the same classification as in C_i or B.

C_{i+1} is obtained by **reduction** iff :

a) a B-literal L in the rightmost cell of C_i, and an A- or C-literal K to the left of L, are complementary with mgu θ.

b) $L\theta$ is deleted from $C_i\theta$. The C-point associated with each A-literal to the right of K is set just to the right of K if the C-point at C_i is to the left of K. All other literals descending from C_i retain their classification.

C_{i+1} is obtained by **truncation** iff :

a) the rightmost literal L in C_i is a non-B-literal.

b) L is deleted from C_i. If L is an A-literal, the complement of L is inserted at the C-point associated with L. The classification of the inserted literal is a C-literal. The classification of all the other literals in the chains remains unchanged.

2.6.2. Example Problems

The search tree, using the GC procedure for the set of clauses described in the previous derivation strategies is shown in Figure 8 (p.53). Chains that are not encircled are declared inadmissible because they will eventually violate restriction (iv) after applying an extension, or a reduction and an extension operation to the chain.

The set of clauses `S = { ~T~N, RPN, L~Q, ~R~L, MQN, L~M, ~P~L, T }` is presented to point out the disadvantage of the added selectivity of SL-resolution and the ME procedure. Figures 9, 10, 11 and 12 are the search trees for this

given set of clauses using the ME procedure, st-linear resolution, SL-resolution and the GC procedure respectively. Another set of clauses `S = { ~p(X)~q(X)~r, q(b), q(a), r~q(b), p(a)}`, which was defined by Sutcliffe (1989, p. 17), is also presented to demonstrate the problem of compulsory C-reduction. Figure 13 shows the search tree of this given set of clauses with `~p(X)~q(X)~r` as the chosen top clause.

2.6.3. Effects of the Restrictions

One distinguishable similarity between Figure 4 of SL-resolution and Figure 8 of the GC procedure is that they have the same number of extension operations applied. Although the GC procedure does not impose factoring, it has minimised the application of extension operations by "recycling" truncated A-literals and imposes a reduction operation using the recycled A-literals.

In the first example, the GC procedure and SL-resolution seem to have the same effect as manifested by the same number of extension and reduction operations applied. However, in the second example, the GC procedure obtained a simpler refutation than SL-resolution. The GC procedure has a simpler refutation than SL-resolution because unnecessary repeated refutation is avoided in the GC procedure by reducing a B-literal with a conditionally proven C-literal. Truncated A-literals are wasted in SL-resolution while the GC procedure has a mechanism that "recycles" truncated A-literals.

Reduction with C-literals may have the same effect as factoring in SL-resolution. The role of these two operations in the derivation strategy is to simplify the derivation by minimising the application of extension operations. However, C-reduction is more effective than factoring in most cases. One case of this is shown in Figures 11 and 12. SL-resolution has resolved on the literal L twice by extension (at the third and tenth center chains of Figure 11). The GC procedure resolved on the literal L once by extension at the third center chain and once by C-reduction at the eleventh center chain of

Figure 12. C-reduction removed the B-literal L immediately while in SL-resolution, the process of refuting L at the third center chain, which involves a series of extension operations, is repeated in refuting L at the tenth center chain. Analysing the search tree of Figure 12, the first refutation of L tells that L cannot be TRUE, the GC procedure "learns" this by inserting a C-literal which is a contradiction of the previous truth assignment of L. Hence, when the B-literal L is introduced again to the chain it is immediately removed from the center chain by C-reduction to avoid inconsistency. The idea of factoring is to maintain distinct B-literals on the center chain to avoid redundancy. However, this is only effective at the early stage of the derivation when there are still B-literals at the left side of the center chain. But once these B-literals become A-literals, the factoring operation becomes less effective. In contrast, the C-reduction of the GC procedure will become more effective as the derivation go deeper because there will be more C-literals inserted at the left side of the center chain.

The GC procedure has a narrower search tree than the ME procedure because the ME procedure produces lemmas which increase the number of possible input parent chains during an extension operation. The creation of C-literals in the GC procedure does not affect the number of input chains. In fact, the insertion of C-literals makes restriction (iv) more effective in cutting down redundant resolvants. In Figure 8, when the extension operation is applied to the tenth center chain, two resolvants are inadmissible because of the presence of C-literal ~R at the left side of the resolvants. Hence, the creation of C-literals has a better effect than the addition of lemmas in the ME procedure. Shostak (1976, p. 63) has pointed out that "ME lemmas tend to be highly redundant" and have limited value in application.

Restriction (iii) (compulsory reduction) has narrowed down the search tree and simplify the derivation significantly. However, compulsory reduction with C-literals at the first-order level is incomplete, as is shown in (Sutcliffe and Tabada, 1989, pp.17-18). As shown in Figure 13, compulsory C-reduction is incomplete at the first order level. He suggested that this problem can be overcame by slightly modifying the compulsory

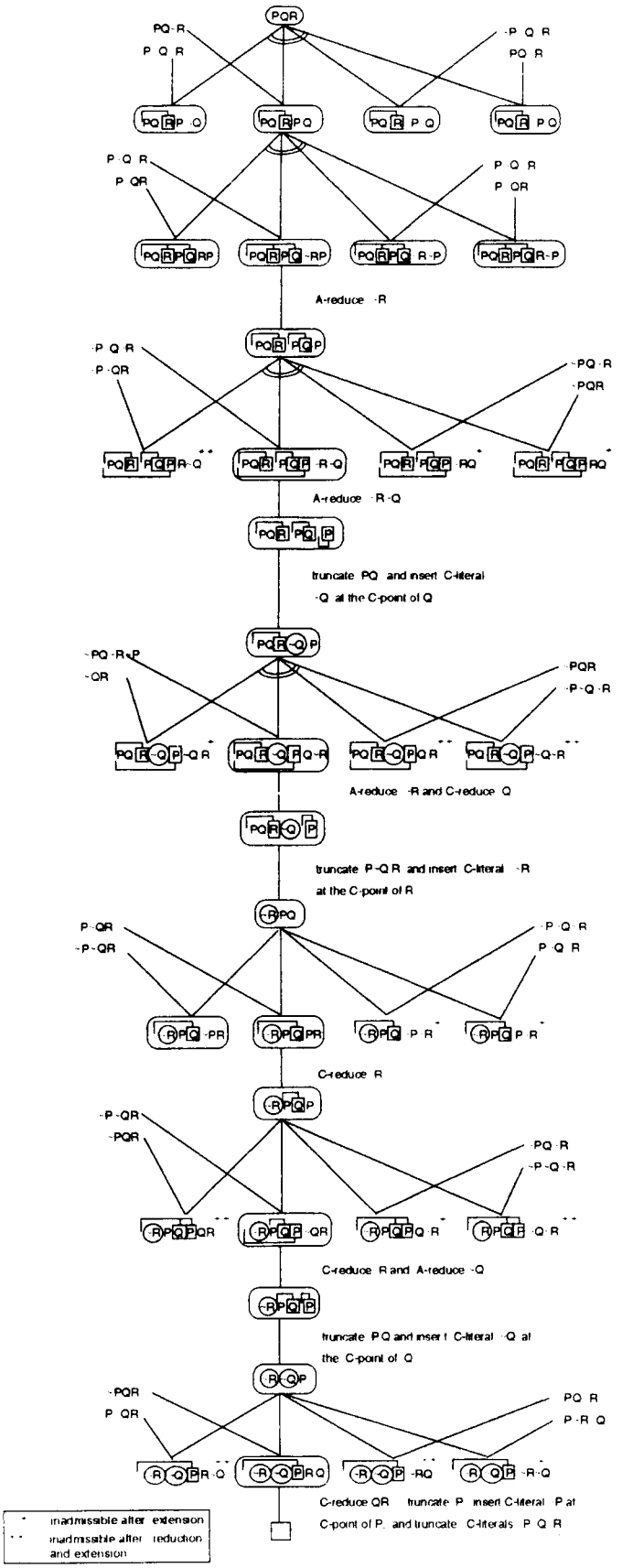


Figure 8. The search tree for S = {PQR, PQ~R, P~QR, ~PQR, P~Q~R, ~P~QR, ~PQ~R, ~P~Q~R} using the GC procedure.

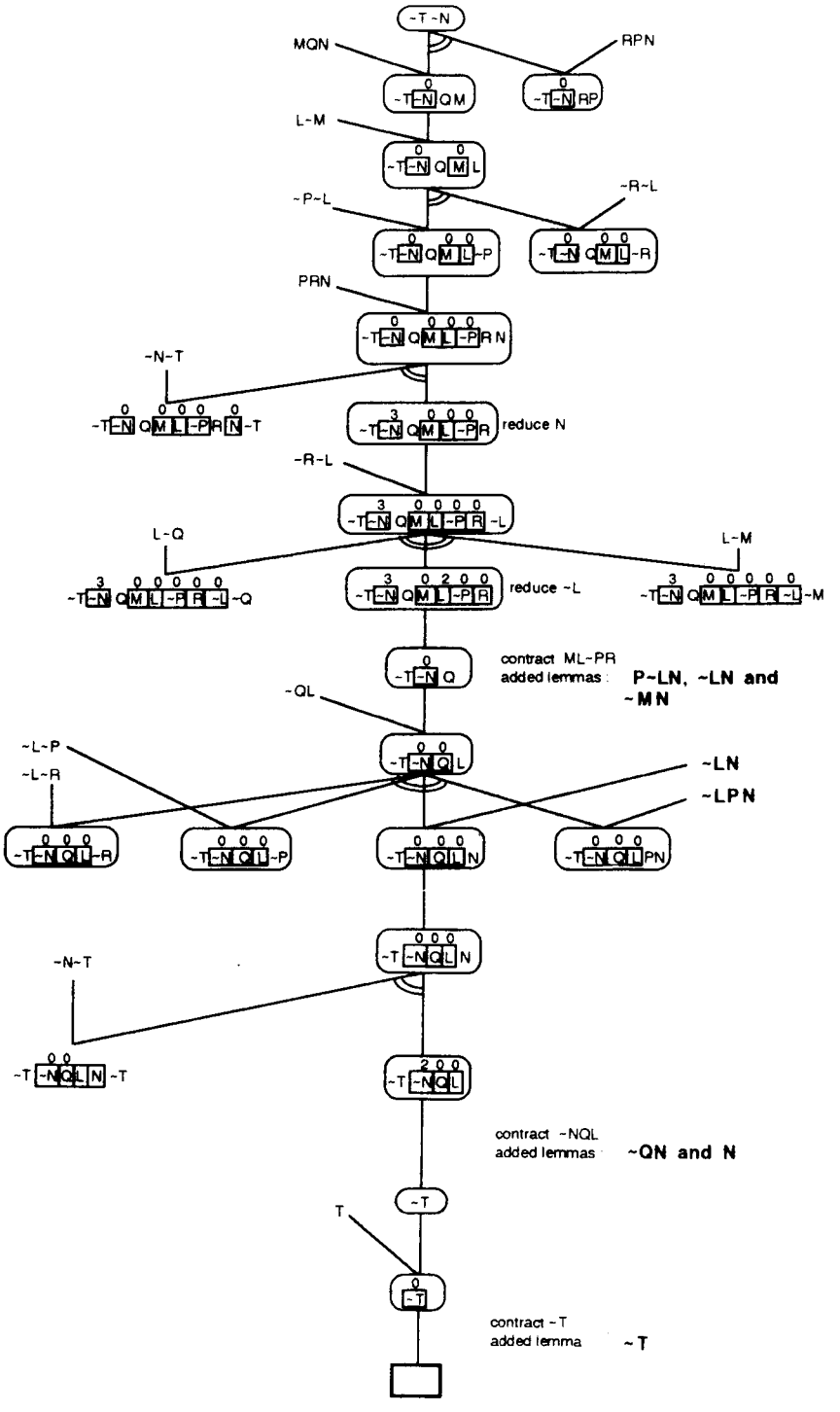


Figure 9. The search tree for S = {~T~N, RPN, L~Q, ~R~L, MQN, L~M, ~P~L, T} using the ME procedure.

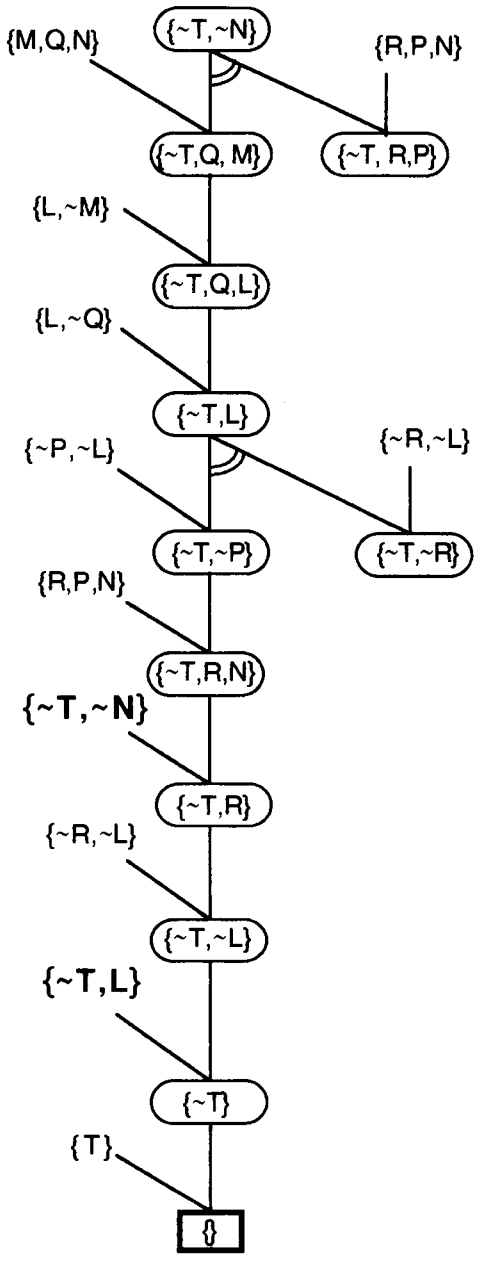


Figure 10. The search tree for S = {~T~N, RPN, L~Q, ~R~L, MQN, L~M, ~P~L, T} using st-linear resolution.

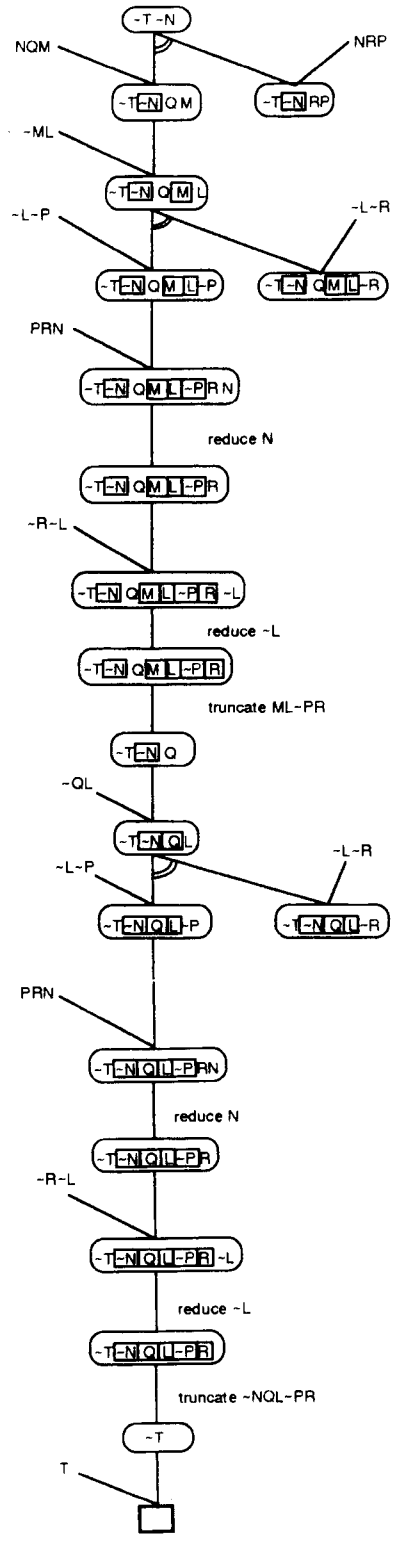


Figure 11. The search tree for S = {~T~N, RPN, L~Q, ~R~L, MQN, L~M, ~P~L, T} using SL-resolution.

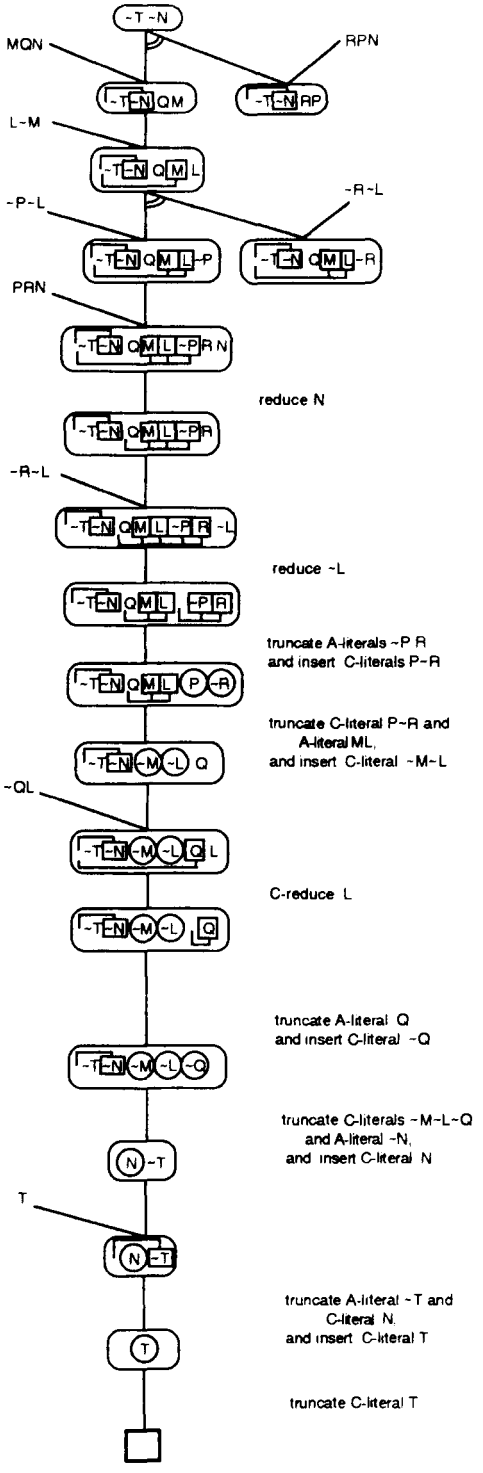


Figure 12. The search tree for S = {~T~N, RPN, L~Q, ~R~L, MQN, L~M, ~P~L, T} using the GC procedure.

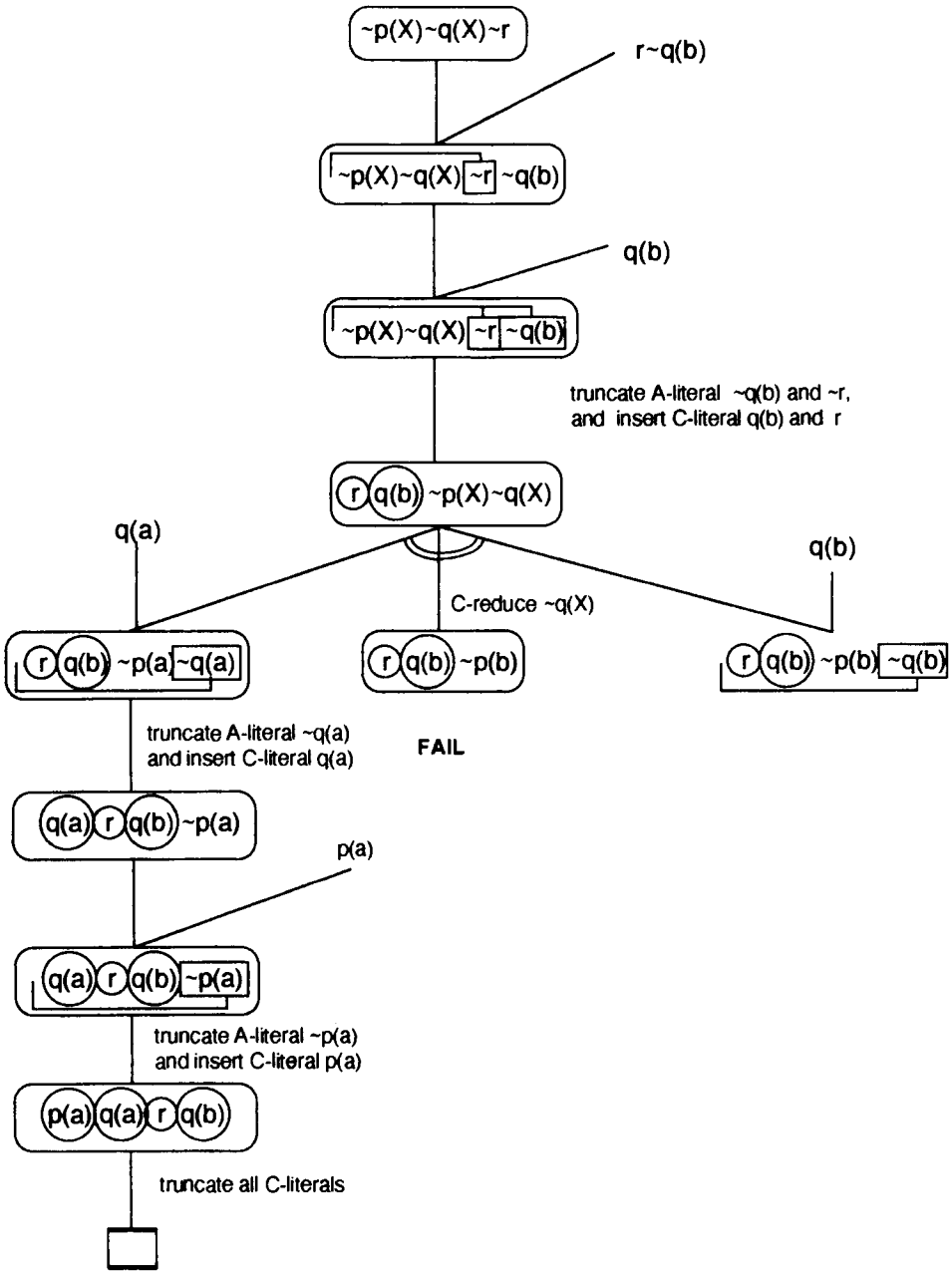


Figure 13. The search tree for S = {~p(X)~q(X)~r, q(b), q(a), r~q(b), p(a)} using the GC procedure with the modification suggested by Sutcliffe (1989).

reduction restriction, ie., by allowing an extension operation as an alternative for C-reduction. Figure 13 demonstrates this modification.

Restriction (b), in the definition of the extension operation, specifies that the literal to be resolved with the selected literal of the center chain can be any literal of the input parent chain. This is different from SL-resolution and the ME procedure which always select the first literal of the input parent chain during an extension operation. The significant contribution of restriction (b) is that the generation of matrix chains for each input clauses is unnecessary which means a saving of memory.

Restriction (iv) of the GC procedure prunes redundant derivations from the search tree. However, it may take one or two inference steps first before a resolvant is found to be redundant. Take, for instance, the resolvant to the left of the fifth center chain in Figure 8. This resolvant chain is actually admissible at this state. It will become inadmissible only after reducing ~Q and applying an extension to resolve on the B-literal R. This slows down the derivation. This can be improved by imposing the second preadmissibility restriction of the ME procedure with slight modification. The additional restriction can be stated this way - "A chain should not contain any B-literal which is identical to any of the preceeding non-B-literals". This additional restriction may serve as a preemptive version of restriction (iv).

2.7. Selective Linear Model (SLM) Inference System

Shostak (1976) noted that the added selectivity of the ME procedure (Loveland 1969) and SL-resolution (Kowalski and Kuehner 1971) may cause repeated refutations of the same literal. This problem was alleviated in the GC procedure (Shostak 1976) by recycling truncated A-literals as C-literals. This solution, however, does not solve the problem of producing certain irrelevant branches of the search tree which can be detected only at a later part of the derivation. Take, for instance, the search tree depicted in

Figure 14(i). There are three resolvants produced by extending on `~p(X)` of the second center chain. Two of these resolvants are redundant. This redundancy is only detected later in the derivation by the GC procedure. Had the literal `~r(X)` been resolved on before `~p(X)`, the generation of these two redundant resolvants would have been avoided. The ME procedure, SL-resolution and the GC procedure have this problem because the selection function must select a literal from the rightmost cell. Brown (1974) developed a linear derivation strategy, called the Selective Linear Model (SLM), that solves this problem.

The SLM derivation strategy has an additional operation, aside from the extension, reduction and truncation operations used in the ME procedure, SL-resolution and the GC procedure, which spreads B-literals whose indices are 0 onto different branches of the center chain. This scheme allows each spread literal to be refuted concurrently with the others, by interleaving the operations on the branches. This makes the selection of literal to be resolved on during an extension operation more flexible because it is not necessary to completely resolve away one literal before considering of another. Figure 14 (ii) shows that the spreading and the concurrent consideration of the branches can prevent redundant resolvants.

Truncated A-literals indexed by 1 are recycled in the SLM derivation strategy in the same manner as C-literals in the GC procedure. When such an A-literal is truncated, the negation of this A-literal (with index changed to 0) is inserted at its depth. This inserted literal is still classified as an A-literal and can be used later in a reduction operation.

A distinguishing feature of SLM is the use of semantic information in derivations. The general advantage of using semantic information is that it will restrict the generation of resolvants to those that are highly likely to be relevant by taking into account the intended meaning of the literals. SLM differs from the GC procedure because it does not recycle truncated A-literals indexed by 0. This is done to prevent applying ancestor

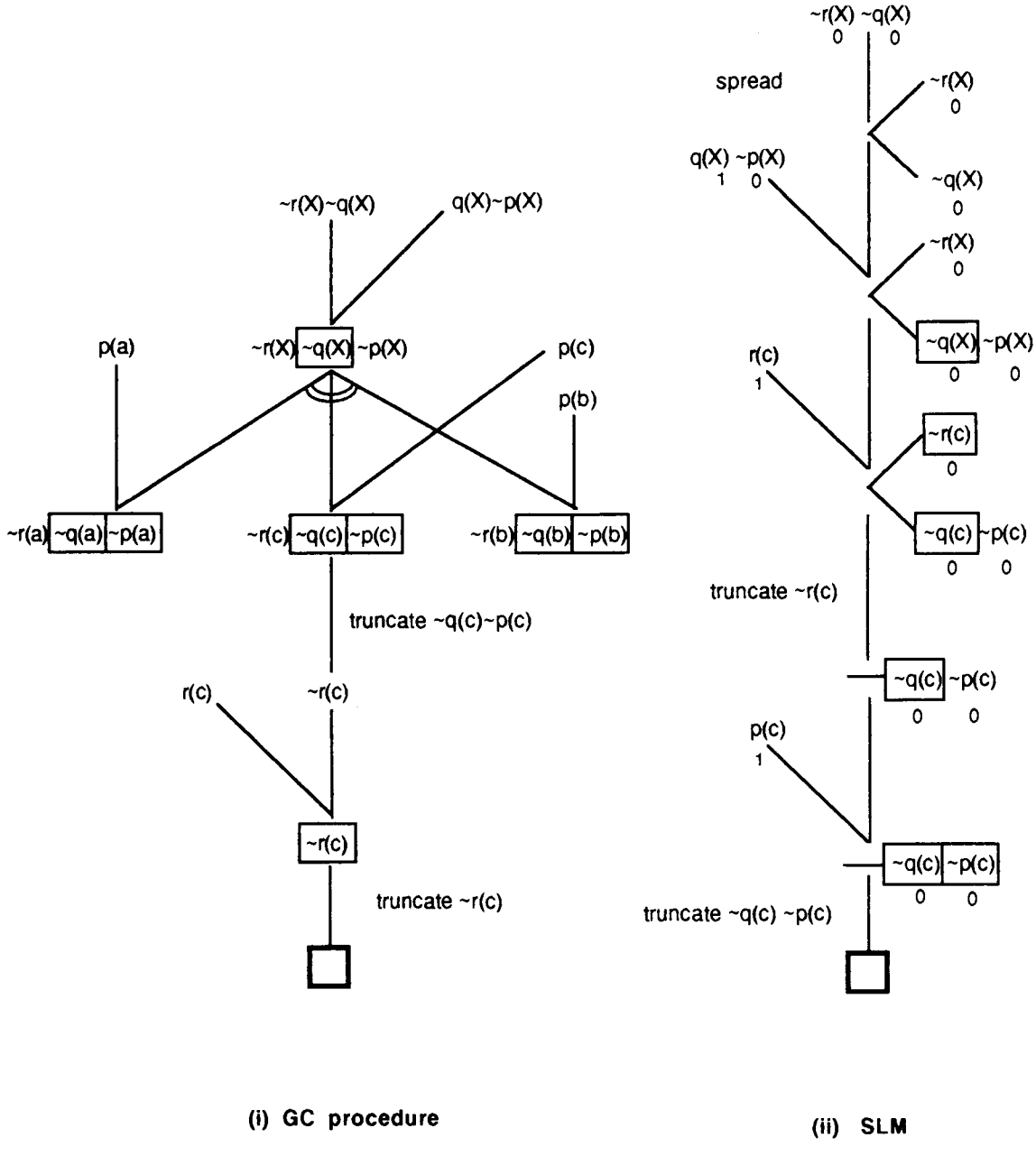


Figure 14. Search trees for S = { ~r(X)~q(X), q(X)~p(X), p(a), p(b), p(c), r(c) } using (i) the GC procedure and (ii) the SLM inference system.

resolution to B-literals indexed by 0 with recycled A-literals which were originally indexed by 0. It should be pointed out that this type of ancestor resolution has similarity to factoring of B-literals indexed by 0. Brown (1974, p. 4) cited that no generally used programming language factors its procedure invocations (a B-literal indexed by 0 is regarded as procedure invocation). Henschen (1974) has also shown that factoring is not necessary in obtaining a refutation for a set of Horn clauses. This principle is generalised in SLM in the case of a set of clauses which has a Horn model and the top clause is false in the model. The reduction operation is dropped in this situation, thereby reducing the size of the search tree. Recently, Plaisted (1989) proposed a positive refinement of the ME procedure, as cited by Nie (1990, p. 2). The basic idea of the refinement is to perform reduction operation only on negative subgoals. A negative subgoal is equivalent to a B-literal indexed by 0 in SLM using the $\mathbf{I_0}$ interpretation (see section 3 for a detailed description of the $\mathbf{I_0}$ interpretation). Nie (1990, p. 2), who implemented the refinement, has shown that selective reduction of subgoals performs better.

Brown (1974, p. 10) claimed that the complexity of SLM refutation is bounded by the complexity of the simplest hyper-minimal M-clash refutation (h M-clash refutation). He defines hyper-minimal M-clash refutations are those refutations which are obtainable from M-clash semantic trees. The M-clash semantic tree is defined in (Kowalski and Hayes 1969).

2.7.1. Formal Definition

2.7.1.1. Input Clauses Conversion

A set of input chains M is derived from the given set of input clauses S by indexing each literal of each clause, using a given interpretation, **I**, and by choosing exactly one sequence of literals in the clause. Each literal in the chain is classified as B-literal. A literal L has an index 0 if for all substitutions θ, $L\theta$ is false in the given interpretation **I**. If all instances $L\theta$ are true in **I**, then the index value is 1. Otherwise the index of L is 2 (i.e.

if neither of the two tests terminate within a specified time). For each clause that contains a literal L indexed by 2, new clauses are created, which will replace the previous clause, by changing the index of L to 1 if there exists a ground substitution θ such that $L\theta$ is TRUE in **I** and to 0 if there exists a ground substitution θ such that $L\theta$ is FALSE in **I**. If there exist both TRUE and FALSE instances of L, then both indices are used. A clause is created for each possible index of L together with the other literals of the original clause. Take, for example, the clause $p(X)_2q(X)_2$ (the truth values of literals p(X) and q(X) cannot be determined yet, hence the subscript 2). If there exist two substitutions θ_1 and θ_2 such that $p(X)\theta_1$ and $q(X)\theta_2$ are TRUE, and $p(X)\theta_2$ and $q(X)\theta_1$ are FALSE in **I**, then the four clauses; $p(X)_1q(X)_1$, $p(X)_0q(X)_1$, $p(X)_1q(X)_0$ and $p(X)_0q(X)_0$ will replace the original clause.

2.7.1.2. Derivation Definition

A selection function of SLM is a function which chooses a tip node of some branch in a center chain, extracts the rightmost cell of the selected tip node and selects a permutation of that cell. The rightmost literal of the selected permutation is the selected literal.

Given a set of input clauses S converted into set of input chains M, a support set T (a subset of M), an interpretation **I** and a selection function ϕ, a sequence of chains $C_1,\ldots, C_n$ is an SLM derivation if restrictions i), ii), iii), and iv) are satisfied.

i) $C_1,\ldots, C_n$ is a linear derivation and $C_1 \in T$.

ii) Each C_{i+1} is obtained from C_i by either

   a) spreading,

   b) extension,

   c) reduction, or

   d) truncation.

iii) No two A-literals indexed by 0 on any branch of any chain have identical atoms unless an A-literal indexed by 1 occurs between them (Hyper Minimality).

iv) C_{i+1} must be obtained from C_i by reduction if restrictions a), b), anc c) are satisfied.

a) Reduction is possible.

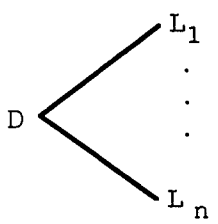
b) The A-literal L and the B-literal K used in the reduction have identical atoms.

c) No A-literal indexed by 1 occurs between L and K.

C_{i+1} is obtained from C_i by **spreading** only if :

a) Truncation is not possible

b) There is more than one B-literal indexed by 0 in $\phi(C_i)$. Let the B-literals indexed by 0 in $\phi(C_i)$ be $L_1,..., L_n$. Let D be obtained by deleting $L_1,..., L_n$ from $\phi(C_i)$. The cell is then replaced by the tree:



and combined with the other nodes to obtain C_{i+1}.

c) The classifications and indices of every literal in C_{i+1} remain as they were in C_i.

C_{i+1} is obtained from C_i by **extension** with an input chain B only if :

a) Truncation is not possible

b) Spreading is not possible.

c) C_i and B share no variables.

d) There exists a literal K in B such that the selected B-literal from $\phi(C_i)$ and K are complementary by unification with mgu θ. Let the selected B-literal from $\phi(C_i)$ be L.

e) The sum of the indices of the two literals must equal 1.

f) The literal K is deleted from B and the remaining literals of B are appended to the right of $\phi(C_i)$. The substitution θ is applied to the result in obtaining C_{i+1}.

g) There exists a ground substitution σ such that for each literal J in C_{i+1}, if J is indexed by 0 then $J\sigma$ is FALSE in **I** and if J is indexed by 1 then $J\sigma$ is TRUE in **I**.

h) $L\theta$ in C_{i+1} is classified as an A-literal. The classification of all other literals, and indices of all literals remain as they were in C_i and B.

i) If L is indexed by 1 then a depth is associated with the new A-literal, set to the left of the leftmost literal in the root of C_{i+1}.[1]

C_{i+1} is obtained from C_i by **truncation** only if :

a) Reduction is not possible

b) The rightmost literal of the rightmost node of some branch of C_i is an A-literal L.

c) L is deleted from C_i. If there are no more literals in the node then the node is automatically deleted.

d) If the A-literal L was indexed by 1 then the complement of L classified as an A-literal is inserted either :

   1) at its depth, or

---

1 In Brown's paper, it is the literal K which is associated with the depth. This is wrongly stated because a depth is associated with an A-literal. Since the newly created A-literal is $L\theta$, the depth should be associated with $L\theta$.

2) immediately to the right of any A-literal indexed by 0 occurring between the position of L and the depth of L.

e) The classification of every literal in C_{i+1} remains as it was in C_i. In the case of an insertion of new A-literal, the index of the inserted A-literal is 0. The indices of all other literals remain as they were in C_i.

C_{i+1} is obtained from C_i by **reduction** only if either I or II is satisfied.

I) The last non-reduction operation was a truncation operation of an A-literal whose index was 1. Let L be the inserted A-literal of the last truncation operation. Then restrictions a) to e) must be satisfied.

a) There exists a B-literal K to the right of L in C_i such that L and K are complementary by unification with mgu θ.

b) The sum of the indices of L and K is equal to 1.

c) The B-literal K is deleted from C_i and the substitution θ applied to the result.

d) Same as (g) of the extension operation.

e) The classifications and indices of every literal in C_{i+1} remain as they were in C_i.

f) The depth of every A-literal indexed by 1 occurring to the right of A-literal L is set immediately to the right of L iff the current depth is to the left of L.

II) The last non-reduction operation was an extension operation. Let D denote the new cell introduced by this extension operation. The restrictions a) to e) must be satisfied.

a) There exists a B-literal K in D and an A-literal L anywhere to the left of K and in the same branch such that L and K are complementary by unification with mgu θ.

b) - f) are as in (I) above.

2.7.2. Semantic Checking

Restriction (f) of the extension operation, which is also used in the reduction operation, applies semantic checking to each literal in the center chain using the given interpretation. The semantic checking ensures that there exists a ground instance consistent with the index of the literal.

SLM can use the trivial interpretation which interprets all positive literals as true and all negative literals as false. This interpretation will be referred as $\mathbf{I_0}$ hereafter. It must be noted that the restriction - "the sum of indices must equal 1", used in the extension and reduction operations will lose its effect, as the restriction will necessarily be satisfied.

Semantic checking can narrow down the search tree more if a non-trivial interpretation is available. However, to establish a non-trivial interpretation for a given set of clauses is not an easy task. Henschen (1976, p. 820) presented two reasons why semantic information is not widely used in theorem provers. Firstly, it is difficult to determine whether or not a clause containing variables is falsified, especially for interpretations whose domains are not fairly small. The second reason is the problem of finding a general representation of an interpretation with reasonable storage requirements. The interpretation $\mathbf{I_0}$ has neither of these problems and is easy to specify.

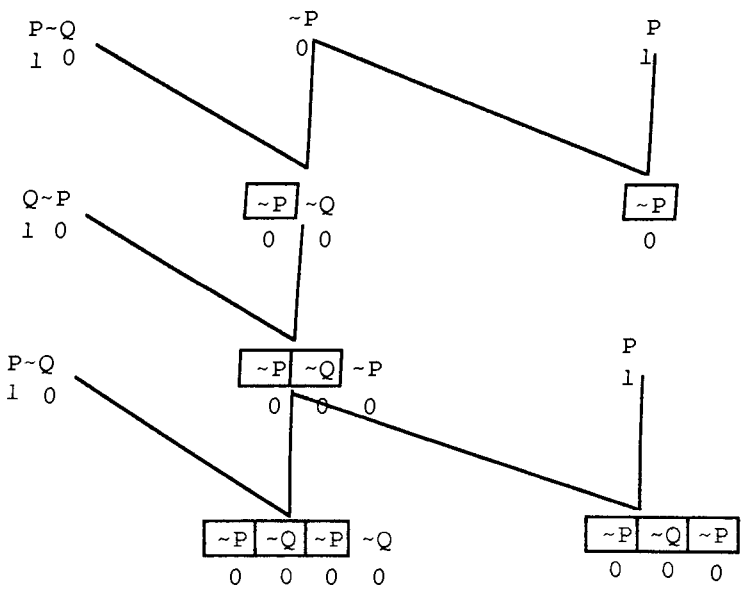
2.7.3. Example Problems

Figure 14 (p. 61) shows the search trees for the set of clauses `S = { ~r(X)~q(X), q(X)~p(X), p(a), p(b), p(c), r(c) }` using (i) the GC procedure and (ii) the SLM derivation strategy with the interpretation $\mathbf{I_0}$. This problem gives evidence of the advantage of spreading the center chain into subchains and interleaving the refutation between subchains. Figure 15 demonstrates the inadequacy of

SLM in detecting endless loops that may occur in a derivation. The set of clauses is `S={~P~Q,PQ,P,Q}`, with the top clause `~P~Q` and using the interpretation $\mathbf{I_0}$. Figures 16 and 17 present the search trees for the set of clauses `S = { ~A, ~D~E, ~C~P, AD~C, CD, E~F~G, FC, GC~Q, QC, P}` using SLM with the interpretation $\mathbf{I_0}$, and the GC procedure respectively. Figure 18 presents the search tree of the set of clauses `S={~p(a), q(a), r(b), t(b), s(X)~q(X), s(Y)~r(Y), p(X)~s(X)~t(Y)~s(Y)}` using SLM with the non-trivial interpretation that each predicate of the set `{p(a),q(a),r(b),t(b),s(a),~s(b)}` is TRUE and all others are FALSE.

2.7.4. Effects of the Restrictions

The hyper minimality restriction (iii) has three effects on derivations. Firstly, it prevents some loops in derivations. Take, for instance, the following derivation from the set of clauses $S = \{\sim P_0, P_1\sim Q_0, \sim P_0 Q_1, P_1\}$, with the chosen top clause $\sim P_0$:



The hyper minimality restriction will declare the fourth center chain inadmissible, thus preventing a loop. This causes the derivation to backtrack and try other possible resolvants of the previous extension operations. The derivation has to backtrack to the

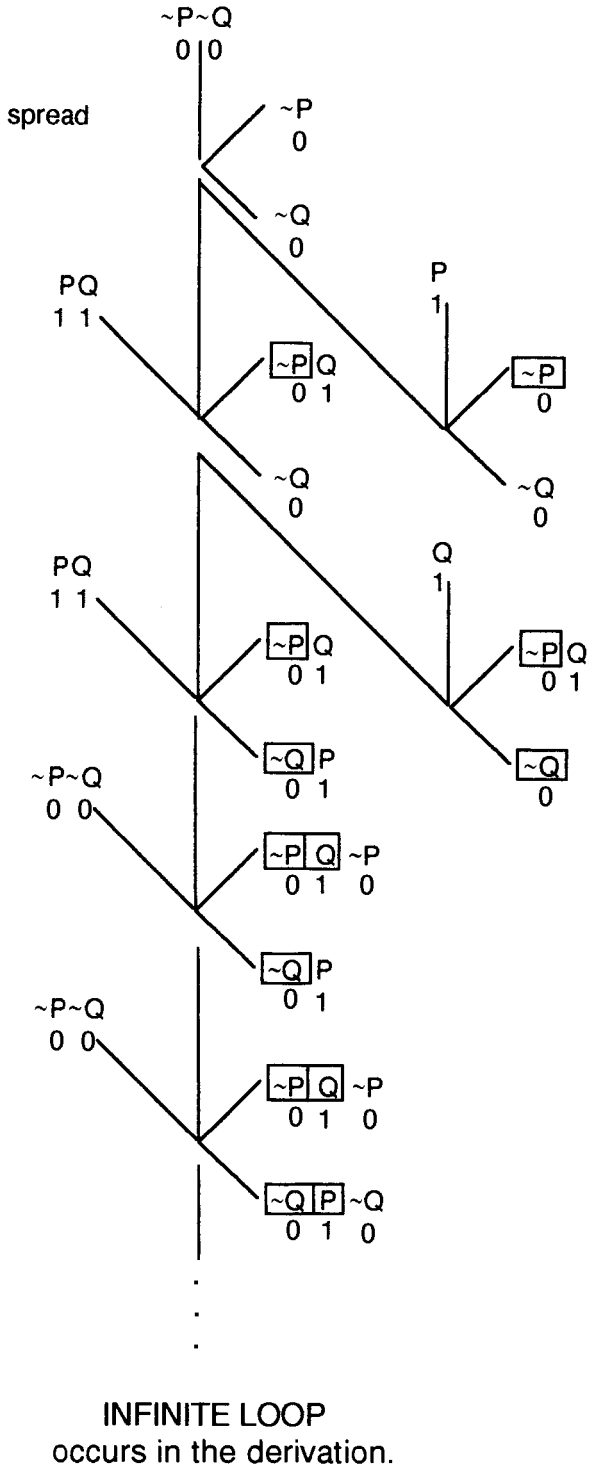


Figure 15. The search tree for S = {~P~Q, PQ, P, Q} using SLM with the interpretation $\mathbf{I_0}$

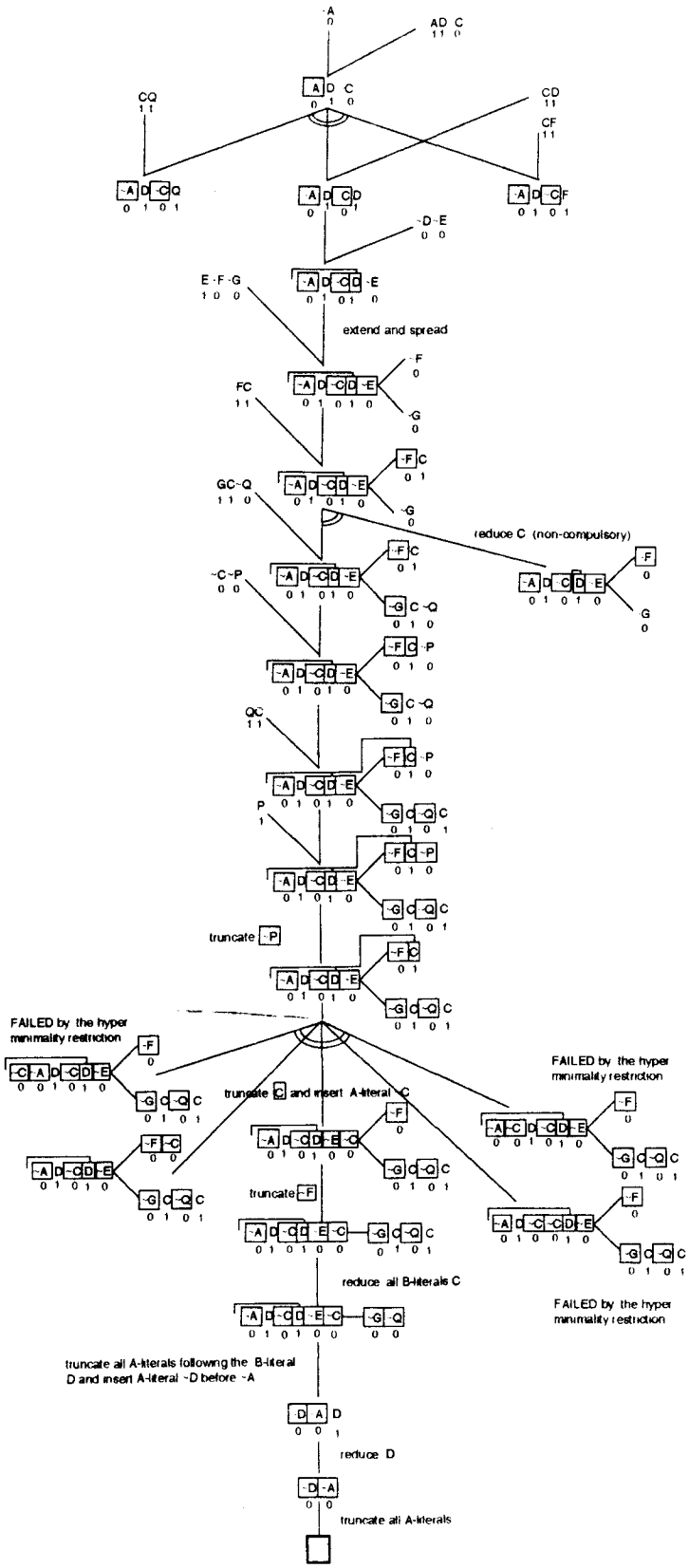


Figure 16. The search tree for S = {~A, ~D~E, ~C~P, AD~C, CD, E~F~G, FC, GC~Q, QC, P} using SLM with the interpretation $\mathbf{I_0}$

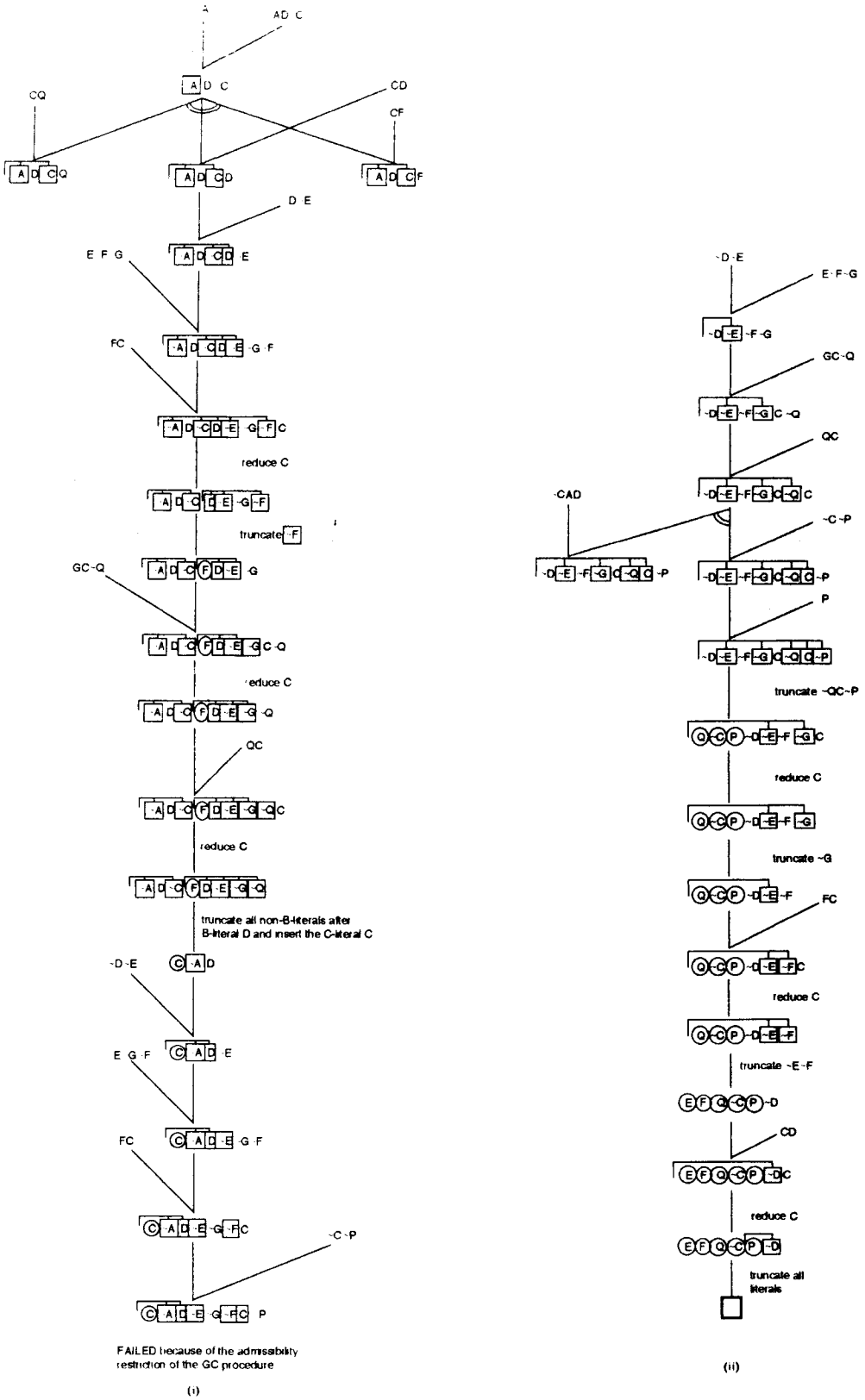


Figure 17. The search tree for S = {~A, ~D~E, ~C~P, AD~C, CD, E~F~G, FC, GC~Q, QC, P} using the GC procedure

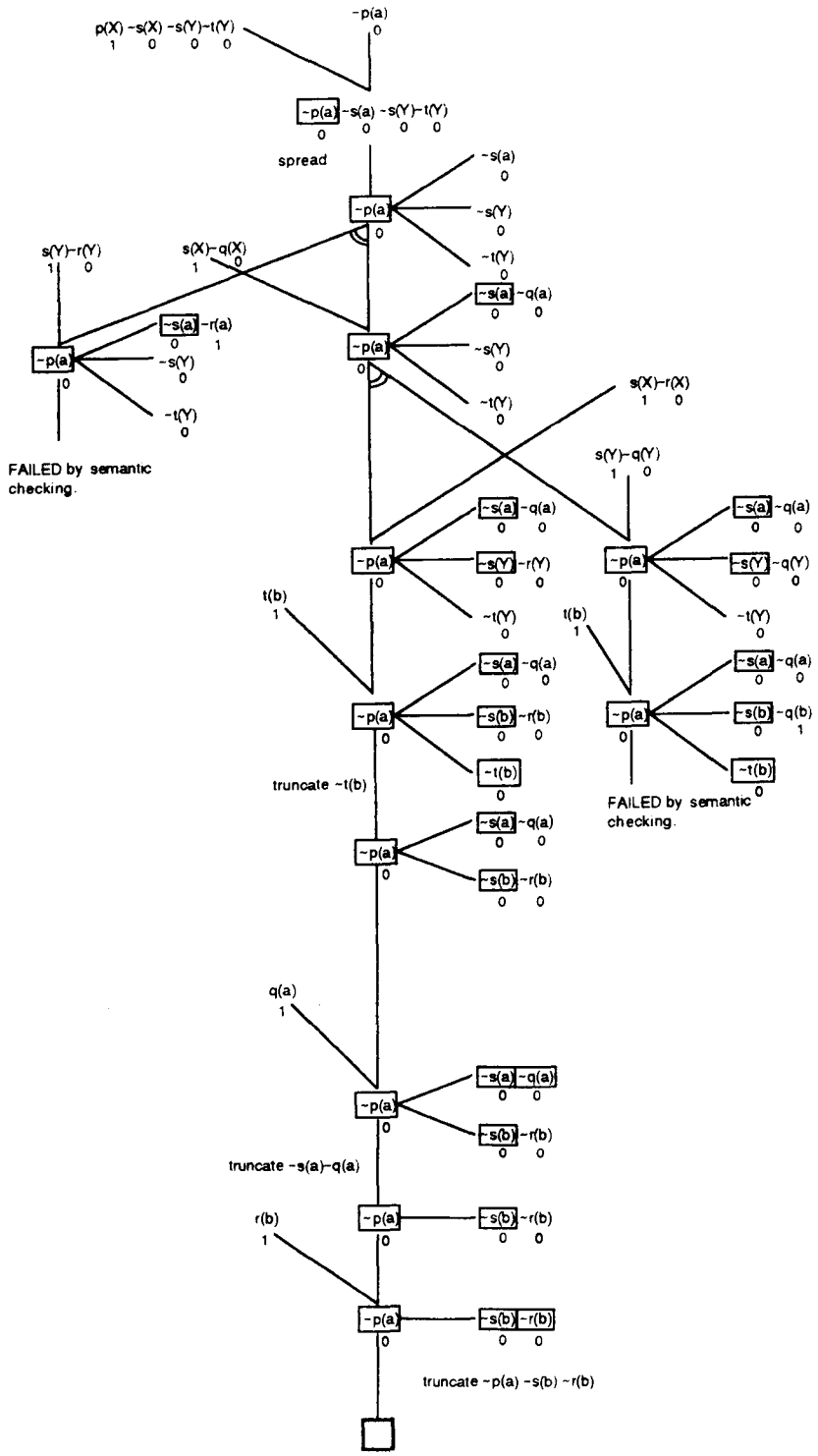


Figure 18. The search tree for S = {~p(a), q(a), r(b), t(b), ~q(X)s(X),~r(X)s(X), ~s(X)p(X)~t(Y)~s(Y)} using SLM with the interpretation I = {p(a), q(a), r(b), t(b), s(a), s(b)} as all TRUE, and others as FALSE.

third center chain and apply the extension operation using the input chain P_1. Again, the resolvant obtained is still inadmissible. It then backtracks to the first center chain (there is no alternative for the second center chain) and applies the extension operation using the input chain P_1. This time the resolvant is admissible and it leads to a minimal refutation. The hyper minimality restriction, however, does not trap all possible causes of loops. It is insufficient in detecting loops, especially for sets of clauses that contain clauses which have more than one literal indexed by 1. Take, for instance, the set of clauses S = $\{\sim P_0 \sim Q_0, P_1 Q_1, P_1, Q_1\}$ whose search tree is shown in Figure 15. SLM cannot detect that the sixth center chain will lead to an endless loop. Without the aid of a good search strategy, SLM would not obtain a refutation from this set of clauses. The ME procedure, SL-resolution and the GC procedure can easily detect loops in such a derivation because they restrict the occurrence of two non-B-literals having identical atoms. This restriction is not imposed in SLM because it is in conflict with the hyper minimality restriction. To illustrate this conflict, consider the search tree shown in Figure 16. If two A-literals having identical atoms are not allowed to exist in the same branch, the eighth center chain, which is the resolvant of applying the extension operation on the seventh center chain, is inadmissible. This forces the reduction of the B-literal C with the A-literal ~C. By doing so, the depth associated with the A-literal D will be moved to the right of A-literal ~C. The remaining B-literal D then has to be resolved on the same way as the previous B-literal D.

The second effect of the hyper minimality restriction is that it reduces some of the irrelevant derivations obtained by the indeterminancy of inserting a truncated A-literal indexed by 1, during a truncation operation. An example of this is the truncation of A-literal C at the eleventh center chain of Figure 16. Three of the chains obtained by the truncation are declared inadmissible by the restriction.

The third effect of the hyper minimality restriction is to ensure that the insertion of an A-literal, which is the complement of a truncated A-literal indexed by 1, has maximum effect in an SLM derivation in terms of compulsory reductions. To illustrate this effect, consider the truncation of the A-literal C of the eleventh center chain

of Figure 16. The hyper minimality restriction will not allow the insertion of A-literal ~C at the depth position of the truncated A-literal C nor in any position to the left of A-literal ~E. The truncation operation will then insert the A-literal ~C to the right of the A-literal ~E. The insertion of A-literal ~C at this position will then satisfy the third restriction of a compulsory reduction. Consequently, all B-literals C on the right side of A-literal ~E must be removed by reduction.

Restriction (iv) defines the restrictions of a compulsory reduction. Restriction (iv.b) specifies that the literals involved in the reduction should have identical atoms. Compulsory reduction of literals having non-identical atoms is an incomplete derivation strategy. The ME procedure, SL-resolution and the GC procedure also force a reduction on literals having identical atoms. However, there is a difference between SLM and these three derivation strategies, in the sense that the ME procedure, SL-resolution and the GC procedure can retrospectively check if a literal should have been reduced. Consider the center chain $[\sim r(a)_0][t(b)_1][\sim p(a)_0][q(X)_1]p(X)_1$ (the subscript associated with each literal is the truth index of the literal) with the only possible input chain $\sim t(b)_0 \sim p(a)_0$. The ME procedure, SL-resolution and the GC procedure will force a reduction operation on the B-literal $p(X)_1$ with the A-literal $[\sim p(a)_0]$ because the resolvant obtained from the extension operation is inadmissible to the restrictions imposed in the three derivation strategies. On the other hand, the resolvant of applying an extension operation to the center chain is still admissible in SLM.

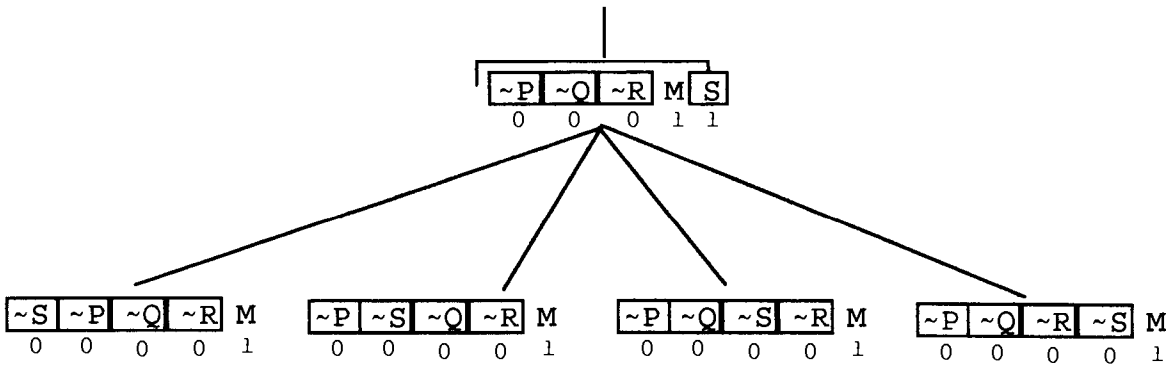
Restriction (iv.c) of compulsory reduction requires that there must not be an A-literal indexed by 1 in between the two literals. The purpose of this restriction is to preserve as much as possible the depth associated with a TRUE A-literal. The advantage of this is to maximise the usefulness of an inserted A-literal after truncating a TRUE A-literal. To demonstrate this effect, consider the search tree shown in Figure 16. Had the B-literal C of the sixth center chain been reduced, the depth associated with the A-literal D would have been moved to the right of A-literal ~C. Obviously, the creation of A-literal ~D after truncating the A-literal D is useless since it will be inserted at the right of A-

literal ~C and truncated without serving its purpose. As shown in the example, by keeping the depth of D at its original position the B-literal D of the fifteenth center chain is removed by compulsory reduction instead of resolving away the literal the same way as the previous B-literal D. Comparing the result with the GC procedure shown in Figure 17 (i), resolving away the literal D is repeated by the GC procedure because the C-point associated with the A-literal D is moved to the right of A-literal ~C during the reduction operation at the sixth center chain. In this example, the GC procedure will fail to obtain a refutation from the chosen top clause ~A. It will then choose another top clause from the set of support, as shown in search tree (ii) of Figure 17. As shown, using the top clause ~D~E leads to a minimal refutation. This is advantageous if the objective is to obtain a minimal refutation. However, it may take more time to obtain a minimal refutation because some of the inference steps may be wasted in the process of searching for a minimal refutation, similar to what happen in Figure 17 (i). Another issue to consider is the ability to obtain a refutation using a specific top clause. The example shows that SLM can obtain a refutation using more of the top clauses from the set of support than the GC procedure. This partly justifies the claim of Brown (1974, p. 1) that SLM has more desirable properties for certain applications of the predicate calculus than the other derivation strategies. One specific application which requires the ability to obtain a refutation for a specific top clause is a deductive question-answering system.

Restriction (g) of the extension operation definition, using a non-trivial interpretation, can detect some redundant resolvants. This effect is demonstrated in Figure 18. When the input chain $s(Y)_1 \sim r(Y)_0$ is used in the extension of the third center chain, the resolvant is immediately pruned from the search tree because the truth index of $\sim r(Y)_0$ was changed. The resolvant obtained by extending the fifth center chain (on the right branch of the search tree) with the input chain $t(b)_1$, is also rejected because the truth index of $\sim q(Y)_0$ was changed after the instantiation. The ME procedure, SL-resolution and the GC procedure do not provide semantic checks.

The selection functions used by the ME procedure, SL-resolution and the GC procedure add more problems to early detection of redundancy because only the literals of the rightmost cell are considered in the selection. If a literal is not in the rightmost cell and cannot be resolved away, possibly as a result of instantiation, the detection of this problem has to wait until all literals to the right of that literal are resolved away. SLM partially alleviates the problem brought about by the added selectivity of these three derivation strategies by spreading false B-literals onto different branches and concurrently resolving them away. Hence, SLM may be able to detect the problem earlier than the ME procedure, SL-resolution and the GC procedure. This effect is demonstrated in Figure 14 (p. 61).

Brown (1974, p. 23) pointed out that for a set of clauses with a Horn model and a derivation with a FALSE top clause, the reduction operation and restriction (d) of the truncation operation can never be used because all the literals in the center chains are FALSE (indexed by 0) in the interpretation. However, for a set of general clauses, restriction (d) of the truncation operation may produce many irrelevant derivations. Take, for instance, the truncation of the A-literal [S_1] in the following derivation:



Three of these chains are irrelevant since they all have the same effect. The hyper minimality restriction is not adequate to prevent the indeterminancy of inserting A-literals created by truncation. In view of this, Brown suggested three methods to reduce this redundancy. These three methods are denoted SLM-1, SLM-2 and SLM-3.

2.7.5. SLM Variations

SLM-1 is obtained from SLM by modifying the truncation operation. The modification is to insert the truncated A-literal into a position only if that position is not equivalent to a position at which the truncated A-literal has already been inserted. Two positions are said to be equivalent if no B-literals occurs between the two positions. This method reduces some of the redundant derivations obtained from truncation without affecting the effectiveness of the hyper minimality and compulsory reduction restrictions. However, SLM-1 has a drawback. The example shown in Figure 19 (p. 79) demonstrates that SLM-1 will fail to obtain a refutation if the chosen top clause is ~Q~R. SLM does not have problem in obtaining a refutation of the chosen top clause, as shown in Figure 20. Brown, however, pointed out that the complexity of SLM-1 refutations remain bounded by the complexity of the simplest h M-clash refutations.

SLM-2 is obtained from SLM by omitting the hyper minimality and compulsory reduction restrictions, and by always placing an A-literal created by truncating an A-literal indexed by 1, at its depth. This method solves the indeterminancy problem of inserting an A-literal created from a truncated A-literal. However, the advantages of this modification are bought at the expense of the effects of the hyper minimality and the compulsory reduction restrictions. One disadvantage of imposing the hyper minimality restriction in SLM-2 is that it may prevent SLM-2 from obtaining a refutation using some top clauses, which could be used in an SLM derivation. To illustrate this point, consider the search tree shown in Figure 21. The tenth center chain would be inadmissible had the hyper minimality restriction been imposed in SLM-2. Obviously, all the efforts to obtain the derivation are wasted since no other alternatives are available. One negative effect of imposing the compulsory reduction restriction in SLM-2 is demonstrated in Figure 22. The reduction of B-literal G at the eleventh center chain (the resolvant obtained by extending the center chain using the input chain ~G~P will not lead to minimal refutation since the B-literal ~P is introduced) has moved the depth of the A-literal D to the right of A-literal ~G. The insertion of A-literal ~D to the right of A-literal ~G when the A-literal

D is truncated from the twelfth center chain, has forced the reduction of B-literal D at the thirteenth center chain. This reduction has moved also the depth of A-literal A to the right of A-literal ~D. Thus, the B-literal A is removed by reduction when the A-literal ~A is inserted at its depth after truncating the A-literal A from the fifteenth center chain. This causes the transfer of the depth of A-literal Q to the right of A-literal ~A. Consequently, the insertion of A-literal ~Q after truncating the A-literal Q from the eighteenth center chain is useless. Had it been inserted to the left of A-literal ~P, the B-literal Q of the twentieth center chain could have been resolved away by reduction. SLM-2 with non-compulsory reduction may still obtain the shown derivation but it has other derivations that may lead to minimal refutation such as applying extension operation to the sixteenth center chain instead of applying the reduction operation. Dropping the compulsory reduction restriction expands the search tree since an extension operation can still be performed to resolve on a B-literal when it can be simply reduced. In fact, Brown conjectured that the number of refutations in the SLM-1 search space is always fewer than in SLM-2. SLM-2 also has no loop-check since the hyper minimality restriction is omitted.

SLM-3 is obtained from SLM by deleting restriction (d) in the truncation definition. Thus, the SLM-3 truncation operation simply deletes an A-literal in the same manner as SL-resolution's truncation operation. This method still maintains the hyper minimality and the compulsory reduction restrictions. However, resolving away of literals are possibly repeated since truncated A-literals are not recycled.

2.7.6. New Variations of SLM

2.7.6.1. Transformation of non-Horn set to subsets of Horn clauses

The problems in SLM of detecting loops and the indeterminancy of inserting A-literals created from truncated A-literals will only occur when the derivation, which

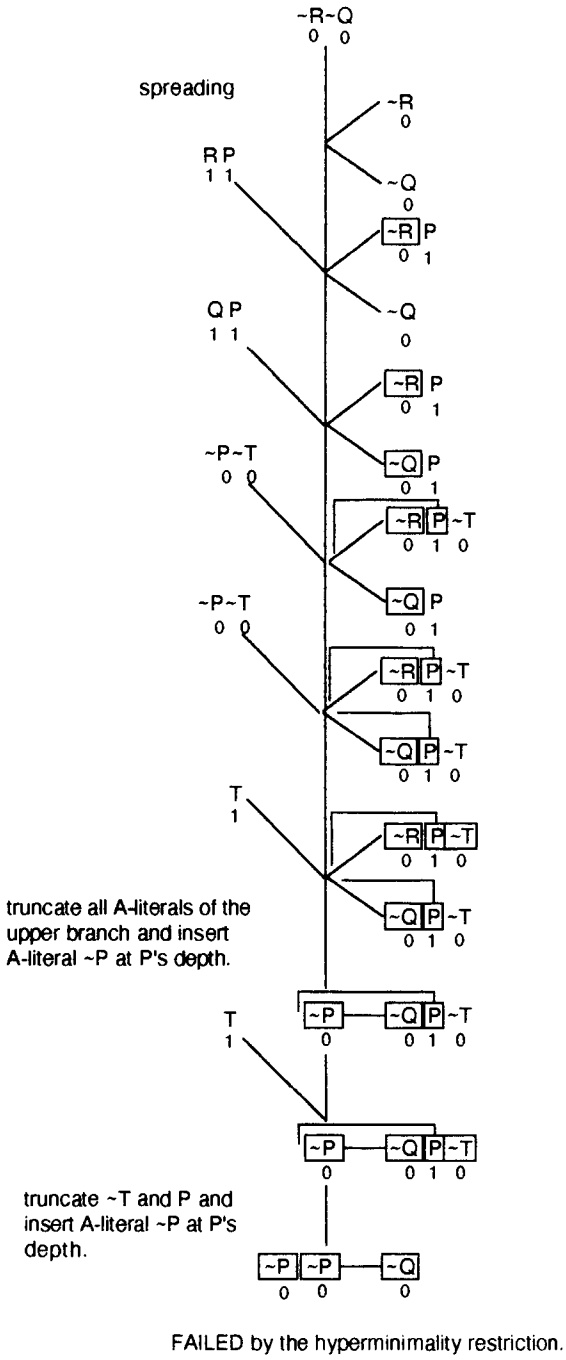


Figure 19. A search tree for S = {~Q~R, ~P~T, QP, RP, T} using SLM-1 with the interpretation $\mathbf{I_0}$.

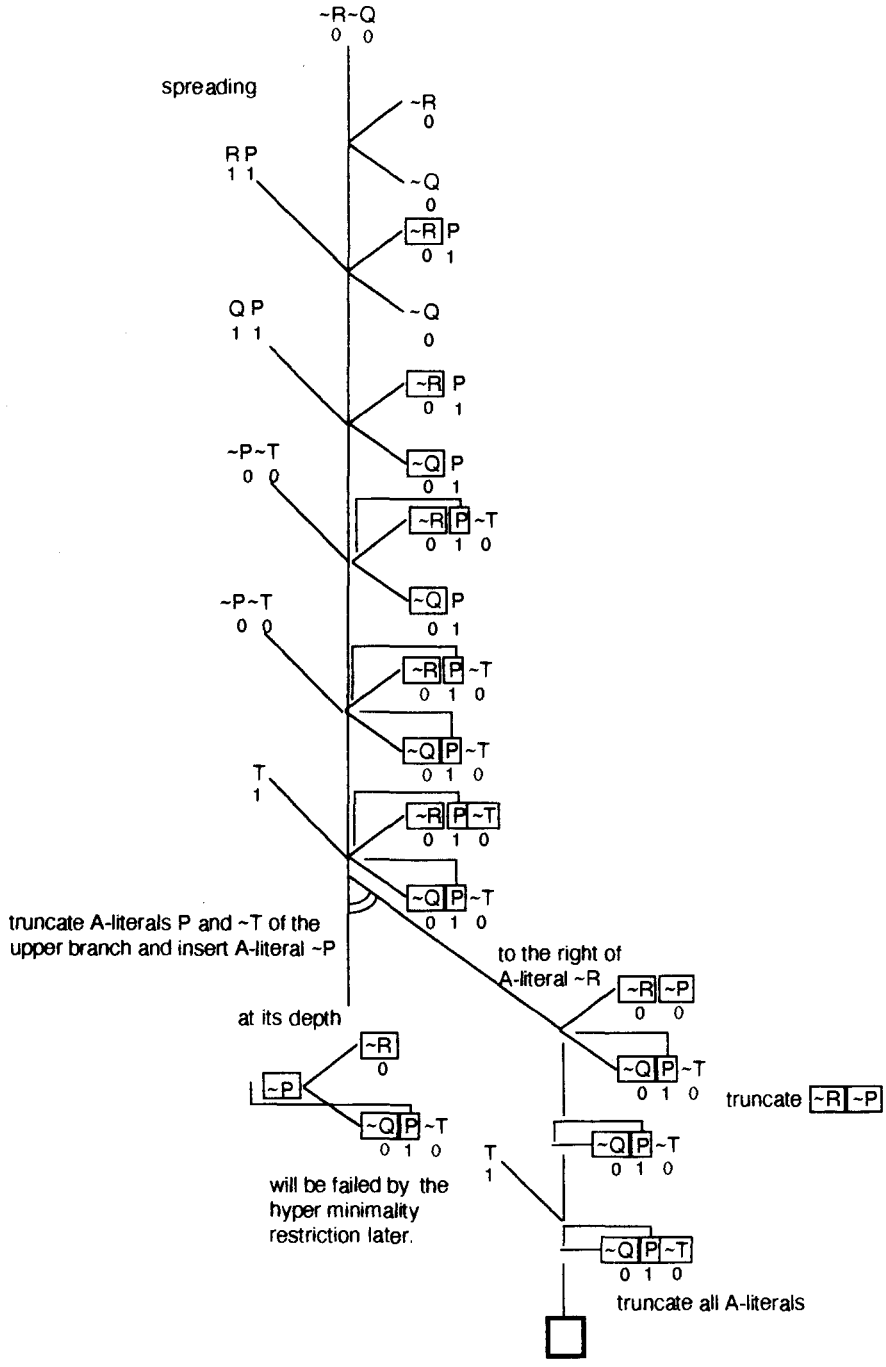


Figure 20. The search tree for S = {~Q~R, ~P~T, QP, RP, T} using SLM with the interpretation $\mathbf{I_0}$.

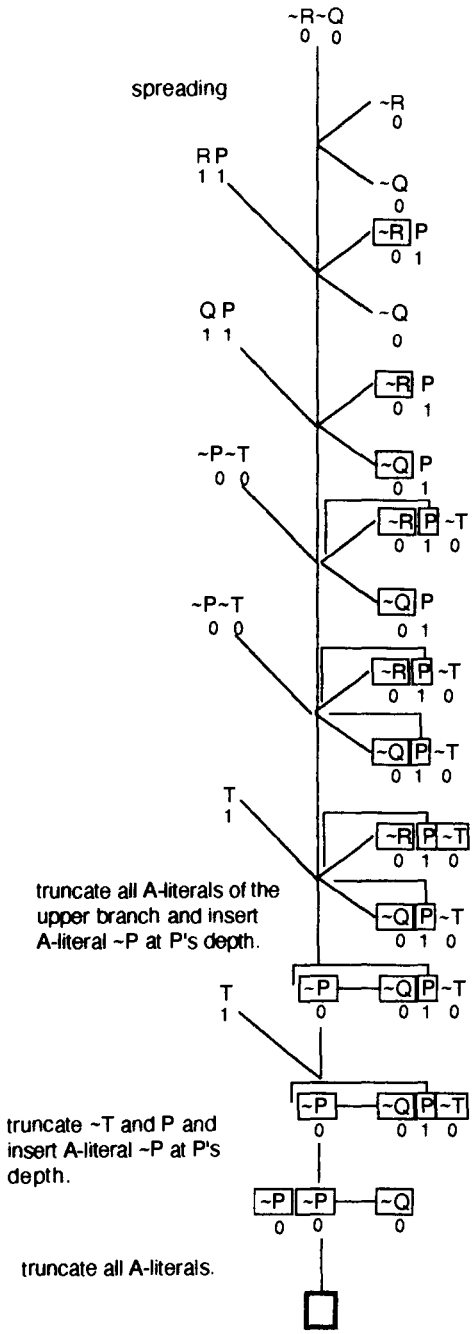


Figure 21. The search tree for S = {~Q~R, ~P~T, QP, RP, T} using SLM-2 with the interpretation $\mathbf{I}_0$.

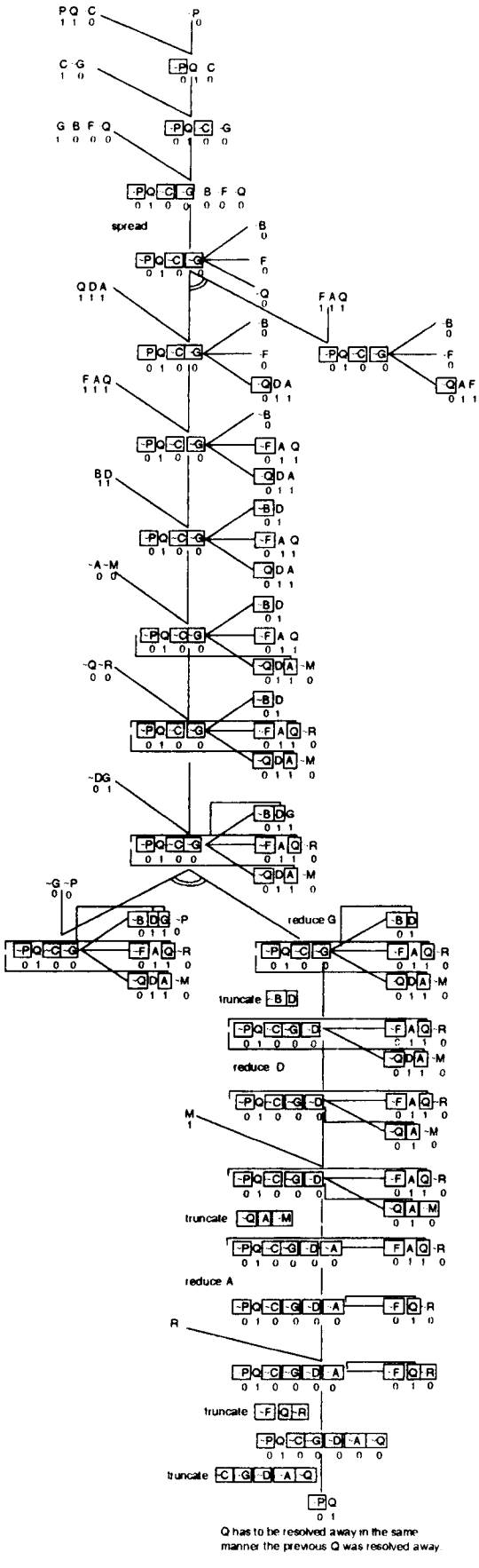


Figure 22. A search tree for S = {~P, ~G~P, ~A~M, ~Q~R, PQ~C, C~G, G~B~F~Q, QDA, FAQ, BD, ~DG, M, R} using SLM-2 (with compulsory reduction) with the interpretation I_0 starts from a FALSE top clause, has a center chain that contains at least

one A-literal indexed by 1. This happens when some of the input chains have more than one B-literal indexed by 1, i.e the interpretation used is not a Horn model of the input clauses. To prevent the occurrence of A-literals indexed by 1 in any center chains of the derivation, the set of clauses which has a non-Horn model may be broken down into subsets of clauses such that each subset has a Horn model, using the splitting technique of Chang (1972) .

The idea of splitting is to split the problem into subproblems and work on each subproblem. The splitting technique starts with a set of clauses $S \cup \{C\}$, where C is a clause to be split into two groups of literals: P and Q, i.e. $C = P \cup Q$. The splitting technique will then produce the two sets $S \cup \{P\}$ and $S \cup \{Q\}$. If consistent refutations can be derived from both of these subsets then $S \cup \{C\}$ is unsatisfiable. Consistent refutations are obtained if the substitutions applied to the common variables of P and Q in each refutation are compatible. Chang claimed that the splitting techniques can improve the proof search efficiency both with respect of time and memory, as cited by Henschen and Wos (1974, p. 591). Henschen (1976, p. 816) suggested that sets of non-Horn clauses could be transformed to subsets of Horn clauses, in order to apply to sets of non-Horn clauses a theory specifically designed for sets of Horn clauses. This technique can be applied in SLM, by splitting the set of general clauses to subsets of Horn clauses and using the interpretation $\mathbf{I_0}$ or splitting the set of clauses with a non-Horn model into subsets of clauses such that each subset has a Horn model.

Once the set of clauses is broken down into subsets of clauses, a refutation for each subset of clauses can be obtained using SLM. To increase the efficacy of SLM in obtaining a refutation from a set of clauses with a Horn model, the following modifications of SLM may be done :

a) The reduction operation is removed. The reduction operation is not needed to prove the unsatisfiability of a set of clauses having a Horn model.

b) The hyper minimality restriction will be changed to - "no two A-literals have identical atoms". There is no need to categorise A-literals by truth index since for a given set of clauses with a Horn model and a FALSE top clause, the literals of the center chain are all indexed by 0.

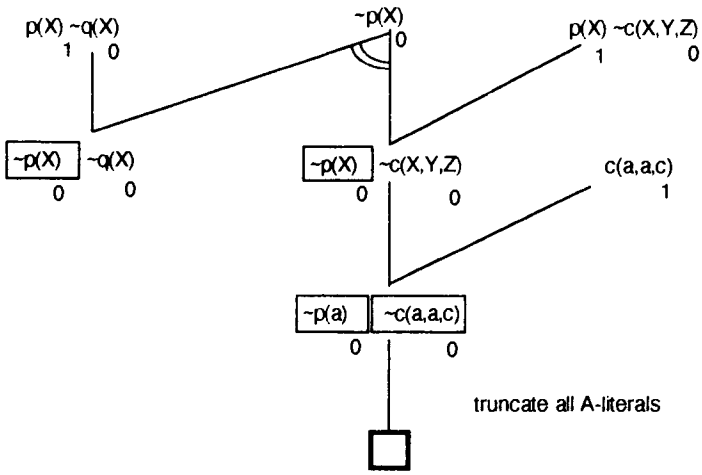
c) The truncation operation definition will be that of SLM-3.

These modifications of SLM constitute the new variation of SLM and will be called SLM-4 hereafter.

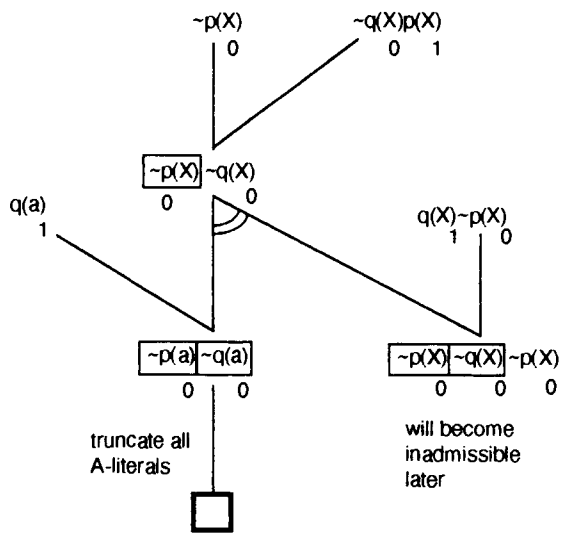
SLM-4 solves the indeterminancy of inserting A-literals, and loop detection problems of SLM. Figure 23 shows the search trees for S = $\{\sim p(X)_0$, $p(X)_1 q(Y)_1 \sim c(X,Y,Z)_0$, $\sim p(X)_0 q(X)_1$, $\sim q(X)_0 p(X)_1$, $c(a,a,c)_1\}$ using SLM-4. The chain $p(X)_1 q(Y)_1 \sim c(X,Y,Z)_0$, which contains more than one literal indexed by 1, is selected to be split. The chain is then split into $p(X)_1 \sim c(X,Y,Z)_0$ and $q(Y)_1$. The first subset S_1 is then formed by combining the first part of the split, $p(X)_1 \sim c(X,Y,Z)_0$ and the rest of the chains of S. Figure 23(i) shows the search tree of the first subset of S. The refutation has instantiated the variable Y with the constant a. To obtain a consistent refutation, the second subset of S is formed by the subcase hypothesis $q(a)_1$ which is the instance of $q(Y)_1$ derived after refuting the first subset, and the rest of the chains of S. The refutation of S_2 is shown in Figure 23(ii). Since S_1 and S_2 have consistent refutations, S is then unsatisfiable. Figure 24 shows the search tree for S using SLM. Clearly, in this example, SLM-4 has no problem of detecting loops and preventing irrelevant derivations obtained during a truncation operation.

2.7.6.2. Adding more restrictions to SLM

The problem with the splitting technique is the complexity of splitting a set of clauses that contains many clauses which have more than one literal indexed by 1. The



(i) The search tree for S_1 = {~p(X),p(X)~c(X,Y,Z),~q(X)p(X),~p(X)q(X), c(a,a,c)}



(ii) The search tree for S_2 = {~p(X),q(a), ~q(X)p(X),~p(X)q(X), c(a,a,c)}

Figure 23. The search trees of the two subsets of S = {~p(X), p(X)q(Y)~c(X,Y,Z), ~q(X)p(X), ~p(X)q(X), c(a,a,c)} using SLM-4 with the interpretation $\mathbf{I_0}$.

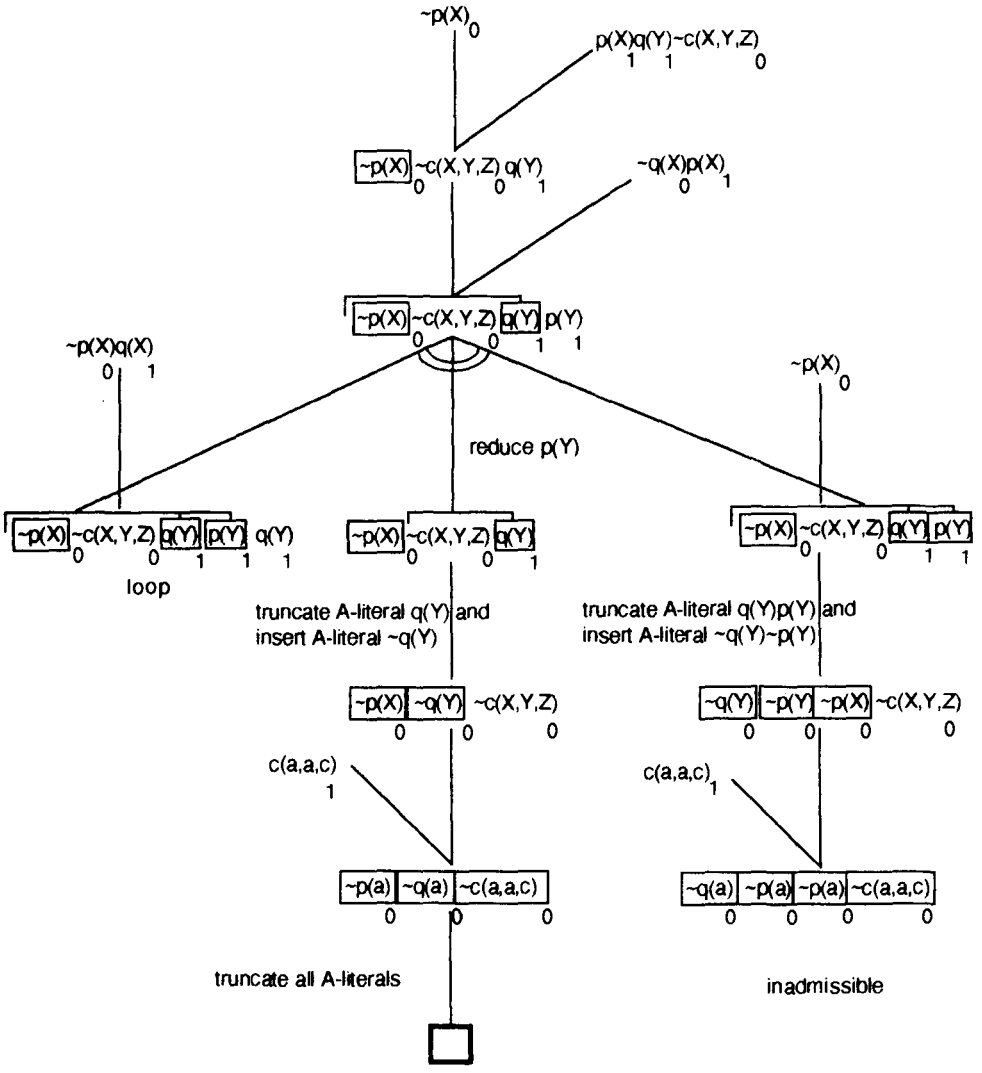
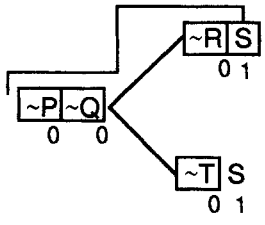


Figure 24. The search tree for S = {~p(X), p(X)q(Y)~c(X,Y,Z), ~q(X)p(X), ~p(X)q(X), c(a,a,c)} using SLM with the interpretation $\mathbf{I_0}$.

difficulty of maintaining consistent refutations for numerous sets adds to the difficulty of implementing such method. In view of this, an alternative variation of SLM is suggested.

The problem with SLM's truncation operation is that it allows alternative points of inserting truncated A-literals, in order to get the maximum use of them. However, this causes problems since it may produce many irrelevant derivations, thereby expanding the search tree unnecessarily. Consider a branch of a center chain which contains N consecutive A-literals indexed by 0 in between an A-literal L indexed by 1, and its depth. If these N consecutive A-literals indexed by 0 are in the same node then N irrelevant derivations are produced during the truncation of L since SLM inserts the truncated A-literal at its depth or at the right of each A-literal indexed by 0 in between the depth and L. They are irrelevant derivations since inserting the truncated A-literal at its depth has the same effect as when it is inserted to the right of any of the A-literals indexed by 0 in between the depth and L. There is, however, a different effect on the derivation if the truncated A-literal is inserted at a position in one node and in another position in a succeeding node although the literals between the two positions are all A-literals indexed by 0. Consider the center chain



Obviously, there is a difference in inserting the A-literal ~S at its depth which is in the root node or inserting it to the right of A-literal ~R in the tip node. The first insertion of A-literal ~S will force the reduction of the B-literal S in the other tip node while the second insertion of the A-literal ~S to the right of A-literal ~R has different effect in the derivation. However, inserting A-literal ~S either at its depth or to the right of A-literal ~P or ~Q has the same effect in the derivation. SLM-1 attempts to reduce these irrelevant derivations by defining an equivalent position restriction. However, it was found out that

the equivalent position restriction may cause also some problems as pointed out in section 2.7.5. In view of this, a modification of the equivalent position restriction is suggested as follows :

> Two positions P_1 and P_2, where P_1 is to the left of P_2 and are in the same node, are equivalent if the literals in between P_1 and P_2 are all A-literals indexed by 0.

The definition of the truncation operation of SLM may then be modified as in SLM-1. In this scheme, the truncation of the A-literal S of the above center chain will only produce two chains while SLM will produce 4 chains. Looking at Figure 19 (p. 79), the suggested method can obtain a refutation of the chosen top clause ~R~Q because when the A-literal P is truncated at the seventh center chain, there are two possible positions that the A-literal ~P can be inserted. The first position is at its depth which is at the root node and the second position is at the right of A-literal ~R which is on the next node.

The loop detection problem of SLM is brought about by allowing two A-literals having identical atoms to coexist in the center chain in order to preserve the depth of A-literals in between them. This problem is complicated by the generalisation of classifying non-B-literals. SLM has no distinction between an A-literal produced by an extension operation and an A-literal created from a truncated A-literal. If there are different classifications of these two types of A-literals, it is easier to detect that a certain atom is repeatedly resolved on by extension operations, which is a distinguishing sign of the occurrence of endless loops. The ME procedure, SL-resolution and the GC procedure prevented this loop problem by not allowing two non-B-literals to have identical atoms. This restriction also makes the reduction operation compulsory for literals having identical atoms. However, SLM avoids such compulsory reduction in order to preserve the depths of A-literals, and imposing such a restriction on SLM would negate this preservation. However, if the restriction is loosened in such a way that it will prevent identical A-literals

created by extension operations, but will allow complementary A-literals iff an A-literal indexed by 1 is in between them, then the loop problem is prevented without hampering the hyper minimality effects. Hence, the following modifications of SLM are suggested :

a) Restrictions (b), (d) and (e) of the truncation operation definition are modified as follows:

For (b) :
The rightmost literal of the rightmost node of some branch of C_i is a non-B-literal. Let this literal be L.

For (d) :
If L is an A-literal indexed by 1 then the complement of L classified as C-literal is inserted either :

1) at its depth, or
2) immediately to the right of an A-literal indexed by 0 iff that position is not equivalent to a position at which the C-literal has already been inserted, and occurs between the position of L and the depth of L.

For (e)
The classification of every literal in C_{i+1} remains as it was in C_i. In the case of an insertion of C-literal, the index of the inserted C-literal is 0. The indices of all other literals remains as they were in C_i.

b) Restriction (iv.b) of the compulsory reduction restriction definition is changed to -

The non-B-literal L and the B-literal K used in the reduction have identical atoms.

c) The hyper minimality restriction is changed to :

There must be no two identical non-B-literals indexed by 0 on any branch of any chain unless an A-literal indexed by 1 occurs between them.

This restriction is intended to insert the C-literal at a position where it has maximum effect in the derivation in terms of compulsory reduction, and to minimise irrelevant derivations which the equivalent position restriction cannot prevent. For example, the truncation of A-literal C on the eleventh center chain of Figure 16 (p. 70). The restriction will not permit the insertion of C-literal ~C either at its depth or to the right of A-literal ~C.

d) Restrictions that will detect loops will be added. They are as follows :

1) No B-literal that is identical to any of the preceding A-literals should occur on any branch of any chain.
2) No two identical A-literals should occur on any branch of any chain.

Restriction (d.1) is similar to the second preadmissibility restriction of the ME procedure. The purpose of this restriction is to prevent loops caused by adding a B-literal which is identical to a preceding A-literal. Stickel (1984, p. 215) affirms that it is unnecessary to attempt to solve a goal (B-literal) while in the process of attempting to solve that same goal (A-literal). Restriction (d.2) is a retrospective check of the first restriction.

Restriction (iv.c), as defined in section 2.7.1.2, only forces a reduction if the literals involved in the reduction have identical atoms. However, this restriction is not enforced retrospectively. Take, for instance, the center chain $[\sim p(a)_0][q(a)_1][\sim f(a)_0][\sim g(X)_0]f(X)_1$. SLM will still admit the resolvant which is

obtained by resolving $f(X)_1$ with an input chain $\sim f(a)_0 \sim q(a)_0$. The resolvant $[\sim p(a)_0][q(a)_1][\sim f(a)_0][\sim g(a)_0][f(a)_1]\sim q(a)_0$ may produce an expanded search tree and a longer refutation than by simply reducing the B-literal $f(X)_1$. In the ME procedure, SL-resolution and the GC procedure, the resolvant is inadmissible. It should be noted that reducing the B-literal $f(X)_1$ does not affect any depths. Hence, a retrospective check of restriction (iv.c) is added to handle this situation.

e) A retrospective check of restriction (iv.c) is defined as follows :

No A- or C-literal which is exactly complementary to a following A-literal may occur on any branch of any chain unless an A-literal indexed by 1 exists between them.

In SLM, the reduction operation case (I) (after truncation of A-literal indexed by 1) may reduce a B-literal which may precede some A-literals indexed by 1. This has some negative effects since the reduction may move the depths of some A-literals unnecessarily. As shown in Figure 22 (p. 82), reducing B-literals which precede some A-literals indexed by 1 has complicated the refutation. In view of this, the reduction operation is redefined in such a way that it will not reduce B-literal until all the following non-B-literals are removed.

f) The reduction operation is redefined as follows:

C_{i+1} is obtained by reduction only if a) to f) are satisfied.

a) The last non-reduction operation is an extension operation or a truncation of an A-literal indexed by 1.

b) The rightmost cell of the selected branch contains a B-literal K and there exists a non-B-literal L which is to the left of K and in the same branch.

L and K are complementary by unification with mgu θ and the sum of their indices is equal to 1.

c) - e) same as II c) - e) of SLM reduction definition.

f) The depth of every A-literal indexed by 1 occurring to the right of the non-B-literal L is set immediately to the right of L iff the depth is to the left of L.

Another disadvantage of SLM is that it does not recycle all proved literals. Sutcliffe (1989, p. 10) in his General Clause Theorem Prover (GCTP), defines a proved literal as a logical consequence of the input clauses used thus far. If a C-literal is inserted to the left of any A- or B-literal then the C-literal is a proved literal, otherwise, it is a conditionally proved literal. SLM does not recycle truncated FALSE A-literals in order to prevent reducing a FALSE B-literal with it. Brown (1974, p. 4) justifies this by pointing out that factoring, in some cases, will add an irrelevancy to the search space. However, it will not happen if the recycled literal is a proved literal. Reducing a B-literal with a proved literal is like recalling that portion of the derivation that obtains the proved literal. Sutcliffe extends the C-literal mechanism of the GC procedure by removing proved literals from the center chain and adding them to the set of input clauses, as unit clauses. He argued that the addition of proved literals as unit clauses to the set of clauses is particularly effective in conjunction with the unit preference strategy. This has also an advantage in the environment of a consecutively bounded search since the proofs of B-literals discovered within one bound are carried over to the next bound. It is, therefore, suggested that a proved literal be added to the set of input clauses as a unit clause iff it is not subsumed by any unit clauses in the set. To maintain the effect of the hyper minimality restriction, conditionally proved literals are treated as in SLM. The following restrictions are added to the operations used in SLM to implement the suggested modifications:

f) For the extension operation :

There exists an input chain B which is either

i) a unit chain (unit preference strategy), or

ii) any chain.

If the selected literal L is indexed by 0, a status flag of 1 is associated with the new A-literal $L\theta$.

g) For the reduction operation :

The status flag of each A-literal indexed by 0 occurring to the right of the non-B-literal L is set to 0.

h) For the truncation operation:

If the A-literal L is indexed by 0 and the status flag is 1 then L is a proved literal. If L is indexed by 1 and it is not preceded by any A- or B-literal then L is a proved literal. If L is a proved literal then the complement of L with the corresponding change to its index and the classification changed to a B-literal, will be added to the set of input chains M iff it is not subsumed by any unit chains in M.

All these modifications and additional restrictions constitute a new variation of SLM, which will be called SLM-5 hereafter. Figure 25 demonstrates the efficacy of SLM-5. As shown, the resolvants obtained in applying extension operations to the fourth center chain with the input chain $\sim R_0 \sim Q_0$ and the fifth center chain with the input chain $\sim S_0 \sim P_0$ are inadmissible. They are inadmissible because loops will occur if the derivation is continued from any of these resolvants. SLM would admit these resolvants as admissible. Using the proved literal P_1 as the input chain during the extension operation of the tenth center chain has simplified the refutation. In SLM, the B-literal $\sim P_0$ of the tenth center chain has to be resolved the same way as the previous B-literal $\sim P_0$ of the third center chain.

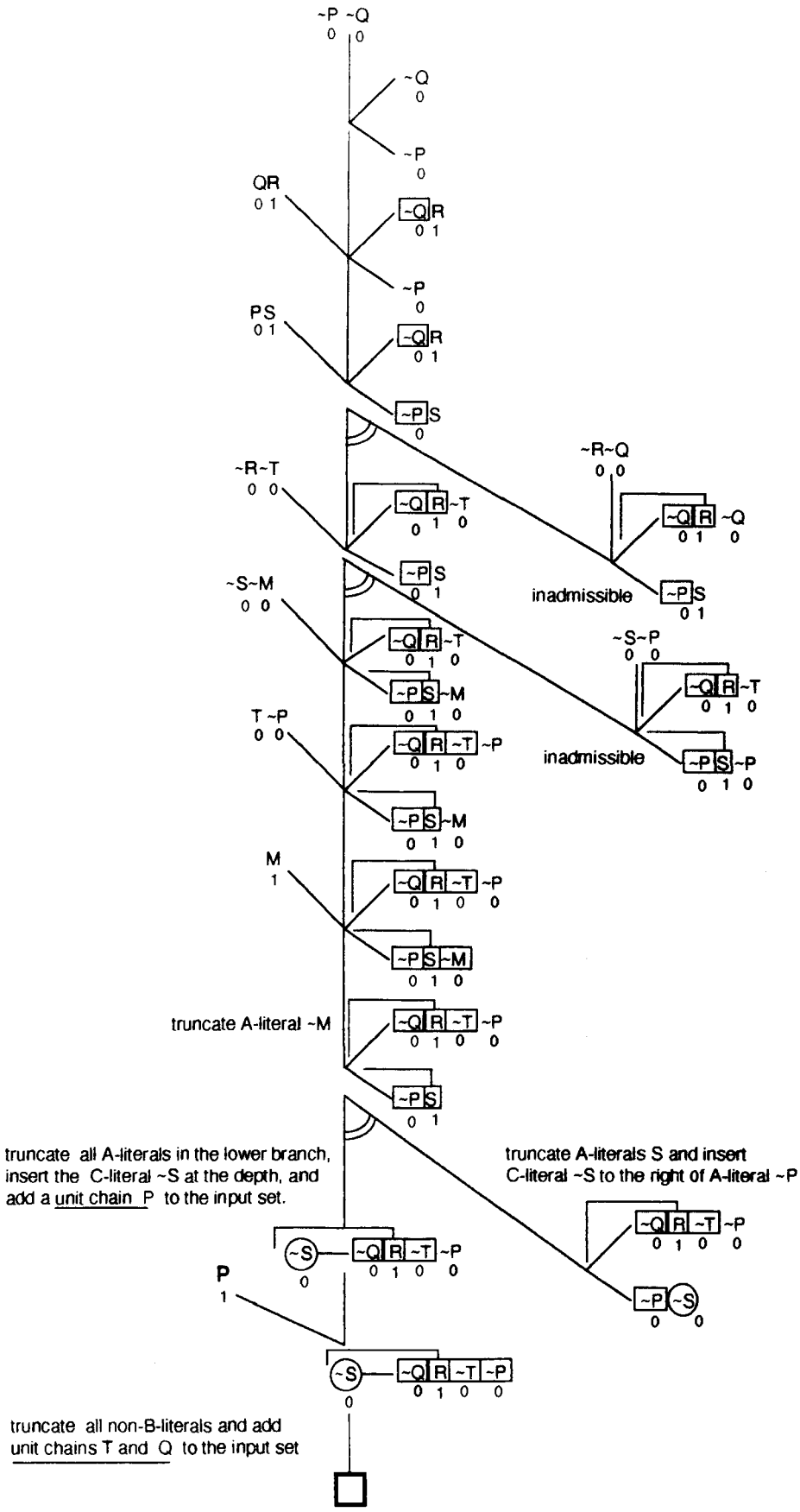


Figure 25. The search tree for S = {~P~Q, ~R~Q, ~S~P, QR, PS, ~R~T, T~P, ~S~M, M} using SLM-5 with the interpretation $\mathbf{I_0}$.

2.8. A Summary

An overview comparison of s-linear, ME procedure, t-linear, SL-resolution, GC procedure, and the SLM derivation strategies, is tabulated in Table 1.

Table 1 - Similarities and Differences of the Six Linear Derivation Strategies

Features	Derivation Strategies					
	s-linear	ME procedure	t-linear	SL-resolution	GC procedure	SLM
Clauses	Set of literals	Chain format	Set of literals	Chain format	Chain format	Chain format
Selection	Any literal in the center clause.	Rightmost B-literal in the center chain.	Any literal in the center clause.	Use a selection function to select a literal from the most recently introduced literals in the center chain.	Use a selection function to select a literal from the most recently introduced literals in the center chain.	Use a selection function to select a tip node of some branch in a center chain, and select a B-literal from the rightmost cell of the tip node.
Factoring	No, but apply merging on identical literals.	No.	No, but apply merging on identical literals.	Yes.	No, but reduction of an A-literal with a C-literal can be viewed as delayed factoring.	No, but reduction of an A-literal with a recycled A-literal can be viewed as delayed factoring.
Ancestor Resol'n	Yes. A restrictive ancestor resolution.	Yes, via reduction.	Yes. A compulsory ancestor resolution.	Yes, via reduction.	Yes, via reduction.	Yes, via reduction.
Spreading	No.	No.	No.	No.	No.	Yes.
Recyle Truncated A-literals	No.	Yes, in the form of lemmas.	No.	No.	Yes, in the form of C-literals.	Partly. Only A-literals indexed by 1 are recycled, as A-literals indexed by 0.
Semantics	No.	No.	No.	No.	No.	Yes.
No Taut's	Yes.	Partly. Tautologous center chains are not allowed if the complementary B-literals are not separated by an A-literal.	No, but is compatible with that restriction.	Yes.	No.	No.

Chapter 3

IMPLEMENTATION

3.1. Introduction

The ME procedure, SL-resolution, the GC procedure, SLM and SLM-5 have been implemented using the PROLOG language. (The s-linear and t-linear derivation strategies are not included in the implementation because of their limitations. The t-linear derivation strategy is only defined for ground derivations. In the case of the s-linear derivation strategy, it is already known that it produces bigger search trees than the other strategies.) Writing theorem provers in PROLOG is not new. Other theorem provers implemented in PROLOG language are (Santane-Toth and Szeride, 1982), (Brown, 1984), (Satz, 1988) and (Sutcliffe, 1989).

PROLOG is a special case of a theorem prover (Bratko 1986, p. 397). It is a programming language based on a specialised version of linear input resolution for Horn clauses (Clocksin and Mellish 1987). PROLOG implements the ideas of the predicate logic as a programming language (Amble 1987, p. 44), which simplifies the development of a theorem prover for first order predicate logic. Unification plays a vital role in theorem proving. However, PROLOG implements a unification algorithm without an occurs check. An occurs check is a check for an occurence of the same variable in expressions being unified, that may cause a looping substitution, i.e. a variable is repeatedly bound to a term containing the variable. This type of unification is unsound for theorem proving. Thus, there is a need to write a PROLOG procedure to handle the occurs check problem. There is also a need to modify the search strategy of PROLOG because a depth-first strategy is not an appropriate search strategy for theorem proving. Implementations of PROLOG on conventional computer architectures have achieved efficiency comparable with pure LISP (Warren et al. 1977) as cited by Kowalski (1982, p. 3). However, Stickel

(1984, p. 211) argued that writing a theorem prover in PROLOG offers uncertain advantages in comparison with writing a theorem prover in any other language, such as LISP. He pointed out that theorem prover written in PROLOG would perform slower than the speed of PROLOG, because several PROLOG inference operations would have to be performed for each theorem-proving inference operation. This is true if a theorem prover is executed using a PROLOG interpreter. With the advent of PROLOG compilers such as the Arity PROLOG compiler (Arity Corporation 1988), the speed of a compiled theorem prover is comparable to that of PROLOG.

The implementations of the five derivation strategies include the following:

1. A self configuration facility.
2. Inference operations used by each derivation strategy such as
   i) an extended extension operation which includes paramodulation (Wos and Robinson 1968)
   ii) reduction
   iii) truncation
   iv) spreading (for SLM and SLM-5 only)
3. The unit preference strategy
4. Pure literal elimination
5. The elimination of tautologies
6. Syntactic checking based on the restrictions imposed in each derivation strategy
7. A check if literals can be extended upon
8. A selection function to select the literal from the center chain to be resolved on during an extension operation.
9. A modified consecutively bounded depth-first search strategy.

Some of these features are not in the original definitions of the derivation strategies. In the implementations, the effects of adding extra features to a derivation strategy is to increase

the efficacy of the resultant system, without losing the basic structure of the original strategy.

3.2. Data Structures

Input clauses are represented by facts in the PROLOG database. Each fact has the following format :

```
a_clause(Clause)
```

where :

Clause = $[L_1, L_2, ..., L_n]$

L_i = a literal consisting of a sign and its atom. An atom preceded by the '++' (defined as an operator) sign denotes a positive literal and the '--' sign denotes a negative literal. (Literals are represented in this manner throughout the implementations.)

Input clauses are converted to input chains of B-literals before using them in derivations. The data structure of a B-literal in the ME procedure, SL-resolution and the GC procedure is in the following format :

$$\texttt{B\_Literal}_{ME/SL/GC} = \texttt{[b, Literal]}$$

In the ME procedure, an input clause should generate N matrix chains where N is the number of literals in the clause. To save memory space, only one chain is formed for every input clause. The extension operation of the ME procedure, however, is modified in such a way that any of the literals of the input clause can be selected. Thus, an input chain is formed by converting all the literals of the input clause to B-literal form in the ME and GC procedures. Input clauses to SL-resolution are factored and each factor is converted to an input chain. In all cases, a unique index is assigned to each input chain formed. The input chain and index are asserted into the PROLOG database as a two-argument fact :

input_chain(Index, Chain)

where :

Index = an integer greater than 0.

Chain = [$B\_Literal_1$, $B\_Literal_2$, ..., $B\_Literal_n$]

SLM and SLM-5 convert input clauses to input chains in a similar manner, but additionally assign a truth index to each literal in the clause. The data structure of a B-literal in SLM and SLM-5 is the following :

$$\mathbf{B\_Literal}_{SLM/SLM-5} = \mathbf{[b,\ Truth\_Index,\ Literal]}$$

where :

Truth_Index = an integer 0 or 1 which denotes FALSE or TRUE respectively.

Only the trivial interpretation $\mathbf{I_0}$ has been implemented to assign a truth index to each literal.

The data structure of a center chain in ME procedure, SL-resolution and GC procedure is a list. The term 'rightmost literal of the chain' in the formal definitions is actually the first literal of the center chain list in the implementations. A center chain in ME procedure and SL-resolution derivations is a list of B-literals and A-literals. The data structure of an A-literal in the ME procedure is a three-element list :

$$\mathbf{A\_Literal}_{ME} = \mathbf{[a,\ Scope,\ Literal]}$$

where :

Scope = an integer which represents the scope associated with the A-literal.

An A-literal in SL-resolution is represented by a two-element list :

$$\mathbf{A\_Literal}_{SL} = \mathbf{[a,\ Literal]}$$

A center chain in a GC procedure derivation is a list of B-literals, A-literals, C-literals and C-point atoms. The data structure of an A-literal in the GC procedure is a three-element list :

$$\mathbf{A\_Literal}_{GC} = \mathbf{[a,\ c\_N,\ Literal]}$$

where :

c_N = an atom that represents the C-point associated with the A-literal

N = an integer

The position of a C-point atom in the center chain is the C-point of the A-literal containing that C-point atom. A C-literal is represented by a two-element list :

$$\mathbf{C\_Literal}_{GC} = \mathbf{[c,\ Literal]}$$

A center chain in SLM and SLM-5 is in the following format :

$$\mathbf{Center\_Chain = [Node_1, \ldots, Node_n]}$$

where :

$Node_i = [L,R,Subchain_{SLM|SLM\text{-}5}]$

L = an integer which serves as a link to the previous node.

R = an integer which serves as a link to the next node.

$Subchain_{SLM}$ = a list of B-literals, A-literals and depth atoms.

$Subchain_{SLM\text{-}5}$ = a list of B-literals, A-literals, C-literals and depth atoms.

In SLM, A-literals have the following two formats :

(i) A-literals indexed by 0

$$\mathbf{A\_Literal}_{SLM} = \mathbf{[a,\ 0,\ Literal]}$$

(ii) A-literal indexed by 1

$$\mathbf{A\_Literal}_{SLM} = \mathbf{[a,\ 1,\ d\_N,\ Literal]}$$

where :

d_N = an atom that represents the depth associated with the A-literal

N = an integer

In SLM-5, A-literals have the following format :

(i) A-literals indexed by 0

$$\mathbf{A\_Literal}_{SLM-5} = \mathbf{[a,\ 0,\ Status,\ Literal]}$$

(ii) A-literal indexed by 1

$$\mathbf{A\_Literal}_{SLM-5} = \mathbf{[a,\ 1,\ d\_N,\ Literal]}$$

where :

Status = an integer 0 or 1 which represents the status flag associated with the A-literal.

d_N = an atom that represents the depth associated with the A-literal

N = an integer

The position of the depth atom in the branch is the depth of the A-literal. A C-literal of SLM-5 is represented in the following format :

$$\mathbf{C\_Literal}_{SLM-5} = \mathbf{[c,\ 0,\ Literal]}$$

A branch can be extracted from a center chain by getting first the root node (the root node is the node whose first two elements are 0,0) from the center chain and extracting the succeeding nodes from the rest of the center chain. The succeeding nodes

are obtained by repeatedly matching the second element of the current node to the first element of any of the remaining nodes in the center chain, until a tip node is obtained. A tip node is obtained if its second element cannot be matched with the first element of any of the remaining nodes in the center chain.

3.3. Self Configuration

Each of the theorem provers implemented configures itself to a certain extent. Before any derivations begin, an examination of the input clauses is done to determine the following :

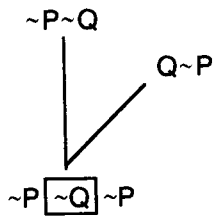
1. the occurence of equality literals.
2. the occurence of pure literals.
3. the occurence of tautologous clauses.
4. whether the set of clauses is a set of Horn clauses or a set of non-Horn clauses.
5. whether the set of clauses is written in a propositional or a first order predicate logic.
6. the minimum and maximum number of literals in a clause (the size of the clause).

If an equality literal exists (i.e. a literal with an `equal(L,R)` atom), the reflexive axiom of equality is added to the input chains by asserting the input chain whose single B-literal is `++equal(X,X)`. The predicate `equal_exist` is also asserted into the PROLOG database to indicate that an equality literal exists.

A check for pure literals is also done during the self configuration. If such a literal exists, the clause containing the pure literal is removed from the database because it

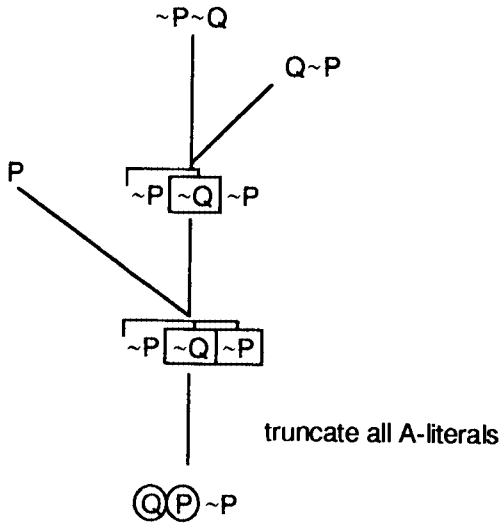
can only pollute the derivation search space. Tautologous input chains are also removed from the database.

If the input clauses are all Horn clauses, a PROLOG fact `clause_type(horn)` is asserted into the database; otherwise, the fact `clause_type(general)` is asserted. If the `clause_type(horn)` exists, the reduction operation is suppressed in the ME procedure, SLM and SLM-5. The basis for suppressing the reduction operation is the completeness of input resolution for Horn clauses (Henschen 1974). However, the effects of the restrictions imposed in the ME procedure, SL-resolution, the GC procedure, SLM and SLM-5 systems need to be considered. In the case of the ME procedure whose reduction operation is equivalent to ancestor resolution, the reduction operation is not necessary for a given set of Horn clauses. In SLM and SLM-5 using the trivial interpretation $\mathbf{I_0}$, the reduction operation is not also necessary for a set of Horn clauses. Although the reduction operation of SLM and SLM-5 is not purely ancestor resolution (because they reuse truncated A-literals indexed by 1), it is still safe to suppress the reduction operation because no A-literal indexed by 1 can occur in any center chains of the derivation. In SL-resolution whose reduction operation involves factoring and ancestor resolution, reduction cannot be suppressed. This is because SL-resolution imposes the admissibilty restriction which constrains the derivation not to produce center chains that contain two literals having identical atoms. Consider the derivation from the set of Horn clauses `S={~P~Q,Q~P,P}`.



The last center chain will be inadmissible if the reduction operation is suppressed. Thus, SL-resolution is incomplete if the reduction operation is suppressed. In the case of the

GC procedure, its reduction operation involves ancestor resolution and C-reduction which has similar effect to factoring. Again, reduction cannot be suppressed because of the C-literal mechanism and the imposed restriction. Consider the following GC procedure derivation of the given set of clauses :



The last center chain will become inadmissible if an extension is performed. Thus, the GC procedure is incomplete if the reduction operation is suppressed.

The minimum and maximum sizes of the input clauses are determined. This information is stored in an asserted fact `clause_size(Min, Max)`. This information is used to order the input clauses from the clause with the minimum number of literals to the clause with the maximum number of literals. This information is also used in the ME procedure for suppressing long lemmas generated in the derivations. Lemmas whose sizes are greater than the maximum size of the original input clauses are not added to the input set.

3.4. Extending the Extension Operation

The extension operation defined by each derivation strategy is extended to include a special case of binary resolution (referred to as subsumed unit extension by Sutcliffe (1989, p. 9)) and paramodulation. A subsumed unit extension is a binary resolution whose input parent chain is a unit chain and whose literal subsumes the negation of the selected B-literal of the center chain. After a subsumed unit extension no backtracking is permitted. The Prolog technology theorem prover (Stickel 1986) and the GCTP (Sutcliffe 1989) include the subsumed unit extension as an inference operation.

The equality axioms which establish that equality is reflexive, symmetric, transitive and allow equal terms to be substituted in any expression (substitution) are common in mathematical theories. The basic resolution principle, however, makes no special provision for the use of these axioms. To introduce equality into a resolution based theorem prover, it is necessary to include clauses that specify the equality axioms in the input set. However, the inclusion of the equality axioms to the set of clauses "is a source of many difficulties" as pointed out by Bundy (1983, p. 62). The alternative is to build into the theorem prover the knowledge required to handle equality appropriately. An inference rule called paramodulation is used for this purpose, whenever the `equal_exist` predicate exists in the datbase. It combines into a single step the operations of instantiation and replacement of (equal) terms (Wos 1984, p. 121). Moreover, Brown (1984, p. 38) asserted that - "adding paramodulation to a resolution theorem prover results in an ability to prove theorems about systems that contain equality in a natural, efficient way."

Paramodulation is added to the extension operation as it is viewed as a sequence of binary resolution steps put into one step. The equality axioms of symmetry, transitivity and substitution are not needed in the input clauses with the inclusion of the paramodulation in the extension operation. The only equality axiom necessary to obtain

completeness, is the reflexivity axiom. The algorithm for deciding which suboperation of extension is attempted first during an extension operation is described in the next section.

3.5. Search Strategy

Syntactic restrictions imposed in each derivation strategy reduce the size of the search space. These restrictions are invoked after every inference step which involves substitution of variables or the addition of new B-literals to the center chain. Two new syntactic restrictions have been added to the GC procedure in the implementation. The first one is not to allow a B-literal which is identical to any preceding non-B-literals (a modified version of the second preadmissibility restriction of the ME procedure). This restriction is a preemptive check of the no two non-B-literals with identical atoms restriction. The second additional restriction is not to allow complementary B-literals unless they are separated by an A-literal (first preadmissibility restriction of the ME procedure). This restriction prevents the use of a tautologous instance of an input chain. Tautologous input chains are not needed to prove the unsatisfiability of the input chains (Loveland 1969a). To illustrate the effect of these two restrictions on GC procedure derivations, consider the example shown in Figure 26. The ninth center chain of the right branch of the search tree is inadmissible if the first restriction is imposed. The restriction detects the redundancy one step earlier than the restriction imposed by the GC procedure. Imposing the second additional restriction on GC procedure derivations would detect the redundancy of the sixth center chain of the right branch of the search tree, three steps before the GC procedure detects the redundancy.

The set of support strategy is used in all implementations. The set of support strategy limits the number of search trees which need to be investigated in the course of searching for a refutation. Kowalski and Kuehner (1971 , p. 232) identified the set of all negative clauses or the set of all positive clauses as possible sets of support. In the

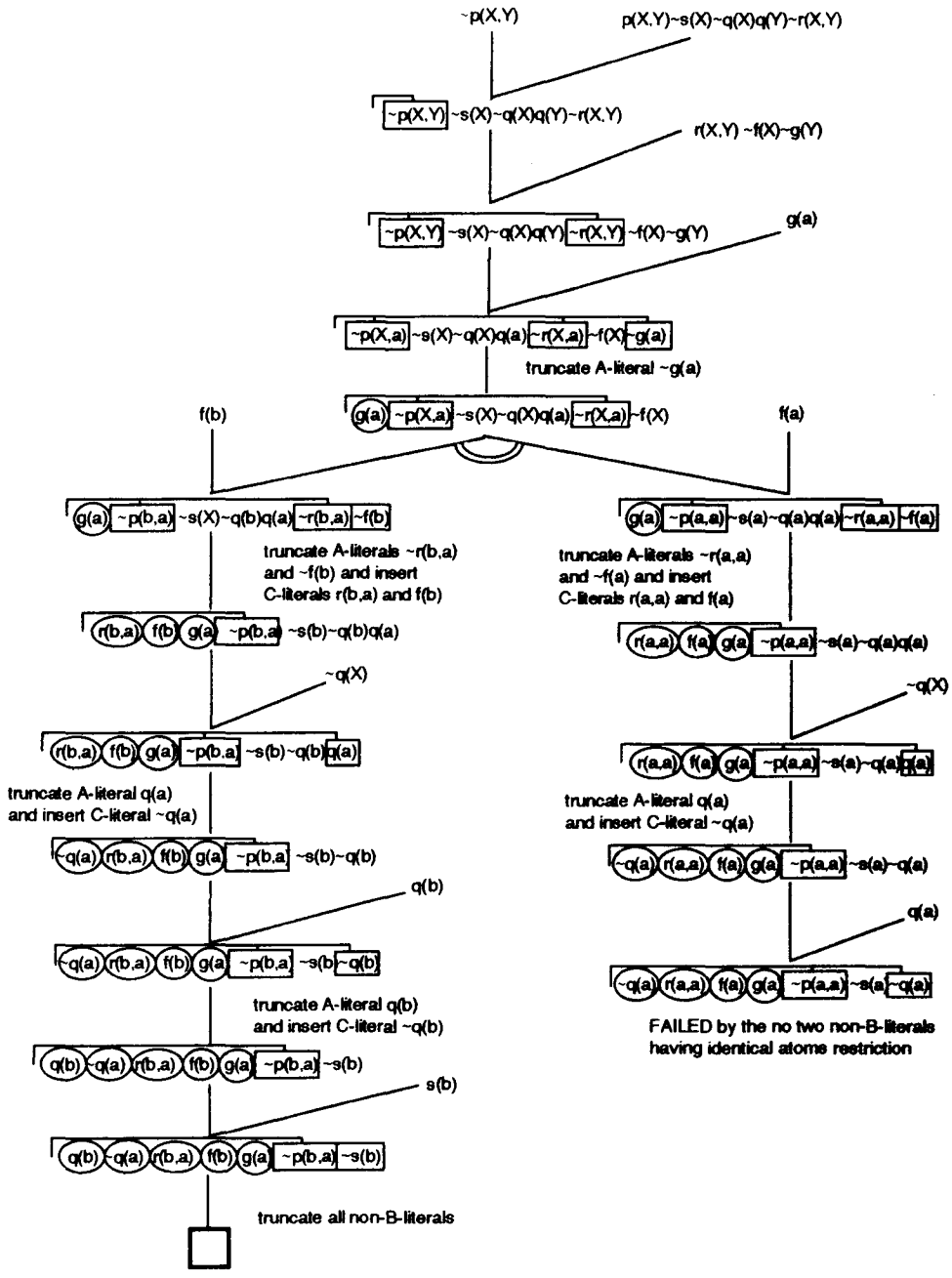


Figure 26. The search tree for S = {~p(X,Y), p(X,Y)~s(X)~q(X)q(Y)~r(X,Y), r(X,Y)~f(X)~g(Y), g(a), f(a), f(b), q(a), q(b), s(b), ~q(X)} with the top clause ~p(X,Y), using the GC procedure.

implementations, the set of negative clauses is the default support set. Sutcliffe (1989, p. 8) argued that using such a negative clause as the top clause leads to 'natural' proofs.

The search trees shown in Figures 27 and 28 demonstrate the effect of the choice of literal to resolve upon, on the size of the search tree. The size of the search tree in Figure 27 is larger than that of the search tree shown in Figure 28. This disparity is caused by the choice of which literal to resolve on first. Resolving on ~q(X) first will cause the substitution of the variable X immediately, while resolving ~p(X) first will delay the substitution of the variable X. Instantiation of variables at an early stage is advantageous because there are typically a lesser number of possible input chains that can be matched with a sufficiently instantiated literal, thereby reducing the number of resolvants. In view of this, a literal with a lesser number of matching clauses will be given higher priority in the selection of literals to be resolved on. Naish (1985, p. 61) suggested that the literal with the lowest number of matching clauses should be selected first to restrict early expansion of the search tree. Moreover, a literal which can be resolved on by an input chain which may introduce more B-literals to the center chain should have lower priority in the selection. Adding more B-literals to the center chains may produce longer refutations. Giving preference to literals which can be resolved on by clauses with fewer literals is in line with the unit preference strategy.

The selection function of the implementations considers all the points mentioned in the previous paragraph. To do this, the extension operation is performed on each candidate literal (but not allowing the binding of variables) and the number of new B-literals for each successful extension is accumulated. Preference is given to literals which accumulate the lowest total. To break ties, literals which can be resolved on by a unit chain and/or have lesser number of variables are preferred. Thus, the following formula is used to compute the weight of each candidate literal, and the literal with the minimum weight is the selected literal :

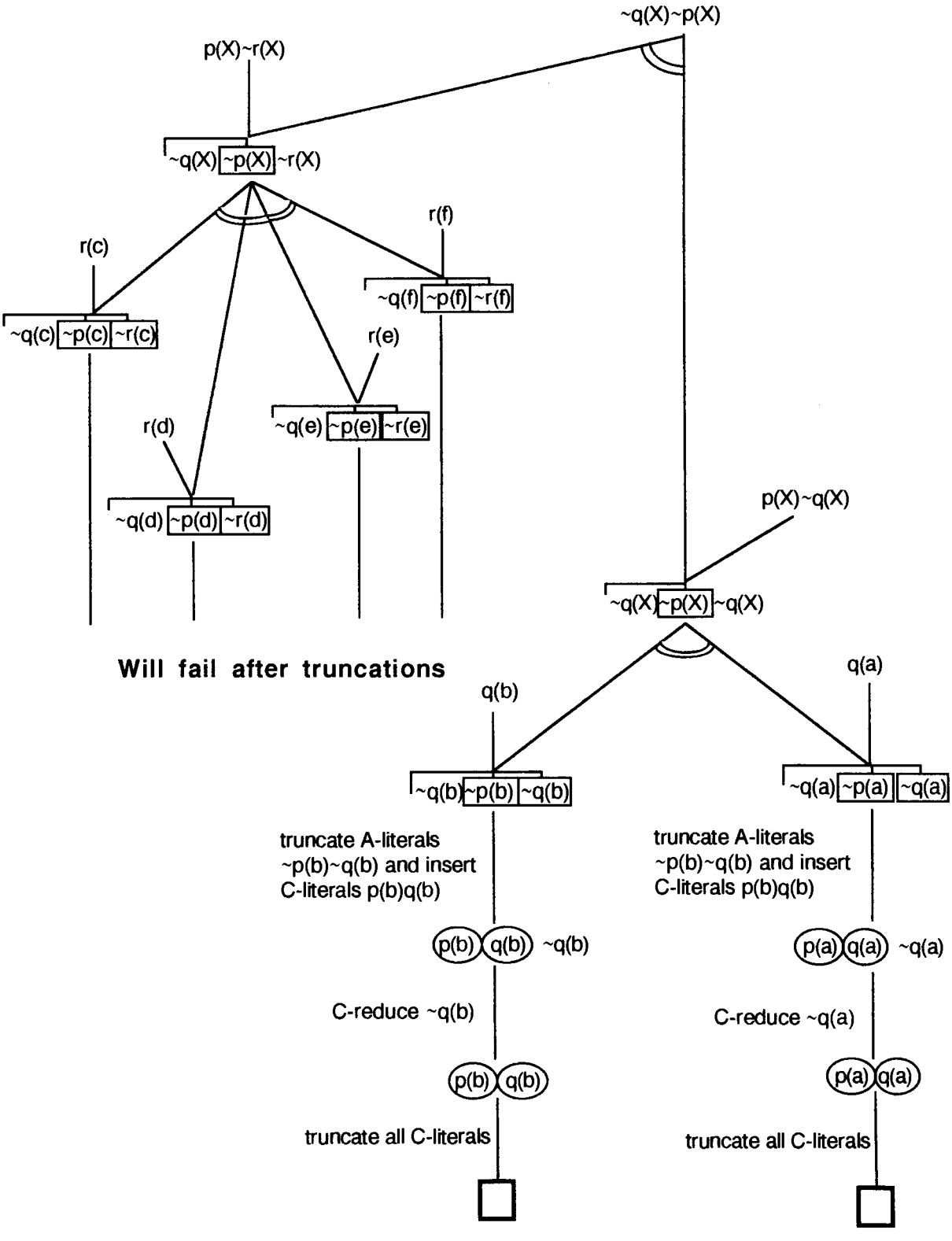


Figure 27. The search tree for S = {~q(X)~p(X), p(X)~q(X), p(X)~r(X), q(a), q(b), r(c), r(d), r(e), r(f) } with the top clause ~q(X)~p(X), using the GC procedure.

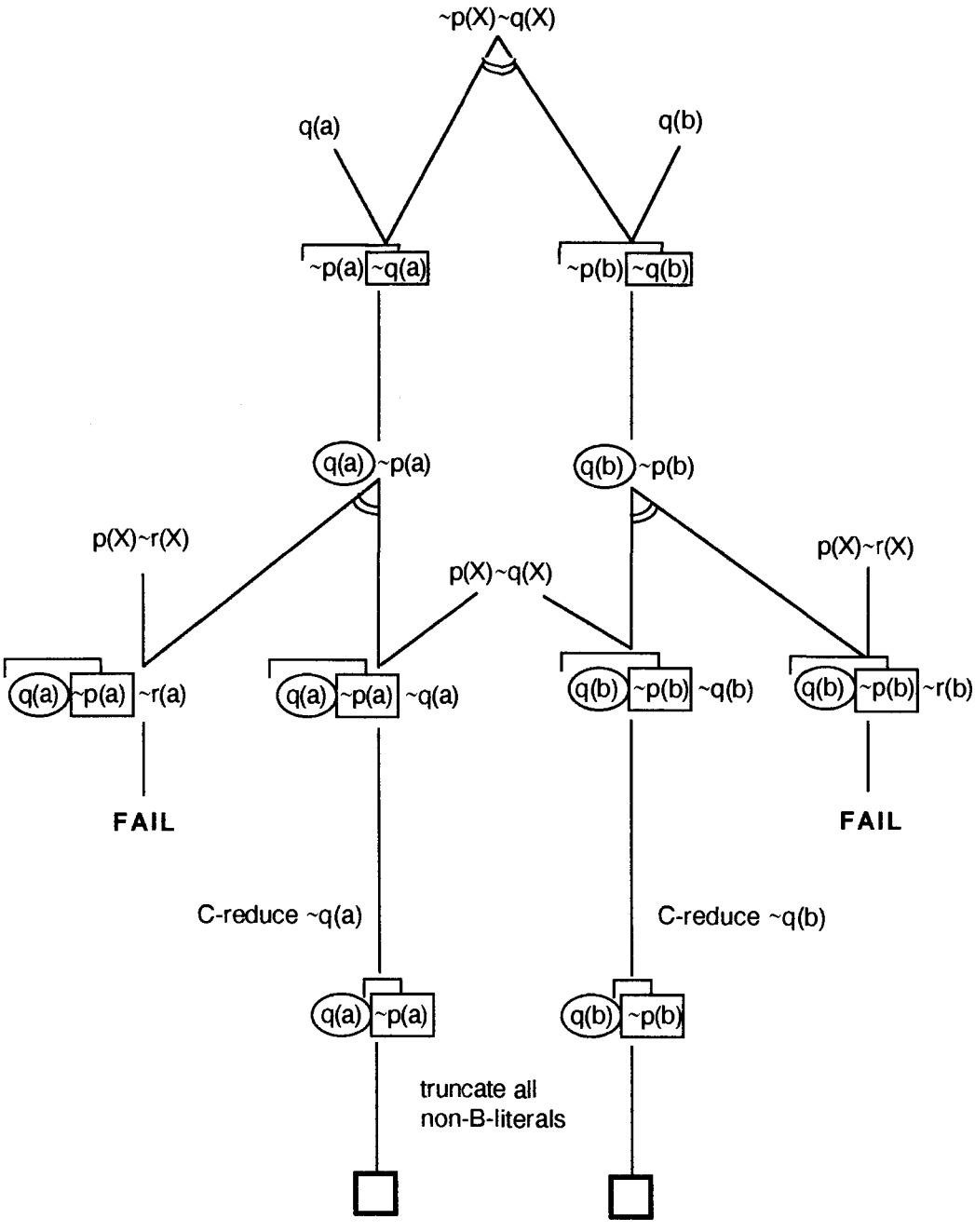


Figure 28. The search tree for S = {~p(X)~q(X), p(X)~q(X), p(X)~r(X), q(a), q(b), r(c), r(d), r(e), r(f) } with the top clause ~p(X)~q(X), using the GC procedure.

$$\text{Weight} = \left[\sum_{i=1}^{m} N_i\right] - \text{Unit} + \text{Var}$$

where :

N_i = the size of the input parent chain to resolve on the literal.

m = the number of possible input parent chains.

Unit = 0 if the center chain literal can be resolved on by a unit chain, otherwise 1.

Var = the number of variables in the center chain literal.

In the ME procedure, SL-resolution and the GC procedure, the implemented selection function extracts the rightmost cell of the center chain, computes the weight for each B-literal of the rightmost cell and selects the literal with the minimum weight. In the case of SLM and SLM-5, the implemented selection function first extracts all the tip nodes of the center chain. The B-literals of the rightmost cell of each tip node are then collected. B-literals which are identical to any A-literals are removed from the collected B-literals. This is done to prevent resolving away the same literal concurrently as much as possible. The remaining B-literals become the candidate literals. If all the collected B-literals have identical A-literals then all are used as candidate literals. The weight of each B-literal from the list of candidate literals is computed using the above formula. The B-literal with the minimum weight is the selected literal. The selection function returns the selected B-literal and the tip node where the literal belongs.

In the implementation of SLM-5, the part of truncation operation which adds proved literals to the input chain database is suppressed if the given set of clauses is a set of Horn clauses. This is because linear input resolution is already complete for set of Horn clauses, hence, adding more clauses to it may only expand the search tree. In the implementation of the spreading operation of SLM-5, the operation is suppressed if the given set of clauses is written at propositional level. The main objective of the spreading operation is to have more choices of B-literals to resolve on in order for the selection

function to select a B-literal which is most likely to fail or succeed. This is significantly effective on clauses with variables because the derivation may be able to detect irrelevant substitution of variables on previous unifications sooner. However, with ground clauses, the effect of the spreading operation is insignificant. It is rather more advantageous to concentrate the effort on one branch. In this scheme, proved literals are more likely obtained earlier in the derivation, and having such literals in the input set is favorable, particularly in conjunction with the subsumed unit extension.

During the self configuration, the input chains are ordered within the Prolog database from the input chain with the minimum number of literals to the input chain with the maximum number of literals. This is to make sure that extension using clauses with fewer literals is attempted first. This search strategy is in line with the unit preference strategy because unit input chains are tried first. The effect of this search strategy, however, is less effective in the ME procedure because the lemmas generated during the derivations are added before the original input chains, to make sure that they are used before the original input chains. The preference of lemmas over the original input clauses is based on the reason that the literals of the lemmas are typically more instantiated than the input clauses. Using a lemma as an input parent chain during an extension operation would most likely provide conditions which would require that more compulsory reductions be performed.

A check whether each B-literal of a center chain can be extended upon is done after every inference step where substitutions of variables occur. The purpose of this check is to detect redundant derivations immediately after the substitution of variables instead of waiting until such a literal, which cannot be extended upon, is selected for extension. Take, for instance, the extension of the center chain ~q(X)[~p(X)]~r(X) shown in Figure 27. The check will immediately reject any resolvants obtained from the said center chain immediately after the substitution of the variable X. This check saves the time taken in truncating first the succeeding A-literals before finding that the resolvant of the previous extension operation is redundant. This check may use much time especially if

there are still many B-literals in the center chain. In view of this, only B-literals which are sufficiently instantiated are subjected to the check. A literal is sufficiently instantiated if its number of non-variable terms is at least half its total number of terms. The algorithm for this check is described in the next section. As described in the algorithm, two types of information may be added to the database : `valid_literal(Atom)` and `redundant_literal(Atom)`, where Atom is the literal's atom. The `valid_literal(Atom)` fact is used to identify a B-literal, whose atom is subsumed by the Atom, that can be extended upon. The `redundant_literal(Atom)` fact is used to identify a B-literal , whose atom subsumed the Atom, that cannot be extended upon. These two facts could possibly be provided by the user (if they are known) together with the set of clauses to accelerate the derivations. However, this should be done with caution since they may affect the completeness of the system. It must be pointed out that the check is suppressed if the given set of clauses is written in propositional logic (i.e., literals are all ground). The reason of this is that all clauses that contain pure literals are removed during self configuration. Since the literals of the input clauses are all ground then there can be no instance that a pure literal may occur in a derivation.

Sutcliffe (1989) modified the standard consecutively bounded depth first search implemented in the Prolog technology theorem prover by Stickel (1985). The modified consecutively bounded depth first search used by Sutcliffe places a bound on the number of A- and B-literals in the current center chain. This version of the consecutively bounded depth first search is used in the implementations of the five derivation strategies. At the beginning of a search, the initial depth bound is either set to a user specified value or (if not specified) the size of the chosen top clause is used. The number of A- and B-literals in the center chain is always monitored after every extension operation. If a refutation is not obtained and the number of A- and B-literals in the center chain exceeds the depth bound, backtracking occurs. The minimum amount by which the depth bound is exceeded is monitored (asserted as `exceed(N)` in the Prolog database where N is the difference after subtracting the depth bound from the number of A- and B-literals in the center chain). If the bounded search fails to find a refutation and the fact `exceed(N)`

exists, the depth bound is incremented by N and the derivation is re-started. If a search fails and `exceed(N)` does not exist, the search is terminated in failure. If at any stage of the search the center chain is empty, the search is stopped and the refutation is completed.

The general algorithms of the five implemented theorem provers are described in Appendix A. The algorithms have been implemenetd using the ARITY Prolog compiler. The source codes of the programs are listed in Appendix B.

3.6. Theorem Prover Description

The implementations of the ME procedure, SL-resolution, the GC procedure, SLM and SLM-5 produce the ME-TP, SL-TP, GC-TP, SLM-TP, SLM5-TP theorem provers respectively. Figure 29 shows the general system diagram of each theorem prover. Each theorem prover involves five files. The executable file which serves as the inference engine, the application database which handles the storage of information generated during runtime, the program description file which contains the description of the theorem prover, the commands description file which contains all the syntax of commands used and their descriptions, the problem files where each file contains a theorem to prove, and the output device where the output of the theorem prover is sent. The output device can be the console, printer or a text file. The five theorem provers are all command-driven.

Appendix C describes how to run each theorem prover in DOS; the format of the theorem to be proven (a set of clauses); how to load the set of clauses into the theorem prover; how to start the derivation; and how to direct the output of the derivation. The three types of output produced during and after the derivation are also described.

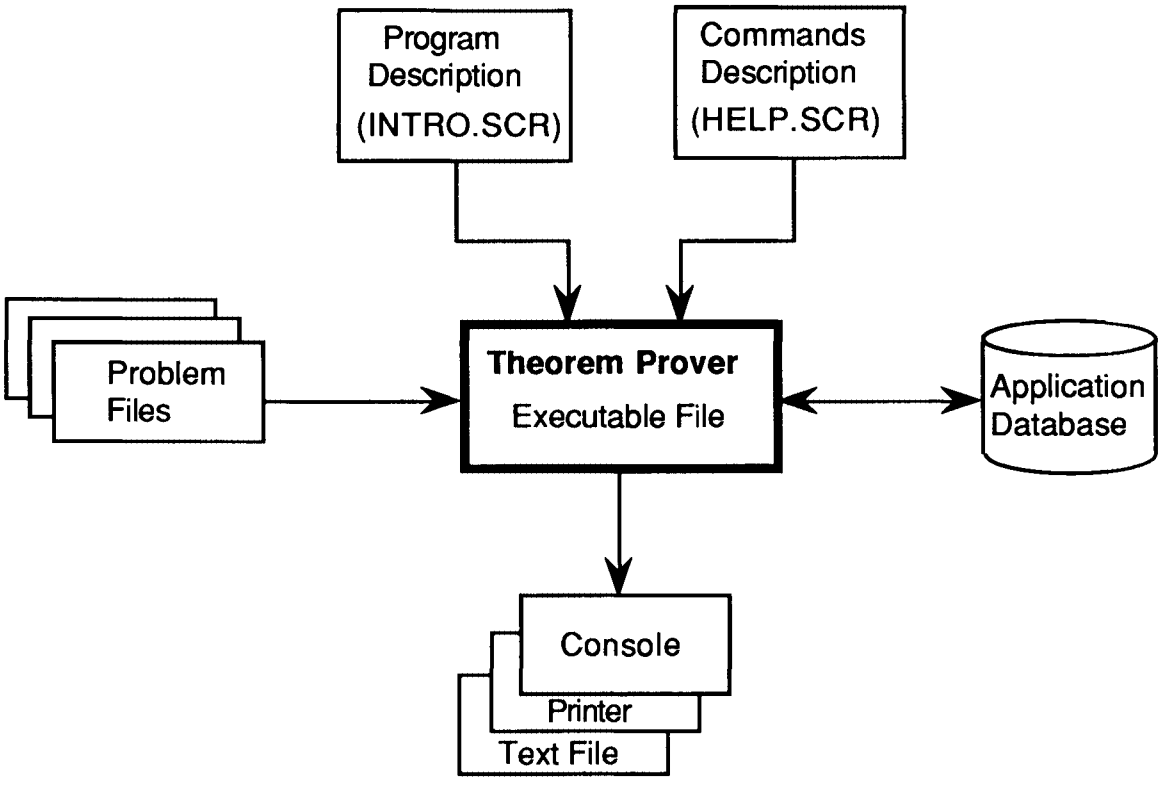


Figure 29. The theorem prover system diagram.

Chapter 4

COMPARISON

4.1. Introduction

The five theorem provers developed were tested with 13 problems taken from Pelletier (1986), 10 problems from Chang (1970), and the Schubert's Steamroller problem as presented in Stickel (1986). These 24 selected problems are listed in Appendix D. They were run on an IBM compatible machine which has a 16 MHz clockspeed with a 20 MB hard disk. The speed of the system was measured in terms of logic inference per second (LIPS) using the naive reverse program, which is commonly used as a benchmark test for PROLOG systems (Amble 1987, Tarnlund 1988). The naive reverse program is as follows :

```
reverse([], []).
reverse([E|List], Result) :-
    reverse(List, Partial),
    append(Partial, [E], Result).

append([], L, L).
append([H|List1, List2, [H|List3]) :-
    append(List1, List2, List3).
```

The naive reverse program was compiled using the Arity/Prolog compiler. It was run to reverse a list of 200 integers, giving the result of 1.38 seconds execution time, which is equivalent to 14,711 LIPS. The number of logical inferences (the number of PROLOG calls) is computed using the formula `LI = (N+1)*(N+2)/2` where N is the size of the list to reverse. During the testing, it was found that the LIPS value changes as the size of

the list is varied. The inconsistency of LIPS value can be attributed to the virtual memory use which is inherent to Arity/Prolog. For comparison purposes, the LIPS value of a system should be obtained by reversing a list of 200 integers. It must be pointed out, however, that "timing results are especially difficult to compare, influenced as they are by so many variables that are independent of the theorem-proving itself" (Stickel 1986, p. 93).

4.2. Comparison of Results

The results of proving the selected 24 problems are shown in Table 1. The proof search bound column of the table contains the search bound which produces the proof. The number of center chains in the derivation are the center chains generated by extension and reduction operations within the proof search bound (i.e., center chains generated before the proof search bound was reached are excluded in the count). The number of inadmissible center chains within the proof search bound were also recorded. The number of center chains in the proof represents the size of the refutation. In the case of the ME-TP and SLM5-TP, the number of center chains in the proof may not represent the actual size of the refutation. The proofs of lemmas or proved literals were not recorded. Hence, it is difficult to determine the actual size of a refutation which uses a lemma or a proved literal generated in previous search bounds, as input parent chain. The derivation duration is the time from the start of the search for the search bound in which the refutation was obtained, until the time when the empty center chain is obtained. The search duration is the total time duration for obtaining the proof.

The space efficiency of each theorem prover has been computed in terms of the number of center chains. This value is referred to as the memory use efficiency (MUE) and it uses the result of the ME-TP as the basis of comparison. The MUE is computed using the following formula :

$$MUE_{TP} = \frac{(RC_{TP} - RC_{ME-TP})}{RC_{ME-TP}} * 100$$

where :

RC = TC - IC

= the number of retained center chains

TC = the number of center chains generated in the derivation by extension and reduction operations

IC = the number of inadmissible center chains generated in the derivation by extension and reduction operations

TP = the SL-TP, GC-TP, SLM-TP or SLM5-TP

A positive MUE implies that the theorem prover uses more memory than the ME-TP and a negative MUE means that the theorem prover uses less memory than the ME-TP.

The execution time efficiency (ETE) of each theorem prover has also been computed. It is used to determine how fast the theorem prover solves each problem compared to the ME-TP. The efficiency was computed using the following formula :

$$ETE_{TP} = \frac{(SD_{TP} - SD_{ME-TP})}{SD_{ME-TP}} * 100$$

where :

SD = the search duration

A positive ETE implies that the theorem prover is slower than the ME-TP while a negative value means that the theorem prover is faster than the ME-TP.

Table 2 - Experimental Results of the Five Theorem Provers with the Size of the Top Clause as the Initial Search Bound.

Theorem Prover	Proof Search bound	Center chains in the derivation		Inadmissible center chains		Center chains in the proof		Duration (sec)		Efficiency (%)	
		Ext'n.	Red'n.	Ext'n.	Red'n	Ext'n.	Red'n.	Derivation	Search	MUE[a]	ETE[b]
				1. Selected problems taken from Pelletier (1986)							
				Pel-10 with (5) as the top clause							
ME-TP	5	8	3	2	0	6	3	0.55	1.65		
SL-TP	4	7	3	2	0	5	3	0.22	0.77	-11.11	-53.33
GC-TP	5	7	3	2	0	5	3	0.38	1.59	-11.11	-3.64
SLM-TP	5	9	3	3	0	6	3	0.66	2.47	0.00	49.70
SLM5-TP	5	8	2	2	0	6	2	0.55	2.03	-11.11	23.03
				Pel-12 with (8) as the top clause							
ME-TP	8	26	13	11	0	15	13	2.37	5.65		
SL-TP	5	16	12	5	0	7	10	1.21	1.81	-17.86	-67.96
GC-TP	8	17	10	10	0	7	10	1.10	4.72	-39.29	-16.46
SLM-TP	10	172	73	67	0	14	17	43.12	138.53	535.71	2351.86
SLM5-TP	8	25	12	10	0	15	12	2.19	7.86	-3.57	39.12
				Pel-14 with (3) as the top clause							
ME-TP	4	5	1	1	0	4	1	0.27	0.99		
SL-TP	3	4	2	1	0	3	2	0.17	0.60	0.00	-39.39
GC-TP	4	4	2	1	0	3	2	0.22	0.94	0.00	-5.05
SLM-TP	4	6	2	2	0	4	2	0.44	1.32	20.00	33.33
SLM5-TP	4	5	1	1	0	4	1	0.33	1.16	0.00	17.17
				Pel-17 with (5) as the top clause							
ME-TP	5	5	1	0	0	5	1	0.39	1.21		
SL-TP	5	5	1	0	0	5	1	0.17	0.66	0.00	-45.45
GC-TP	5	4	2	0	0	4	2	0.33	1.21	0.00	0.00
SLM-TP	5	5	1	0	0	5	1	0.38	1.43	0.00	18.18
SLM5-TP	5	5	1	0	0	5	1	0.49	1.53	0.00	26.45

cont'd. over

Table 2 (cont'd.)

Theorem Prover	Proof Search bound	Center chains in the derivation		Inadmissible center chains		Center chains in the proof		Duration (sec)		Efficiency (%)	
		Ext'n.	Red'n.	Ext'n.	Red'n	Ext'n.	Red'n.	Derivation	Search	MUE[a]	ETE[b]
					Pel-20 with (5) as the top clause						
ME-TP	3	3	0	0	0	3	0	0.22	0.66		
SL-TP	3	3	0	0	0	3	0	0.22	0.55	0.00	-16.67
GC-TP	3	3	0	0	0	3	0	0.22	0.66	0.00	0.00
SLM-TP	3	3	0	0	0	3	0	0.27	0.77	0.00	16.67
SLM5-TP	3	3	0	0	0	3	0	0.27	0.77	0.00	16.67
					Pel-21 with (4) as the top clause						
ME-TP	4	4	1	0	0	4	1	0.33	1.04		
SL-TP	4	4	2	0	0	4	2	0.28	1.16	20.00	11.54
GC-TP	4	3	2	0	0	3	2	0.22	0.99	0.00	-4.81
SLM-TP	4	5	2	0	0	4	2	0.49	1.54	40.00	48.08
SLM5-TP	4	4	1	0	0	4	1	0.49	1.54	0.00	48.08
					Pel-23 with (4) as the top clause						
ME-TP	5	7	3	1	0	4	3	0.55	2.42		
SL-TP	4	3	1	0	0	3	1	0.16	1.76	-55.56	-27.27
GC-TP	5	6	3	1	0	3	3	0.55	2.47	-11.11	2.07
SLM-TP	6	21	7	13	0	4	2	1.59	5.99	66.67	147.52
SLM5-TP	5	9	4	2	0	5	4	1.04	3.46	22.22	42.98
					Pel-24 with (6) as the top clause						
ME-TP	5	7	2	0	0	7	2	0.77	1.98		
SL-TP	6	9	3	0	0	9	3	0.61	3.02	33.33	52.53
GC-TP	5	6	3	0	0	6	3	0.61	1.87	0.00	-5.56
SLM-TP	7	7	3	0	0	7	3	1.16	3.74	11.11	88.89
SLM5-TP	7	7	2	0	0	7	2	1.10	3.84	0.00	93.94
					Pel-25 with (7) as the top clause						
ME-TP	5	30	2	3	0	11	0	2.58	5.33		
SL-TP	6	8	3	0	0	8	3	0.66	5.33	-62.07	0.00
GC-TP	5	24	8	3	0	7	4	2.69	5.50	0.00	3.19
SLM-TP	6	8	3	0	0	8	3	1.20	8.46	-62.07	58.72
SLM5-TP	5	30	2	3	0	11	0	4.18	7.91	0.00	48.41

cont'd. over

Table 2 (cont'd.)

Theorem Prover	Proof Search bound	Center chains in the derivation		Inadmissible center chains		Center chains in the proof		Duration (sec)		Efficiency (%)	
		Ext'n.	Red'n.	Ext'n.	Red'n	Ext'n.	Red'n.	Derivation	Search	MUE[a]	ETE[b]
				Pel-27 with (2) as the top clause							
ME-TP	5	7	0	0	0	7	0	0.72	1.81		
SL-TP	5	8	0	0	0	8	0	0.54	1.65	14.29	-8.84
GC-TP	5	7	1	0	0	7	1	0.83	2.09	14.29	15.47
SLM-TP	5	8	0	0	0	8	0	0.88	2.25	14.29	24.31
SLM5-TP	5	8	0	0	0	8	0	0.87	2.36	14.29	30.39
				Pel-30 with (7) as the top clause							
ME-TP	4	4	1	0	0	3	1	0.38	1.60		
SL-TP	4	4	1	0	0	3	1	0.27	1.42	0.00	-11.25
GC-TP	4	4	1	0	0	3	1	0.33	1.53	0.00	-4.38
SLM-TP	4	4	1	0	0	3	1	0.38	1.86	0.00	16.25
SLM5-TP	4	4	1	0	0	3	1	0.44	1.98	0.00	23.75
				Pel-31 with (6) as the top clause							
ME-TP	3	4	0	0	0	4	0	0.33	0.88		
SL-TP	3	4	0	0	0	4	0	0.22	0.76	0.00	-13.64
GC-TP	3	4	0	0	0	4	0	0.32	0.88	0.00	0.00
SLM-TP	3	4	0	0	0	4	0	0.38	1.10	0.00	25.00
SLM5-TP	3	4	0	0	0	4	0	0.44	1.16	0.00	31.82
				Pel-32 with (7) as the top clause							
ME-TP	4	6	0	0	0	6	0	0.44	1.15		
SL-TP	4	7	0	0	0	7	0	0.39	1.10	16.67	-4.35
GC-TP	4	5	1	0	0	5	1	0.50	1.21	0.00	5.22
SLM-TP	4	7	0	0	0	7	0	0.66	1.49	16.67	29.57
SLM5-TP	4	7	0	0	0	7	0	0.66	1.54	16.67	33.91
				2. Problems taken from Chang (1970).							
				Chang-1 with (4) as the top clause							
ME-TP	4	5	0	0	0	4	0	0.87	1.26		
SL-TP	3	4	0	0	0	3	0	0.60	1.05	-20.00	-16.67
GC-TP	4	3	1	0	0	3	1	0.39	0.82	-20.00	-34.92
SLM-TP	4	5	0	0	0	4	0	0.99	1.49	0.00	18.25
SLM5-TP	4	5	0	0	0	4	0	1.04	1.48	0.00	17.46

cont'd. over

Table 2 (cont'd.)

Theorem Prover	Proof Search bound	Center chains in the derivation		Inadmissible center chains		Center chains in the proof		Duration (sec)		Efficiency (%)	
		Ext'n.	Red'n.	Ext'n.	Red'n	Ext'n.	Red'n.	Derivation	Search	MUE[a]	ETE[b]
				Chang-2 with (7) as the top clause							
ME-TP	8	1344	0	587	0	10	0	225.96	475.82		
SL-TP	–	–	–	–	–	–	–	–	–	–	–[c]
GC-TP	7	743	662	444	252	11	5	262.38	281.16	-6.34	-40.91
SLM-TP	8	76	0	16	0	10	0	25.87	57.84	-92.07	-87.84
SLM5-TP	8	76	0	16	0	10	0	26.36	58.66	-92.07	-87.67
				Chang-3 with (5) as the top clause							
ME-TP	7	15	0	0	0	7	0	2.52	4.94		
SL-TP	5	43	2	12	2	6	0	8.30	13.51	106.67	173.48
GC-TP	7	42	52	12	18	6	1	18.24	20.65	326.67	318.02
SLM-TP	8	75	0	6	0	10	0	21.86	48.78	360.00	887.45
SLM5-TP	8	75	0	6	0	10	0	22.63	50.21	360.00	916.40
				Chang-4 with (5) as the top clause							
ME-TP	7	181	0	61	0	7	0	24.55	27.57		
SL-TP	–	–	–	–	–	–	–	–	–	–	–[c]
GC-TP	7	42	52	12	18	6	1	18.24	20.93	-46.67	-24.08
SLM-TP	7	15	0	1	0	7	0	3.79	6.59	-88.33	-76.10
SLM5-TP	7	15	0	1	0	7	0	3.84	6.65	-88.33	-75.88
				Chang-5 with (9) as the top clause							
ME-TP	4	4	0	0	0	4	0	0.71	1.10		
SL-TP	3	3	0	0	0	3	0	0.27	0.72	-25.00	-34.55
GC-TP	4	3	1	0	0	3	1	0.54	0.93	0.00	-15.45
SLM-TP	4	4	0	0	0	4	0	0.77	1.21	0.00	10.00
SLM5-TP	4	4	0	0	0	4	0	0.72	1.21	0.00	10.00
				Chang-6 with (9) as the top clause							
ME-TP	7	4	0	0	0	4	0	0.61	16.75		
SL-TP	6	1254	33	255	20	5	1	444.02	481.32	25200.00	2773.55
GC-TP	7	2827	2985	1417	578	5	2	2068.71	2090.96	95325.00	12383.34
SLM-TP	7	350	0	38	0	7	0	106.22	123.08	7700.00	634.81
SLM5-TP	7	350	0	38	0	7	0	109.91	126.94	7700.00	657.85

cont'd. over

Table 2 (cont'd.)

Theorem Prover	Proof Search bound	Center chains in the derivation		Inadmissible center chains		Center chains in the proof		Duration (sec)		Efficiency (%)	
		Ext'n.	Red'n.	Ext'n.	Red'n	Ext'n.	Red'n.	Derivation	Search	MUE[a]	ETE[b]
Chang-7 with (7) as the top clause											
ME-TP	5	8	1	1	0	5	1	0.93	2.31		
SL-TP	4	8	0	1	0	5	0	0.77	1.87	-12.50	-19.05
GC-TP	5	8	1	1	0	5	1	0.94	2.36	0.00	2.16
SLM-TP	5	8	1	1	0	5	1	1.04	2.69	0.00	16.45
SLM5-TP	5	8	1	1	0	5	1	1.15	2.97	0.00	28.57
Chang-8 with (9) as the top clause											
ME-TP	5	17	0	4	0	12	0	2.47	3.73		
SL-TP	7	37	2	10	0	18	0	4.23	16.25	123.08	335.66
GC-TP	5	13	4	4	0	8	4	2.36	3.51	0.00	-5.90
SLM-TP	7	19	6	5	0	10	4	3.46	26.81	53.85	618.77
SLM5-TP	6	21	2	5	0	12	0	4.28	6.42	38.46	72.12
Chang-9 with (8) as the top clause											
ME-TP	6	16	2	3	0	8	2	1.92	6.70		
SL-TP	6	16	2	3	0	8	2	1.48	6.10	0.00	-8.96
GC-TP	6	16	2	3	0	8	2	2.26	8.18	0.00	22.09
SLM-TP	7	16	2	3	0	8	2	2.25	9.84	0.00	46.87
SLM5-TP	7	16	2	3	0	8	2	2.69	10.99	0.00	64.03
Chang-10 with (12) as the top clause											
ME-TP	5	48	0	9	0	7	0	21.31	36.64		
SL-TP	5	69	0	8	0	7	0	29.61	52.46	56.41	43.18
GC-TP	5	48	4	9	0	7	0	26.80	43.61	10.26	19.02
SLM-TP	5	46	0	11	0	7	0	19.67	35.87	-10.26	-2.10
SLM5-TP	5	46	0	11	0	7	0	19.94	36.14	-10.26	-1.36
Chang-10 with (13) as the top clause											
ME-TP	5	24	0	6	0	7	0	11.65	31.75		
SL-TP	5	21	0	3	0	7	0	10.54	28.90	0.00	-8.98
GC-TP	5	24	2	6	0	7	0	13.68	34.93	11.11	10.02
SLM-TP	5	24	0	6	0	7	0	10.32	28.51	0.00	-10.20
SLM5-TP	5	24	0	6	0	7	0	10.44	28.72	0.00	-9.54

cont'd. over

Table 2 (cont'd.)

Theorem Prover	Proof Search bound	Center chains in the derivation		Inadmissible center chains		Center chains in the proof		Duration (sec)		Efficiency (%)	
		Ext'n.	Red'n.	Ext'n.	Red'n	Ext'n.	Red'n.	Derivation	Search	MUE[a]	ETE[b]
					Chang-10 with (14) as the top clause						
ME-TP	5	48	0	9	0	7	0	31.52	56.30		
SL-TP	5	93	0	10	0	7	0	61.03	82.33	112.82	46.23
GC-TP	5	48	4	9	0	7	0	29.66	49.32	10.26	-12.40
SLM-TP	5	46	0	11	0	7	0	22.57	42.35	-10.26	-24.78
SLM5-TP	5	46	0	11	0	7	0	22.80	42.67	-10.26	-24.21
					Chang-10 with (15) as the top clause						
ME-TP	4	16	0	3	0	5	0	10.49	17.52		
SL-TP	4	161	1	19	1	9	0	121.44	143.52	992.31	719.18
GC-TP	4	16	0	3	0	5	0	7.53	12.91	0.00	-26.31
SLM-TP	4	17	0	4	0	5	0	7.36	12.63	0.00	-27.91
SLM5-TP	4	17	0	4	0	5	0	7.36	12.74	0.00	-27.28
					3. Schubert's steamroller problem with (26) as the top clause						
ME-TP	11	35741	6282	18372	2470	39	1	15657.93	21150.93		
SL-TP	11	442	3	253	1	51	0	133.80	886.61	-99.10	-95.81
GC-TP	11	122	72	70	26	21	26	77.66	859.15	-99.54	-95.94
SLM-TP	–	–	–	–	–	–	–	–	–	–	–[d]
SLM5-TP	11	1047	19	501	19	31	0	640.00	884.00	-97.42	-95.82

Note : [a] MUE is the memory use efficiency of each theorem prover relative to the ME-TP result.

[b] ETE is the execution time efficiency of each theorem prover relative to the ME-TP result.

[c] Run out of disk space (10 MB) after running more than 24 hours.

[d] The derivation was aborted due to a 'not enough global stack' error.

4.2.1. Memory Use Efficiency Comparison

As shown in Table 1, none of the five theorem provers consistently used the least amount of memory in solving the selected problems. This shows that the space efficiency of a theorem prover is dependent on the properties of the problems.

The ME-TP required less memory than the others for problems PEL-27, CHANG-3, CHANG-6 and CHANG-8. The ME-TP obtained a smaller search space in these problems because unit lemmas, which were generated in previous search bounds, were used as input chains during extension operations. These results gave evidence that adding lemmas, especially unit lemmas, is effective in conjunction with subsumed unit extension and the unit preference strategy. It also has a positive effect in a consecutively bounded depth-first search environment because lemmas generated in one search bound are carried over to the next search bound The other theorem provers, especially the SL-TP, used a great deal of memory in finding a proof for the CHANG-3 problem. This problem of SL-TP is mainly caused by the excessive application of the factoring operation. The GC-TP had difficulty in solving the CHANG-6 problem. As shown in the results, the GC-TP performed too many C-reductions (no A-reduction was performed because CHANG-6 is a set of Horn clauses). However, lemma generation does not always have positive effects. The negative effect of lemma generation in the ME-TP is shown in the results for the Schubert's steamroller problem. The results show that the ME-TP used more memory than SL-TP, GC-TP and SLM5-TP. This was because the ME-TP generated many lemmas of which some were non-unit lemmas. The addition of these lemmas increases the number of possible input parent chains, which caused expansion of the search tree. The negative effect of generating non-unit lemmas is also obvious in the results for PEL-12.

The SL-TP required less memory than the other theorem provers in solving problems CHANG-5 and CHANG-7. This efficiency of SL-TP compared to the others was due mainly to the initial factoring of the input clauses prior to the derivation.

Examining problem CHANG-5 (in Appendix A), the SL-TP converts clause (6) into two chains, one chain for the original clause and another chain for the factor of the clause. The application of factoring to each input clause prior to the derivation is particularly effective in conjunction with the unit preference strategy because input chains with fewer literals are tried first in extension operations. However, SL-TP used more memory than the other theorem provers in solving problems CHANG-1, CHANG-3, and CHANG-10(12) (XXXX(N) means problem XXXX with clause (N) as the top clause). The expanded search trees of SL-TP in these three problems are caused by the indiscriminate application of factoring. The search trees obtained by SL-TP in solving problems CHANG-2 and CHANG-4 were too large for the computer to handle. The expanded search trees in these problems are caused by the factoring of input clauses during the conversion of input clauses to input chains, and the factoring operations applied during the derivation. This confirms the argument of Brown (1974, p. 4) that factoring may add an irrelevancy to the search space. It is notable that SL-TP obtains good results in problems written at the propositional level, such as PEL-10, PEL-12, PEL-14 and PEL-17. This is because factoring operations performed in these problems are all compulsory. However, in first order predicate calculus problems, especially those recursive type problems such as CHANG-2, CHANG-3, CHANG-4 and CHANG-10, the factoring operation does cause some problems.

The GC-TP required less memory than the other theorem provers in solving problems PEL-12, CHANG-1 and the Schubert's steamroller. As shown in the results of solving the Schubert's steamroller problem, the GC-TP performed a lot better than the ME-TP. This efficiency of the GC-TP compared to the ME-TP is due to the effects of the C-literal mechanism. The C-literal mechanism has two positive effects. The first effect is that it may help narrow down the search tree because the admissibility restrictions do not allow any A- or B-literals which are or become identical to any of the inserted C-literals. In the ME-TP derivation, the preadmissibility restriction does not consider the recycled literals (in the form of lemma literals) in assessing the admissibility of a center chain. The second advantage of the C-literal mechanism over the lemma generation of the ME-TP is

that it may simplify the refutation because one C-reduction of the GC-TP is equivalent to one ME-TP extension using a lemma, removal of other B-literals from the lemma by reduction, and a truncation. A shorter refutation is advantageous because less memory is used. However, in a consecutively bounded depth-first search environment, lemmas generated by the ME-TP have more lasting effects than the C-literals of the GC-TP, because C-literals produced within one search bound are not carried over to the next search bound.

The C-reduction of the GC-TP can be viewed as delayed factoring in the sense that the leftmost of two unifiable B-literals in a center chain may be removed by reduction after the other is resolved away and inserted as a C-literal at a position preceding the first B-literal. The results of solving PEL-12 show that C-reduction has a better effect than the factoring operation. The GC-TP uses less extension operations than the SL-TP because of the use of recycled literals. However, in some cases the GC-TP produces larger search trees as demonstrated in the results of solving CHANG-6, CHANG-10(13) and CHANG-10(14). This negative effect of the GC-TP can be attributed to the insertion of C-literals which give more choices in the selection of non-B-literals during reduction operations. As shown in the results for CHANG-6, the GC-TP performed more reduction operations than SL-TP. Although the C-literal mechanism is similar in effect to the lemma generation of the ME-TP, the ME-TP performed better than the GC-TP in these problems. The reason is that the C-literal mechanism of the GC-TP always converts a truncated A-literal to a C-literal, even if its atom is subsumed by the atom of a unit chain. This causes problems because a B-literal can either be reduced or resolved away by extension, thereby expanding the search tree. In the case of the ME-TP, a formed lemma is only added if it is not subsumed by any input chains.

SLM-TP and SLM5-TP produce smaller search trees in solving CHANG-2, CHANG-4, CHANG-10(12) and CHANG-10(14). This is due to the spreading operation which provides more flexibility in the selection of literal to be resolved on. Having more choices of literals to resolve on will give more chances for the selection

function to select a literal which leads to the detection of failure or to a successful refutation. The spreading operation, however, is less effective in problems written at propositional level. The results in problems PEL-12 and PEL-14 demonstrate this effect. This is because the selection function alternately selects B-literals from different branches of the center chain, which results in less opportunities to apply reduction. Hence, the spreading operation is suppressed in the implementation of SLM-5 if the set of clauses is at propositional level. As demonstrated in the results, SLM5-TP uses less memory than the SLM-TP in problems written at propositional level. Generally, the overall results showed that SLM5-TP has better memory use efficiency than SLM-TP.

For sets of Horn clauses written at first order level, the results for SLM5-TP are the same as for SLM-TP. However, the results of the two theorem provers differ for sets of non-Horn clauses. The results of solving the Schubert's steamroller problem show that SLM5-TP performed a lot better than SLM-TP and ME-TP. SLM5-TP performed better than SLM-TP because SLM5-TP added proved literals to the set of input chains which is favorable in conjunction with subsumed unit extension and the unit preference strategy. The addition of proved literals also has positive effects in a consecutively bounded depth-first search environment. SLM5-TP also performed better than the ME-TP in solving the Schubert's steamroller problem because of the spreading operation and because all the added chains from proved literals areunit chains.

4.2.2. Execution Time Efficiency Comparison

In most cases, when a theorem prover produces the least number of center chains the execution time is also the least. Obviously, lesser efforts are required when fewer center chains are produced, thereby requiring less execution time.

There are some cases, however, when a theorem prover has a low execution time although it has the same or slightly larger number of center chains produced than the

others. Take, for instance, the results in problems PEL-12, PEL-27, PEL-31, PEL-32 and CHANG-9. Although the memory use efficiency of SL-TP in these problems is similar to those of the other theorem provers, its execution time is less than that of the others. The reason for this difference is the simplicity of the operations used in SL-TP. The truncation operation of SL-TP simply removes A-literals. The ME-TP truncation operation involves lemma formation, updating of scopes and a subsumption test to check if the lemma has to be added to the input chains. These require great effort which slows down the derivation. In the case of GC-TP, the updating of the C-point of A-literals affected by each reduction operation is an additional effort which also slows down the derivation. The complexity of the data structure of SLM-TP and SLM5-TP, in addition to the extra effort required to update depths of A-literals affected by reduction operations, increases the execution time.

4.2.3 Overall Performance Comparison

The over-all performance of the five theorem provers was measured in terms of the speed (execution time) difference between each theorem prover in solving non-trivial problems. Non-trivial problems are identified as those problems which require more than 60 seconds to solve by at least one theorem prover. Table 3 presents the speed differences between each of the theorem provers in non-trivial problems. The speed difference, SPD, is computed using the following formula :

$$SPD_{TP} = \frac{SD_{max} - SD_{TP}}{SD_{max} - SD_{min}} * 100$$

where :

SPD_{TP} = the speed difference of a theorem prover

SD_{max} = the maximum search duration to prove the problem

SD_{min} = the minimum search duration to prove the problem

SD_{TP} = the search duration of a theorem prover

Table 3. - Speed Difference of the five Theorem Provers in Problems which require more than 60 seconds to obtain a Refutation.

Problem	Speed Difference (%)					Maximum Search Duration	Level of Difficulty
	ME-TP	SL-TP	GC-TP	SLM-TP	SLM5-TP		
PEL-12	97.20	100.00	97.87	0.00	95.57	138.53	2
CHANG-2	0.00	_	46.57	100.00	99.80	475.82[a]	6
CHANG-4	0.00	_	31.65	100.00	99.71	27.57[a]	5
CHANG-6	100.00	77.89	0.00	94.87	94.69	2090.96	4
CHANG-10(14)	65.11	0.00	82.57	100.00	99.20	82.33	1
CHANG-10(15)	96.26	0.00	99.79	100.00	99.92	143.52	3
SCHUBERT	0.00	99.86	100.00	_	99.88	21150.93[b]	7
Mean[c]	33.87	_	61.26	_	98.76		
Mean[d]	55.78	71.21	75.16	_	98.12		
Mean[e]	45.16	_	48.35	89.50	98.39		
Mean[f]	94.83	51.16	57.77	77.95	96.87		

Note: a : SL-TP was not able to obtain result on this problem. The shown maximum search duration is taken from the result of the ME-TP.

b : SLM-TP was not able to obtain result on this problem. The shown maximum search duration is taken from the result of the ME-TP.

c : Disregarding the results of SL-TP and SLM-TP.

d : Disregarding the results in CHANG-2 and CHANG-4 problems.

e : Disregarding the results in the SCHUBERT problem.

f : Disregarding the results in CHANG-2, CHANG-4 and SCHUBERT problems.

The theorem prover that has bigger SPD value indicates that it performs better than the others in terms of execution time. The maximum search duration of a problem is the maximum execution time required to solve the problem by one of the theorem provers. Each non-trivial problem has a designated level of difficulty. The designation of level of difficulty to each problem is based on the maximum search duration. Problems in which one of the theorem provers did not produce results have a higher level of difficulty than the others. That is why CHANG-4, in which SL-TP did not produce a result, has a higher level of difficulty than CHANG-6 although CHANG-6 has a greater maximum search duration than CHANG-4.

A weighted mean for each theorem prover is computed to obtain an overall performance comparison. The weighted mean of a theorem prover is computed using the following formula :

$$Mean_{TP} = \frac{\sum_{i=1}^{m} (SPD_{TPi} * LD_i)}{\sum_{i=1}^{m} LD_i}$$

where :

LD_i = the level of difficulty of problem i

m = the number of problems considered

As shown in Table 3, there are four rows of weighted means. The first row is the weighted means for ME-TP, GC-TP and SLM5-TP only. The weighted means of SL-TP and SLM-TP were not included in this row because SL-TP did not produce results in CHANG-2 and CHANG-4 while SLM-TP did not produce a result in the SCHUBERT problem. The weighted means in this row show that SLM5-TP performed significantly better than the ME-TP and GC-TP in all the non-trivial problems. The second row of weighted means provides an overall comparison between ME-TP, SL-TP, GC-TP and

weighted mean of SLM-TP is not included in this row because it gave no result in the SCHUBERT problem. This row also shows the superiority of SLM5-TP over the others in solving the non-trivial problems considered. The third row of weighted means shows the overall performance of ME-TP, GC-TP, SLM-TP and SLM5-TP in solving the non-trivial problems except the SCHUBERT problem. SL-TP is not included because it has no results in CHANG-2 and CHANG-4. It is notable that SLM-TP performed better than the ME-TP and GC-TP in solving the non-trivial problems considered. SLM5-TP still came out as the best. In the last row, the computed weighted means exclude the results obtained in problems CHANG-2, CHANG-4 and SCHUBERT. The results show that SLM5-TP performed better than the others in solving the considered problems. The results of ME-TP in these problems are closely comparable to the results of SLM5-TP. Overall, the SLM5-TP performed better than the other theorem provers in solving the non-trivial problems.

Chapter 5

CONCLUSION

5.1. Summary of Features

In the analysis of s-linear resolution, ME procedure, t-linear resolution, SL-resolution, GC procedure and SLM derivation strategies, the following major features were found :

i) Extension operation
   a) selection function
   b) resolvants of ancestor resolution subsume the center clause.

ii) Reduction operation
   a) compulsory ancestor resolution on literals having atoms which are or become identical.
   b) compulsory merging operation.
   c) compulsory C-reduction.
   d) compulsory reduction when the literals involved have identical atoms and no A-literal indexed by 1 is between them

iii) Truncation operation
   a) production of lemmas
   b) creation of C-literals
   c) insertion of A-literals created from truncated A-literals at more than one position

iv) Spreading operation

v) Syntactic checks
   a) Admissibility restrictions
   b) Hyperminimality restriction

vi) Semantic check

In s-linear and t-linear resolutions, the literals of the center clause are resolved away in any order. This means that even if each literal of the center clause can be resolved away in one way, there would still be N! derivations where N is the number of literals in the center clause. Hence, s-linear and t-linear derivations have large search trees. This inefficiency of s-linear and t-linear resolutions was eliminated by the ME procedure, SL-resolution, GC procedure and SLM by using a selection function which only selects one literal from a center chain to resolve upon. Consequently, the search trees of the ME procedure, SL-resolution, GC procedure and SLM are smaller compared to s-linear and t-linear search trees. The selection function is in effect similar to the strategy used in ordered input resolution. The restriction that the resolvants of ancestor resolution must subsume the previous center clause is imposed in different ways by the six derivation strategies. This restriction makes sure that newly introduced literals can factor with existing literals in the center clause.

The reduction operation is an inference rule used in the ME procedure, SL-resolution, the GC procedure and SLM. The reduction operations of the ME procedure, SL-resolution, the GC procedure and SLM are partly implementing the implicit merging operations of s-linear and t-linear resolutions. The reduction operation of SL-resolution is more powerful than that of the ME procedure because it includes ancestor resolution and explicit factoring. The compulsory reduction operation of the GC procedure is a more strict application than SL-resolution because it does not require that the literals involved have atoms that are, or become identical. SLM imposes compulsory reduction only if the literals involved have identical atoms and no A-literal indexed by 1 is between them. It has been shown that this restriction, together with the hyper minimality restriction, maximises the use of recycled A-literals.

The truncation operation is employed in the ME procedure, SL-resolution, the GC procedure and SLM. The truncation operation of SL-resolution only removes A-

literals from the center chain. The ME and GC procedures make use of all truncated A-literals. SLM reuses truncated A-literals indexed by 1. The ME procedure recycles truncated A-literals in the form of lemmas. The GC procedure implements the recycling of truncated A-literals via C-literals. This is more efficient than the lemma mechanism of the ME-TP because a C-reduction of the GC-procedure is equivalent to one extension using a lemma, removal of other B-literals from the lemma by reduction and a truncation. SLM reuses truncated A-literals indexed by 1, by inserting them as A-literals indexed by 0, at possibly more than one position in the center chain. Allowing the insertion of recycled A-literals at more than one position, however, produces irrelevant derivations which expand the search tree.

The spreading operation used in SLM allows each spread literal to be resolved away concurrently with the others, by interleaving the operations on the branches. In the ME procedure, SL-resolution and the GC procedure, the selection function has limited choice of literals because only B-literals in the rightmost cell of a one 'branch' center chain, are considered. If a literal is not in the rightmost cell cannot be resolved away, possibly as a result of instantiation, the detection of this problem has to wait until all literals to the right of that literal are resolved away. Spreading partially alleviates this problem by allowing the selection function to choose a B-literal to resolve on from any of the center chain branches.

The six derivation strategies impose syntactic checking to trim redundant derivations. Loveland proved that a refutation as small as a minimal non-linear resolution does not contain tautologous clauses, and a restriction to ensure this is imposed in the s-linear and SL-reslution strategies. The ME and the GC procedures are not compatible with the no-tautologies restriction. The ME procedure imposes three preadmissibility restrictions. The first one prevents using tautologous input chains during extension operations. The second preadmissibility restriction prevents the occurence of endless loops in a derivation. The third preadmissibility restriction enforces compulsory reduction on B-literals which have identical atoms with preceding A-literals. It also serves as a

retrospective check of the second preadmissibility restriction. SL-resolution imposes a more restrictive syntactic restriction than the ME procedure, which reject tautoilogies and enforce factoring of B-literals that are or become identical. In the GC procedure, an extended implementation of the first two preadmissibility restrictions of the ME procedure is imposed. SLM's hyper minimality restriction and the restrictions that satisfy the application of compulsory reduction, maximise the use of recycled A-literals. The hyper minimality restriction also partly prevents the occcurence of loops in the derivation.

SLM applies semantic checking to each literal in the center chain using a given interpretation, during extension and reduction operations. The practical effect of semantic checking is the pruning of some irrelevant derivations from the search tree. However, there is difficulty in implementing a non-trivial interpretation. Two problems were identified by Heschen (1976) in the implementation of a non-trivial interpretation. First, it is difficult to determine whether or not a clause containing variables is falsified, especially for interpretations whose domains are not fairly small. The second problem is the difficulty of finding a general representation of an interpretation with a reasonable storage requirements.

5.2. Systems' Performance

SLM may have longer refutations than those obtained by the ME procedure, SL-resolution or the GC procedure because it does not factor nor ancestor resolve B-literals indexed by 0. However, SLM produces a narrower search tree than those obtained by the other three derivation strategies because it has a lesser number of reductions. It has also been shown that SLM can obtain a refutation using more of the top clauses from the set of support than the GC procedure. This feature is desirable in certain applications which require the ability to obtain a refutation from a specific top clause.

SLM-1, SLM-2 and SLM-3 are variations of SLM designed to reduce the number of irrelevant derivations caused by the indeterminancy of inserting A-literals created from truncated A-literals. SLM-1 minimises the number of irrelevant derivations by only inserting the A-literal in a position that is not equivalent to a previous position. However, this has a negative effect because the restriction may prevent a refutation for a certain top clause. SLM-2 always inserts the A-literal created from a truncated A-literal at its depth. This solves the indeterminancy problem, but its effect is bought at the expense of omitting the hyper minimality and compulsory reduction restrictions. SLM-3 solves the indeterminancy problem by not recycling A-literals. Obviously, SLM-3 may obtain longer refutations because some literals will be repeatedly resolved away when they could be simply reduced using a recycled A-literal. SLM-4 and SLM-5 are new variations of SLM which are intended to alleviate the problems of SLM. SLM-4 requires a set of clauses which has a non-Horn model to be broken down into subsets of clauses such that each subset has a Horn model. Each subset has to be refuted using SLM-4. SLM-4 traps loops by not allowing identical A-literals to coexist in any center chain. A-literals are simply deleted during truncation which solves the indeterminancy problem. The problem with this variation of SLM lies in the difficulty of splitting a set of clauses that contains many clauses which have more than one literal indexed by 1. SLM-5 is formulated based on the problems encountered in SLM and SLM-1. It maintains the main features of SLM but modifies the equivalent position restriction of SLM-1, adds more restrictions to detect loops, modifies the extension operation such that a unit chain is selected first as input parent chain, and modifies the reduction operation definition by always selecting the B-literals to be reduced from the rightmost cell. SLM-5 suppresses the spreading operation if the set of clauses is written at propositional level. The reduction operation is also suppressed in refuting a set of Horn clauses. SLM-5 makes use of proved literals which are not subsumed by any unit chains, by adding them to the input chain database as unit chains. However, the addition of proved literals is only done when refuting a set of non-Horn clauses. This restriction is based on the idea that linear input resolution is complete for sets of Horn clauses, thus, the addition of more clauses to the original set of Horn clauses may only expand the search tree.

It is conjectured that SLM-5 answers the question posed by Brown (1974, p. 32) - "is there an interesting linear inference system whose refutations are bounded by the complexity of some hyper minimal refutations ?" He suggested that by always placing "an A-literal created by truncation, at its depth and no where else, leads to such a system". Although SLM-5 does not follow this suggestion precisely, SLM-5 produces shorter refutations and smaller search trees than SLM. SLM-5 minimises the repetitive resolving away of literals because it reuses more truncated A-literals than SLM. The addition of proved literals to the input chain database partly solves the indeterminancy problem of SLM in inserting truncated A-literals. SLM-5 has better restrictions than SLM for detecting loops and pruning irrelevant derivations from the search tree.

The ME procedure, SL-resolution, GC procedure, SLM and SLM-5 were implemented in PROLOG. The Arity/PROLOG compiler was used to compile the implemented theorem provers. The unit preference strategy, set of support strategy, pure literal elimination, elimination of tautologies, match check, selection function based on a computed weight of B-literals, and a modified consecutively bounded depth first search strategy were included in the implementations. The extension operation was also extended to include subsumed unit extension and paramodulation.

The implemented theorem provers were tested using twenty four selected problems. The results show that none of the theorem provers consistently perform better than the others. The effects of the lemma generation in the ME-TP, the initial factoring of input clauses and factoring operation during derivations used in SL-TP, the C-literal mechanism of GC-TP, the spreading operation of SLM-TP and SLM5-TP, and the generation of proved literals in SLM5-TP were compared.

In some problems, the addition of lemmas to the input chain database has positive effects on the derivation. However, the addition of lemmas to the input chain database also increases the number of possible input chains. As a result, the ME-TP

produces larger search trees in some of the problems, compared to the other theorem provers.

The results show that the factoring of input clauses before converting them to input chains has a positive effect in SL-TP derivations, particularly in conjunction with the unit preference strategy. The results show that SL-TP produced shorter refutations than the other theorem provers for some problems. This is because some input clauses were factored prior to the derivation and these factors were used as input parent chains during extension operations, thus, minimising the number of factoring operations applied. However, this initial factoring also causes problems in SL-TP derivations, especially on recursive type problems such as those presented in Chang (1970). This is because some of the factors cause irrelevancy in the search space. The initial factoring of the input clauses also increases the number of input clauses, because a clause may produce more than one factor.

In the experimentation, the GC-TP produced better results than the other theorem provers for some problems. This efficiency of GC-TP is brought about by its C-literal mechanism. Two positive effects of C-literal mechanism were observed. Firstly, the C-literal mechanism in conjunction with the syntactic restrictions of GC-TP helps trim the search space of some redundant derivations. Secondly, reducing a B-literal with a C-literal reduces the length of a refutation because repeated resolving away of B-literals is minimised. These effects were shown in the results of some problems. However, in a consecutively bounded depth-first search environment where more than one search bound is required, the lemma generation of the ME-TP appears to be better than C-reduction because lemmas generated in one search bound are carried over to the next search bound. An examination of results also revealed that the insertion of C-literals may expand the search tree, particularly in sets of Horn clauses where C-literals are all proved literals. This may expand the search tree because some of these C-literals may subsume some unit input chains. Thus, a B-literal of the center chain may be reduced by a C-literal or be resolved upon by a unit chain.

The results of some of the problems show the positive effect of the spreading operation used in SLM-TP and SLM5-TP. These results confirm Brown's arguments which point out that having more flexibility in the selection of literal to resolve on is critical for the performance of a derivation strategy. However, it was discovered that the spreading operation is less effective in problems written at propositional level. The results show that SLM5-TP, which suppresses the spreading operation for problems written at propositional level, performs better than SLM-TP.

Based on the overall results, SLM5-TP performed consistently well. It solved all the 24 selected problems within a reasonable memory use and execution time, and never produced the worst result on difficult problems.

5.3. Future Directions

A number of problems which are worthy of further investigation are suggested.

5.3.1. Improving the consecutively bounded depth first search strategy.

Early version of each theorem provers placed the bound on the number of inferences. The early version produced different results than when the bound is placed on the number of A- and B-literals in the center chain. In some cases the earlier version gave better results than the second version but in some problems it also gave poor results. The question is - are there better ways of implementing the consecutively bounded depth-first search strategy that give better and more stable results? The works of Nie and Plaisted (1989) on the refinements of this search strategy may provide the answer to this question.

5.3.2. Improving the selection function.

It was found in the experimentation that the implemented selection function significantly improved the performance of the six derivation strategies. Although it requires great effort to compute the weight of each candidate literal to be resolved on, using the selection function still gives better results compared to simply selecting the rightmost literal. This indicates the importance of the selection function to a theorem prover. Hence, it is worth investigating whether a better way of implementing a selection function can be found that requires less effort but is as effective or more effective than the selection function used in this study.

5.3.3. Implementing a more complex interpretation.

The implemented interpretation for SLM-TP and SLM5-TP is the trivial interpretation which interprets all positive literals as TRUE and all negative literals as FALSE. It would be worth an investigation to find a general representation of a non-trivial interpretation which can be implemented.

REFERENCES

Amble, T. (1987). Logic Programming and Knowledge Engineering, Addison-Wesley Pub., Great Britain.

Andrews, P.B. (1968). Resolution with Merging. Journal of the Association for Computing Machinery, 15(3), pp. 367-381.

Arity Corporation (1988). Arity/Prolog, Concord, Massachusetts.

Bledsoe, W. (1981). Non-resolution Theorem Proving. In B. L. Webber and N. J. Nilsson (Eds.), Readings in Artificial Intelligence, Morgan Kaufmann Publishers, Inc., California, pp. 90-108.

Bratko, I. (1986). PROLOG Programming for Artificial Intelligence, Addison-Wesley Publishing Co. Inc., Great Britain.

Brown, Frank M. (1974). SLM. Memo No. 72, May.

Brown, Cynthia (1984). A Self-Modifying Theorem Proving. Proceedings of the National Conference on Artificial Intelligence, 1, The American Assocication for Artificial Intelligence, USA, pp. 38-41.

Chang, C. L. (1970). The Unit Proof and the Input Proof in Theorem Proving. Journal of the Association for Computing Machinery, 17(4), pp. 698-707.

Chang, C. L. (1972). Theorem Proving with Variable-Constrained Resolution. Inform. Sci. 4, pp. 217-231.

Chang, C. and Slagle, J. (1979). Using Rewriting Rules for Connection Graphs to Prove Theorems. Artificial Intelligence, 12(2), pp. 159-178.

Clocksin, W. and Mellish, C. (1987). Programming in Prolog, 3rd. ed., Springer-Verlag, Germany

Genesereth, M. and Nilsson, N. (1987). Logical Foundation of Artificial Intelligence, Morgan Kaufmann Publishers, Inc., California.

Green, C. (1981). Application of Theorem Proving to Problem Solving. In B. L. Webber and N. J. Nilsson (Eds.), Readings in Artificial Intelligence, Morgan Kaufmann Publishers, Inc., California, pp. 202-222.

Henschen, L. J. (1976). Semantic Resolution for Horn Sets. IEEE Transactions on Computers, 25(8), pp. 816-822.

Henschen, L. J and Wos, L. (1974). Unit Refutations and Horn Sets. Journal of the Association for Computing Machinery, 21(4), October 1974, pp. 590-605.

Hunt, E. B. (1975). Artificial Intelligence, Academic Press Inc., New York.

Korf, R. F. (1985). Depth-first Iterative Deepening : An Optimal Admissible Tree Search. Artifical Intelligence, 27, pp. 97-109.

Kowalski, R. and D. Kuehner (1971). Linear Resolution with Selection Function. Artificial Intelligence, North-Holland Publishing Co., pp. 227-260.

Kowalski, R. (1975). A Proof Procedure Using Connection Graph. Journal of the Association for Computing Machinery, 22(4), pp. 572-595.

Kowalski, R. (1982). Logics as a Computer Language. In K.L. Clark amd S. A. Tarnlund (Eds.), Logic Programming, Academic Press, Inc., pp. 3-18

Loveland, D. W. (1968a). A Linear Format for Resolution. Symposium on Automatic Demonstration, Lecture Notes in Mathematics 125, Springer-Verlag, Berlin and New York.

Loveland, D. W. (1968b). Mechanical Theorem Proving by Model Elimination. Journal of Association of Computing Machinery, 15(2) pp. 236-251.

Loveland, D. W. (1969a). Theorem-provers Combining Model Elimination and Resolution. Machine Intelligence 4, pp. 73-86.

Loveland, D. W. (1969b). A Simplified Format for the Model Elimination Theorem Proving Procedure. Journal of Association of Computing Machinery, 16,3, pp. 349-363.

Luckham, D. (1970). Refinement Theorems in Resolution Theory. Symposium on Automatic Demonstration, Lecture Notes in Mathematics 125, Springer-Verlag, Berlin and New York, pp. 163-177.

Luger, G. and Stubblefield, W. (1989). Artificial Intelligence and the Design of Expert Systems, The Benjamin/Cumming Publishing Co. Inc.

McCharen, J., R. Overbeek and L. Wos(1976). Complexity and related enhancements for Automated Theorem-Proving programs. Computers and Mathematics with Applications 2, pp. 1-16.

Nie, Xumin (1990). Model Elimination and Its Positive Refinement. AAR Newsletter. 15 May 1990.

Nie, X. and D. A. Plaisted (1989). Refinements to Depth-First Iterative-Deepening Search in Theorem Proving. Aritficial Intelligence, 41, pp. 223-235.

Pelletier, J.F. (1986). Seventy-Five Problems for Testing Automatic Theorem Provers. Journal of Automated Reasoning 2, Reidel Publishing Co., pp. 191-216.

Plaisted, D. A. (1989). A Sequent Style Model Elimination Strategy and a Positive Refinement. Journal of Automated Reasoning, in press. (Also TR89-014, Department of Computer Science, University of North Carolina, Chapel Hill, 1989.)

Ramsay, A. (1988). Formal Methods in Artificial Intelligence, Cambridge University Press, Great Britain.

Ringwood, G. A. (1988). SLD: A Folk Acronym? Newsletter of the Association for Logic Programming, 2,1, pp. 5-6.

Robinson, J. A. (1965). A Machine Oriented Logic based on the Resolution Principle. Journal of Association of Computing Machinery, 12,1, pp. 23-41.

Robinson, J. (1965). Automatic Deduction with Hyper-resolution. International Journal of Computer Mathematics 1, pp. 227-234.

Robinson, J. (1979). Logic : Form and Function, Elsevier North Holland, New York.

Satz, Ronald (1988). Solution to Schubert's Steamroller Problem, AAR Newsletter, January, p. 7.

Slagle, J. (1971). Artificial Intelligence: The Heuristic Programming Approach, McGraw-Hill Series, New York.

Santane-Toth, E. and P. Szeredi (1982). 'PROLOG Applications in Hungary', Logic Programming, eds. K.L. Clark amd S. A. Tarnlund, Academic Press, Inc., pp. 19-32.

Shostak, R. E. (1976). Refutation Graphs, Artificial Intelligence, North Holland Publishing Co.

Sickel, S. (1976). A Search Technique for Clause Interconnectivity Graphs. IEEE Transactions on Computers, 25(8), pp. 823-835.

Stickel, M. E. (1982). A Nonclausal Connection-Graph Resolution Theorem Proving Program. Proceedings of the National Conference on Artificial Intelligence, Morgan Kauffman,Pittsburgh, PA, Los Altos California, pp. 229-233.

Stickel, M. E.(1984). A Prolog Technology Theorem Prover. Proceedings of the First International Symposium on Logic Programming, pp. 211-217.

Stickel, M. E. (1986). A Prolog Technology Theorem Prover : Implementation by an Extended Prolog Compiler. In Goos, G. and J. Hartmanis (Eds.), 8th Conference on Automated Deduction, Springer-Verlag, pp. 573-587.

Stickel, M. E. (1986). Schubert's Steamroller Problem : Formulations and Solutions. Journal of Automated Reasoning 2, Reidel Publishing Co., pp. 89-101.

Stickel, M. E. (1987). An Introduction to Automated Deduction. In Bibel, W. and Ph. Jorrand (Eds.), Fundamentals of Artificial Intelligence, Springer-Verlag, pp. 75-132.

Stickel, M.E.and W. M. Tyson (1985). An Analysis of Consecutively Bounded Depth-First Search with Applications in Automated Deduction. Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 1073-1075.

Sutcliffe, G. (1989). A General Clause Theorem Prover written in Prolog. Research Report 1989/2, Western Australian College of Advanced Education, Dept. of Computer Studies.

Sutcliffe, G. and W. Tabada (1990). Compulsory Reduction in LInear Derivation Systems. submitted as Research Note to Artificial Intelligence

Tabada, W. and G. Sutcliffe (1990). An Analysis and Comparative Study of Five Linear Derivation Strategies. Research Report 1990/1, Western Australian College of Advanced Education, Dept. of Computer Studies.

Tarnlund, S.A. (1988). Test of an Inference System for Parallel Logic Programming, Newsletter of the Association for Logic Programming, July, p. 7.

Wos, L., D.F. Carson, and G.A. Robinson (1964). The Unit Preference Strategy in Theorem Proving. Proc. AFIPS Annual Fall Joint Computer Conference, 26(1), Spartan Books, New York, pp. 615-621.

Wos, L., D.F. Carson, and G.A. Robinson . (1965). Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. Journal of Association of Computing Machinery, 12(4), pp. 536-541.

Wos, L., Robinson, J. A., and Carson, D.F. (1967). The Concept of Demodulation in Theorem Proving. Journal of Association of Computing Machinery, 14, pp. 698-709.

Wos, L, and G. A. Robinson (1968). Paramodulation and Set of Support. Proceedings Symposiom Automatic Demonstration, Versailles, France, 1968, Springer-Verlag, New York (1970), pp. 276-310.

Appendix A

ALGORITHMS

A.1. Derivation Algorithm

The general algorithm in obtaining a derivation is as follows :

```
DO WHILE the current center chain is not an empty list
    Apply the inference rules to the current center chain to
    obtain a new center chain
    IF the new center chain satisfies the restrictions then
        Let the new center chain be the current center chain
    ELSE
        Backtrack and find another possible solution
    END-IF
END-DO
```

A.2. Inference Operation Selection Algorithm

A.2.1. For the ME procedure, SL-resolution and the GC procedure.

```
IF the first element of the current center chain is a non-B-
literal then
    Apply the truncation operation to the current center
    chain to obtain new center chain (no backtracking
    allowed)
ELSE
```

```
    IF reduction of literals with identical atoms is
    possible then
        Apply compulsory reduction (no backtracking allowed)
    ELSE
        Apply the reduction operation OR the extension
        operation to obtain a new center chain
    END-IF
END-IF
```

A.2.2. For SLM and SLM-5.

```
IF the last non-reduction operation was a truncation of an
A-literal indexed by 1 then
    Select the branch where the A- or C-literal generated
    from truncated A-literal is inserted
    IF the compulsory reduction restriction is satisfied
    then
        Apply compulsory reduction
    ELSE
        Apply reduction to the branch OR the extension
        operation to the center chain
    END-IF
ELSE
    IF the last non-reduction operation was the spreading
    operation then
        Apply the extension operation
    ELSE
        IF a branch of the center chain is truncatable then
```

```
            Apply the truncation operation (no backtracking
            allowed if the truncated A-literal is indexed by
            0)
        ELSE
            IF the rightmost cell of the newly inferred
            branch of the center chain contains more than
            one B-literal indexed by 0 then
                Apply the spreading operation
            ELSE
                IF the compulsory reduction restriction is
                satisfied then
                    Apply compulsory reduction
                ELSE
                    Apply reduction to the branch OR the
                    extension operation to the center chain
                END-IF
            END-IF
        END-IF
    END-IF
END-IF
```

A.3 Selection Function Algorithm

A selection function is used to select a B-literal from the center chain to resolve on during an extension operation.

A.3.1. Selection function algorithm for the ME procedure, SL-resolution and the GC procedure.

```
Extract the rightmost cell from the center chain.
IF the rightmost cell contains only one literal then
        Let the literal be the selected literal.
ELSE
    Compute the weight for each literal of the rightmost
    cell.
    Let the literal with the minimum weight be the selected
    literal.
END-IF
```

A.3.2. Selection function algorithm for SLM and SLM-5.

```
Extract all tip nodes from the center chain
Extract the rightmost cell for each tip node
Collect all B-literals from the rightmost cells
Remove all B-literals which are identical to any A-literals
of the center chain
IF all B-literals have identical A-literals then
    Let the collected B-literals be the list of literals to
    be considered for selection (candidate list)
ELSE
    Let the rest of the collected B-literals be the
    candidate list
END-IF
IF the candidate list contains a single literal then
    Let the single literal be the selected literal
ELSE
```

```
    Compute the weight for each B-literal of the candidate
    list
    Let the B-literal with the minimum weight be the
    selected literal
END-IF
Find the tip node where the selected B-literal belongs
```

A.4 Extension Operation Algorithm

Generally, the extension operation is implemented using the following algorithm:

```
Select a B-literal L to resolve on from the center chain
using the selection function.
IF a unit chain is available and its literal K subsumes the
negation of L then
    Apply binary resolution using the unit chain as the
    input parent chain (no backtracking allowed).
ELSE
     IF an input chain C has a B-literal K which is
    complementary unifiable with L then
        Apply binary resolution to L using C as the input
        parent chain (allow backtracking to select another
        literal from C or get another input chain)
        OR
        IF an equality literal exists then
            Apply paramodulation to L
        END-IF
    END-IF
END-IF
```

A.5 Match Check Algorithm

The algorithm for checking whether a center chain contains a literal which cannot be extended upon is as follows:

```
IF a B-literal L which is sufficiently instantiated can be
selected from the center chain then
    IF the atom of L subsumes a valid literal atom
        Select another B-literal from the center chain and
        check
    ELSE
        IF the atom of L is subsumed by one of the redundant
        literal atom then
            The check fails
        ELSE
            IF L can be extended upon with an input chain C
            then
                IF all the other literals of C can be
                extended then
                    Assert the atom of L as valid literal
                    atom, select another B-literal from the
                    center chain and check
                ELSE
                    Assert the atom of L as redundant
                    literal atom and the check fails
                END-IF
            ELSE
```

```
                Assert the atom of L as redundant literal
                atom and the check.fails
            END-IF
        END-IF
    END-IF
ELSE
    The check succeeds
END-IF
```

Appendix B

LISTING OF PROGRAM'S SOURCE CODE

```
/* ==========================================================
     Module:     METP.ARI
     Purpose:    Main program of the ME-TP theorem prover
     Required Modules :
                 INTERPRE.ARI
                 DRIVER.ARI
                 SET_MAN.ARI
                 ME_RULES.ARI
                 SUPPORT.ARI
========================================================== */

% ---------------------------------------------------------------------------------------
%          Invoke when CTRL BRK is pressed

% ---------------------------------------------------------------------------------------
restart :-
    what_to_do.
what_to_do :-
    print(1,[nl,$The CTRL-BRK was pressed. Exit to DOS (Y/N) ? $]),
    get0(X),
    member(X,"Yy"),
    halt.

what_to_do :-
    main.

% ---------------------------------------------------------------------------------------

%          The main procedure of the theorem prover

% ---------------------------------------------------------------------------------------
main :-
    cls,
    fileerrors(_, off),
    asserta(ds(me)),
    introduction,
    repeat,
        print(1,[nl,nl,$ME-TP :- $]),
        ratom(Command),

interpreter(Command).
```

```
/* ============================================================
      Module:     SLTP.ARI
      Purpose:    Main program of the SL-TP theorem prover
      Required Modules :
                  INTERPRE.ARI
                  DRIVER.ARI
                  SET_MAN.ARI
                  SL_RULES.ARI
                  SUPPORT.ARI
============================================================ */

% ---------------------------------------------------------------------------------------
%           Invoke when CTRL BRK is pressed

% ---------------------------------------------------------------------------------------
restart :-
    what_to_do.
what_to_do :-
    print(1,[nl,$The CTRL-BRK was pressed. Exit to DOS (Y/N) ? $]),
    get0(X),
    member(X,"Yy"),
    halt.

what_to_do :-
    main.

% ---------------------------------------------------------------------------------------

%           The main procedure of the theorem prover

% ---------------------------------------------------------------------------------------
main :-
    cls,
    fileerrors(_, off),
    asserta(ds(sl)),
    introduction,
    repeat,
        print(1,[nl,nl,$SL-TP :- $]),
        ratom(Command),

interpreter(Command).
```

```
/* ============================================================
      Module:     GCTP.ARI
      Purpose:    Main program of the GC-TP theorem prover
      Required Modules :
                  INTERPRE.ARI
                  DRIVER.ARI
                  SET_MAN.ARI
                  GC_RULES.ARI
                  SUPPORT.ARI
============================================================ */

:% ---------------------------------------------------------------------------------------
%           Invoke when CTRL BRK is pressed

% ---------------------------------------------------------------------------------------
restart :-
    what_to_do.
what_to_do :-
    print(1,[nl,$The CTRL-BRK was pressed. Exit to DOS (Y/N) ? $]),
    get0(X),
    member(X,"Yy"),
    halt.

what_to_do :-
    main.

% ---------------------------------------------------------------------------------------

%           The main procedure of the theorem prover

% ---------------------------------------------------------------------------------------
main :-
    cls,
    fileerrors(_, off),
    asserta(ds(gc)),
    introduction,
    repeat,
        print(1,[nl,nl,$GC-TP :- $]),
        ratom(Command),

interpreter(Command).
```

```
/* ============================================================
      Module:     SLM.ARI
      Purpose:    Main program of the SLM-TP theorem prover
      Required Modules :
                  INTERPRE.ARI
                  SLM_DRV.ARI
                  SLM_SEL.ARI
                  SLM_SUP.ARI
                  SLM_RULE.ARI
                  SLM_REST.ARI
                  SUPPORT.ARI
============================================================ */

:% ---------------------------------------------------------------------------------------------
%           Invoke when CTRL BRK is pressed

% ---------------------------------------------------------------------------------------------
restart :-
    what_to_do.
what_to_do :-
    print(1,[nl,$The CTRL-BRK was pressed. Exit to DOS (Y/N) ? $]),
    get0(X),
    member(X,"Yy"),
    halt.
what_to_do :-
    main.

% ---------------------------------------------------------------------------------------------

%           The main procedure of the theorem prover

% ---------------------------------------------------------------------------------------------
main :-
    cls,
    fileerrors(_, off),
    asserta(slm_version(1)),
    introduction,
    repeat,
        print(1,[nl,nl,$SLM-TP :- $]),
        ratom(Command),
        interpreter(Command).
```

```
/* ============================================================
      Module:     SLM5.ARI
      Purpose:    Main program of the SLM5-TP theorem prover
      Required Modules :
                  INTERPRE.ARI
                  SLM_DRV.ARI
                  SLM_SEL.ARI
                  SLM_SUP.ARI
                  SLM5_RUL.ARI
                  SLM5_RES.ARI
                  SUPPORT.ARI
============================================================ */

:% ---------------------------------------------------------------------------------------------
%           Invoke when CTRL BRK is pressed

% ---------------------------------------------------------------------------------------------
restart :-
    what_to_do.
what_to_do :-
    print(1,[nl,$The CTRL-BRK was pressed. Exit to DOS (Y/N) ? $]),
    get0(X),
    member(X,"Yy"),
    halt.
what_to_do :-
    main.

% ---------------------------------------------------------------------------------------------

%           The main procedure of the theorem prover

% ---------------------------------------------------------------------------------------------
main :-
    cls,
    fileerrors(_, off),
    asserta(slm_version(5)),
    introduction,
    repeat,
        print(1,[nl,nl,$SLM5-TP :- $]),
        ratom(Command),
        interpreter(Command).
```

```
/* ============================================================
      Module:     ME_RULES.ARI
      Purpose:    Contains the operation and syntactic restrictions
============================================================ */

% ---------------------------------------------------------------------------------------
%           Reduction operation for the ME procedure

% ---------------------------------------------------------------------------------------
reduce(Chain, Resolvant, _, _) :-
      clause_type(horn), !, fail.
reduce(Chain, Resolvant, [b,L], Type) :-
      clause_type(general),  % No reduction for Horn clauses
      choose([a,S,K], Prec, Succ, Chain),
      [! select([b,L], Others, Prec),
      reducible(L, K, Type),
      count_A(NS, Prec),
      append(Others, [ [a,NS,K]|Succ] , Resolvant) !].

% ---------------------------------------------------------------------------------------

%           Check if the two literals are reducible

% ---------------------------------------------------------------------------------------
reducible(L,K, Type) :-
        complementary(L,K, L_A,K_A),
        match(L_A, K_A, Type).

% ---------------------------------------------------------------------------------------
%           Find if the two Atoms are identical or unifiable

% ---------------------------------------------------------------------------------------
match(A1, A2, id) :-
        identical_atom(A1,A2), !.
match(A1, A2, unify) :-
        unify(A1, A2).

% ---------------------------------------------------------------------------------------
%           Count the number of A-literals on the left side of the chain

% ---------------------------------------------------------------------------------------
count_A(0, []) :- !.
count_A(N, [ Literal|Chain ]) :-
      count_A(N1, Chain),
      increment_A_ctr(Literal,N1, N).

% ---------------------------------------------------------------------------------------
%           Increment the A-ctr if the literal is an A-literal

% ---------------------------------------------------------------------------------------
increment_A_ctr([a|_],N1, N) :-
        N is N1+1, !.
increment_A_ctr(_, N, N).

% ---------------------------------------------------------------------------------------
%           ME Truncation  operation

% ---------------------------------------------------------------------------------------
truncate([ [a,S,L] | Rest_Chain ], Lemma, Resolvant) :-
            truncate_all([ [a,S,L] | Rest_Chain ], Lemma, Resolvant).
```

```
% ----------------------------------------------------------------------------------------
%           Truncate all the preceeding A-literals

% ----------------------------------------------------------------------------------------
truncate_all([ [b,L] | Rest_Chain ], [], [ [b,L] | Rest_Chain ]) :- !.
truncate_all([], [], []) :- !.
truncate_all([ [a,S,L] | Rest_Chain ], [Index|Lemma_Ndx], Result) :-
            form_a_lemma(0,Lemma, Rest_Chain),
            update_scopes(Rest_Chain, Updated),
            negate(L,K),
            insert_lemma(Index, [K|Lemma]),
            truncate_all(Updated, Lemma_Ndx, Result).

% ----------------------------------------------------------------------------------------
%           Form a lemma
% Algorithm:

%           Pick up an A-literal and evaluate if its scope exceeds then number of A-

%           literals (A_Ctr) preceeding it. Increment the counter and pick up again

%           another A-literal from the rest of the chain until no more A-literals.

% ----------------------------------------------------------------------------------------
form_a_lemma(A_Ctr, Lemma, Chain) :-
            pick_suc([a,Scope,L], Succ, Chain),
            evaluate(A_Ctr, Scope, L, Lemma, Lemma_Rest),
            inc(A_Ctr, N),
            form_a_lemma(N, Lemma_Rest, Succ), !.
form_a_lemma(_, [], Rest).

% ----------------------------------------------------------------------------------------

%           Evaluate if the scope of an A-literal exceeds the number of preceding literal

% ----------------------------------------------------------------------------------------
evaluate(A_Ctr, Scope, L, [K|Lemma], Lemma) :-

      Scope > A_Ctr,

      negate(L,K), !.

      evaluate(A_Ctr, Scope, L, Lemma, Lemma).

% ----------------------------------------------------------------------------------------
/*                        Update the Scopes of the following A-literals
Algorithm:
    (1) If the rest of center chain is empty then the update is also
                empty and stop the recursion.
            else
    (2)  if the first element of center chain is an A-literal then
                    insert this A-literal at the update with N as the scope
                    Increment the scope by 1
                    Continue the update of the rest of center chain
              else
    (3)     - Insert the first literal of the center chain and insert
                        it as it is at the update
                    Continue the update of the rest of center chain          */

% ----------------------------------------------------------------------------------------
```

```
update_scopes(Chain, Updated) :-
        choose([a,Scope,L], Prec, Succ, Chain),
        count_A(N, Prec),
        Scope > N,
        append(Prec, [[a,N,L]|Succ], New_Chain),
        update_scopes(New_Chain, Updated), !.

update_scopes(Updated, Updated).

% ----------------------------------------------------------------------------------------
%          Insert a lemma if it is not subsumed by a chain

% ----------------------------------------------------------------------------------------
insert_lemma(0, Lemma) :-      /* -- Disregard if the # of literals is --- */
        clause_size(_,Max),    /* -- greater than or equal the maximum --- */
        length(Lemma, N),      /* -- size of the input clauses ---------- */
        N >= Max, !.
insert_lemma(Index, Lemma) :-
        not subsumed_input(Lemma),
        form_a_chain(Lemma, Input),
        get_chain_ndx(Index),
        store_fact(a, input_chain(Index, Input)).
insert_lemma(0, Lemma).


% ----------------------------------------------------------------------------------------
%          Find an input chain which is subsumed by the lemma

% ----------------------------------------------------------------------------------------
subsumed_input(Lemma) :-
        input_chain(_, Chain),
        equivalent(Lemma, Chain), !.

% ----------------------------------------------------------------------------------------
%          Determine if the lemma is equivalent to the chain

% ----------------------------------------------------------------------------------------
equivalent([],[]) :- !.
equivalent([L|Lemma], Chain) :-
      select([b,K],Rest, Chain),
      unify(K,L),
      equivalent(Lemma, Rest).

% ----------------------------------------------------------------------------------------
%          ME Extension operation

% ----------------------------------------------------------------------------------------
extend(Chain, Resolvant, Input, Type) :-
        selection_function([b, Literal_L], Others, Chain),
        resolve(Literal_L, Right_Cell, Input, Type),
        append(Right_Cell, [ [a,0,Literal_L]|Others ], Resolvant).

% ----------------------------------------------------------------------------------------

/*
                              Preadmissibility Restriction
      The chain is not preadmissible iff:
        (i)  the rightmost cell contains tautology literals
       (ii)  the chain contains an A-literal identical to a following B-literal
```

(iii) the chain contains A-literals with identical atoms.*/

```
% ----------------------------------------------------------------------------------------------
syntax_check(id, _) :- !.
syntax_check(unify, Chain) :-
        pick_suc(L, Others, Chain),
        pick_pre(K, Prec, Others),
        inadmissible(L, K, Prec), !, fail.
syntax_check(unify,_).

% ----------------------------------------------------------------------------------------------
%           Check if the two literals are inadmissible

% ----------------------------------------------------------------------------------------------
inadmissible([b, K], [b,L], In_Between) :-  /* preadmissibility I */
        tautology(L,K),
        all_B_(In_Between), !.
inadmissible([b, K], [a,_,L], _) :-        /* preadmissibility II */
        literal_atom(K, Sign, Atom1),
        literal_atom(L, Sign, Atom2),
        identical_atom(Atom1, Atom2), !.
inadmissible([a,_, K], [a,_,L], _) :-        /* preadmissibility III */
        identical_atom(L,K), !.

% ----------------------------------------------------------------------------------------------

%           Determine if the list contains all B-literals

% ----------------------------------------------------------------------------------------------
all_B_([]) :- !.
all_B_([[b,_]|In_Between]) :-

all_B_(In_Between).
```

```
/* ==========================================================
      Module:    SL_RULES.ARI
      Purpose:   Contains the operations and the syntactic restrictions employed by SL-
                 TP
========================================================== */

% ---------------------------------------------------------------------------------------
%                                  SL Reduction operation

% ---------------------------------------------------------------------------------------
reduce(Chain, Resolvant, B_Literal, Type) :-
      extract_right_most_cell(Right_Most_Cell, [A_Literal|Left_Cells], Chain),
      member(Any_Literal, Left_Cells),
      [! select(B_Literal, Rest_RMC, Right_Most_Cell),
      can_be_reduce(B_Literal, Any_Literal, Type),
      append(Rest_RMC, [A_Literal|Left_Cells], Resolvant) !].

% ---------------------------------------------------------------------------------------
    Determines if the two literals can be factored or ancestor-resolved

% ---------------------------------------------------------------------------------------
can_be_reduce([b,L], [a,K], Type) :-          % --- ancestor resolution ---
      complementary(L, K, L_A,K_A),
      match(L_A, K_A, Type), !.
can_be_reduce([b,L], [b,K], Type) :-          % --- factoring ------------
      literal_atom(L,Sign, L_A),
      literal_atom(K,Sign, K_A),
      match(L_A, K_A, Type).

% ---------------------------------------------------------------------------------------

%           Match the two atoms and return id if they are identical
%           or unify if variables were instantiated

% ---------------------------------------------------------------------------------------
match(A1, A2, id) :-
      identical_atom(A1,A2), !.
match(A1, A2, unify) :-
      unify(A1, A2).

% ---------------------------------------------------------------------------------------
%           SL truncation operation

% ---------------------------------------------------------------------------------------
uncate([ [a,_] | Resolvant ], No_Trunc, Resolvant2) :-
            strip_a_literals(Resolvant, Resolvant2, No_Trunc).

% ---------------------------------------------------------------------------------------
%           Strip the chain from all A-literals preceeding the
%           leftmost B-literal

% ---------------------------------------------------------------------------------------
strip_a_literals([], [], []) :- !.
strip_a_literals([ [b,X]|Rest], [ [b,X]|Rest],[] ) :- !.
strip_a_literals([ [a,_]|Rest], Resolvant, [0|Trunc]) :-
            strip_a_literals(Rest, Resolvant, Trunc).

% ---------------------------------------------------------------------------------------

%           SL Extension operation
```

```
% ----------------------------------------------------------------------------------------
extend(Chain, Resolvant, Input, Type) :-
        selection_function([b, Literal_L], Others, Chain),
        resolve(Literal_L, Right_Cell, Input, Type),
        append(Right_Cell, [ [a, Literal_L]|Others ], Resolvant).

% ----------------------------------------------------------------------------------------
                                /* Admissibility Restriction
    (1) The chain is not admissible if the left cells contain
            identical atoms.
    (2) The chain is not admissible if the rightmost cell
            contains B-literal which is tautologous (identical atom) to
            any of the B-literals in the left cells, OR
            the rightmost cell contains literal which is identical to
            any of the A-literals in the left cells
    (3) The input chain is not admissible because the unification find
            it to be tautologous (restriction c.ii of reduction definition).
    (4) The input chain is not admissible because the unification find
            that it contains factorable literals or tautologous literals
            (restriction c.ii of reduction definition). */

% ----------------------------------------------------------------------------------------
syntax_check(id, _) :- !.
syntax_check(Type, Chain) :-
        extract_right_most_cell(RMC, LC, Chain),
        inadmissible(RMC, LC),
        !, fail.
syntax_check(_, _).

% ----------------------------------------------------------------------------------------
%           Check if the chain is admissible by checking
%           (1) If the rightmost cell does not contain a B-literal which is
%                       (a) tautologous to one of the B-literals in the left cells
%                       (b) identical to one of the A-literals in the left cells
%           (2) If the left cells does not contain two literals having identical atoms

% ----------------------------------------------------------------------------------------
inadmissible(RMC, LC) :-                % *** RMC should not contain identical atoms
      pick_suc([b,L], Succ, RMC),
      member([b,K], Succ),
      identical_atom(L,K), !.
inadmissible(RMC, [A|LC]) :-            % *** RMC should not contain B-literal which
      class(A,a,L),                     %***has identical atom with rightmost A-literal
      member([b,K], RMC),
      identical_atom(L,K), !.
inadmissible(RMC, [_|LC]) :-            % *** RMC should not contain a B-literal which is
      member([a,L], LC),                % ***  identical to an A-literal in the left cell
      member([b,K], RMC),
      identical(L,K), !.
inadmissible(_, LC) :-                  % *** LC should not contain any two literals
      pick_suc([_,L],Succ, LC),         % *** having identical atoms
      member([_,K], Succ),
      identical_atom(L,K),!.
```

```
/* ==========================================================
      Module:    GC_RULES.ARI
      Purpose:   Contains the operations and the syntactic restrictions employed by GC-
                 TP
========================================================== */

% ---------------------------------------------------------------------------------------
%          GC procedure Reduction operation

% ---------------------------------------------------------------------------------------
reduce(Chain, Reductant, [b,L], Type) :-
    extract_right_most_cell(Right_Most_Cell, Left_Cells, Chain),
    choose(Non_B, LS, RS, Left_Cells),
    not class(Non_B, b, _),
    [! select([b,L], Rest, Right_Most_Cell),
    reducible(Non_B, L, Type),
    collect_C_point(LS, [], Depth_List),
    remove_C_point(Depth_List, RS, Remove_Cpoints, New_RS),
    mega_append([Rest,LS,Remove_Cpoints,[Non_B|New_RS] ], Reductant) !].

% ---------------------------------------------------------------------------------------
%  Apply the reduction by selecting a B-literal from the rightmost cell
%  of the chain to match with a non-B-literal

% ---------------------------------------------------------------------------------------
apply_reduction(Non_B, [b,L], Right_Most_Cell, Rest, Type) :-
    select([b,L], Rest, Right_Most_Cell),
    reducible(Non_B, L, Type), !.

% ---------------------------------------------------------------------------------------
%          Remove the depths from right cells

% ---------------------------------------------------------------------------------------
remove_C_point(Depths, LC, [D|Rest_Cpoints], Left_Cells) :-
        select(D, Other_Cpoints, Depths),
        select(D, Other_LC, LC),
        remove_C_point(Other_Cpoints, Other_LC, Rest_Cpoints, Left_Cells), !.
remove_C_point(Depths, LC, [], LC).

% ---------------------------------------------------------------------------------------
%          Find if the two literals are reducible

% ---------------------------------------------------------------------------------------
reducible(Non_B, K, Type) :-
      non_B_atom(Non_B, L),
      complementary(L, K, L_A, K_A),
      match(L_A,K_A, Type).

% ---------------------------------------------------------------------------------------
%          Matching two atoms based on mode (compulsory or non-compulsory)

% ---------------------------------------------------------------------------------------
match(Atom1, Atom2, id) :-
    identical_atom(Atom1,Atom2), !.
match(Atom1, Atom2, unify) :-
    unify(Atom1, Atom2).

% ---------------------------------------------------------------------------------------
%          Collect the C-points associated with A-literals
```

```
% ----------------------------------------------------------------------------------------
collect_C_point([], L,L) :- !.
collect_C_point([ Literal|LS], Initial, Depth_List) :-
            extract_C_point(Literal, Initial, Depths),
            collect_C_point(LS, Depths, Depth_List).

% ----------------------------------------------------------------------------------------
%           Extract the C-point if the literal is an A-literal

% ----------------------------------------------------------------------------------------
extract_C_point([a,D,_], Initial, [D|Initial]) :- !.
extract_C_point( _, L,L).

% ----------------------------------------------------------------------------------------
%           GC Truncation  operation

% ----------------------------------------------------------------------------------------
truncate([ L | _ ], _, _) :-
        class(L,b,_), !, fail.
truncate(Chain, Times, Resolvant) :-
          truncate_all(Chain, Times, Resolvant).

% ----------------------------------------------------------------------------------------
%           Truncate all non-B-literals

% ----------------------------------------------------------------------------------------
truncate_all([],[], []) :- !.
truncate_all([ [b,L]|Rest ], [], [ [b,L]|Rest ]) :- !.
truncate_all([ Literal|Rest ], Times, Result) :-                    % truncate an A-literal
        remove(Literal, Rest, New_Rest, Times, Rest_Times),
        truncate_all(New_Rest, Rest_Times, Result).

% ----------------------------------------------------------------------------------------
%    Remove the A-literal and insert C-literal at its depth OR remove the C-literal.

% ----------------------------------------------------------------------------------------
remove([a,D,L], Rest, New_Rest, [0|Times], Times) :-
        choose(D, Prec, Succ, Rest),
        insert_C(L, Prec,Succ, New_Rest), !.
remove([c,L], Rest, Rest, Times, Times).

% ----------------------------------------------------------------------------------------
%    Insert a C-literal

% ----------------------------------------------------------------------------------------
insert_C(L, Prec,Succ, New_Rest) :-
      negate(L,K),
      append(Prec, [[c,K]|Succ], New_Rest).


% ----------------------------------------------------------------------------------------
%           GC Extension operation

% ----------------------------------------------------------------------------------------
extend(Chain, Resolvant, Input, unify) :-
          [! selection_function([b, Literal_L], Others, Chain),
          gen_symbol(D) !],
          resolve(Literal_L, RMC, Input, Type),
          [! append(RMC, [ [a,D,Literal_L]|Others], Right),
          append(Right, [D], Resolvant) !].
```

```
% ----------------------------------------------------------------------------------------
%           Generate the C-point symbol

% ----------------------------------------------------------------------------------------
gen_symbol(Symbol) :-
    ctr_inc(0,Ctr),
    name(Ctr,N),
    append("c_",N,List),
    name(Symbol,List).                          % counter 0 is reserved for C-ptr


% ----------------------------------------------------------------------------------------
%           Admissibility Restriction

% ----------------------------------------------------------------------------------------
syntax_check(id, Chain) :- !.
syntax_check(unify, Chain) :-
        pick_suc(Literal1, Succ_Literals, Chain),
        pick_pre(Literal2, In_Between, Succ_Literals),
        inadmissible(Literal1, Literal2, In_Between),
        !, fail.
syntax_check(_, _).


% ----------------------------------------------------------------------------------------
                        /* Check if the two literals are inadmissible
   (1) If the first literal is a B- and the second is an A-literal and
            they are identical then they are inadmissible.
        else
   (2) if the two literals are non-B-literals and they
            have identical atoms then they are inadmissible
        else
   (3) No tautologous literals unless an A-literal exists between them. This
        implies that no tautologous input chain should be used.
        (First preadmissibility restriction of the ME procedure)   */

% ----------------------------------------------------------------------------------------
inadmissible([b,K], Non_B, _) :-
        non_B_atom(Non_B, L),
        literal_atom(K, Sign, A1),
        literal_atom(L, Sign, A2),
        identical_atom(A1, A2), !.
inadmissible(Literal1, Literal2, _) :-
        non_B_atom(Literal1, A1),
        non_B_atom(Literal2, A2),
        identical_atom(A1, A2), !.
inadmissible([b,K], [b,L], In_Between) :-
        tautology(L,K),
        all_B_(In_Between).

% ----------------------------------------------------------------------------------------
% Extract the literal atom (with sign) of a non-B-literal

% ----------------------------------------------------------------------------------------
non_B_atom(L, A) :-
      class(L,C, A),
      C \== b.

% ----------------------------------------------------------------------------------------
%           Check if the list contain all B-literals
```

```
% ----------------------------------------------------------------------------------------
all_B_([]) :- !.
all_B_([[b,_]|Rest]) :-
        all_B_(Rest).
```

```
/* ============================================================
      Module:    SET_MAN.ARI
      Purpose:   Load the set of clauses and contain the procedures used in self-
                 configuration facility of ME-TP, SL-TP and GC-TP.
============================================================ */

% ---------------------------------------------------------------------------------------
%   Compile the file by asserting the set of clauses and
%   apply the self configuration facility

% ---------------------------------------------------------------------------------------
compile(File) :-
      ds(DS),
      [-File], !,
      check_format,
      assertz(clause_file(File) ),
      factor_clauses(DS),
      find_configuration,
      generate_matrix_chains(DS), %  modify this for SL-resolution
      tautology_elimination,
      pure_literal_elimination,
      abolish(a_clause/1),
      expunge.        % equivalent to purging the deleted fact permanently
compile(File) :-
   print(1,[nl,$***This file is not available...$,nl,$***Try another one...$]).

% ---------------------------------------------------------------------------------------
%  Factor the set of clauses if the derivation strategy used is SL-TP

% ---------------------------------------------------------------------------------------
factor_clauses(sl) :-
      a_clause(Clause),
      generate_factors(Clause),
      fail.
factor_clauses(sl) :-
      abolish(a_clause/1),
      retract(factored(Clause)),
      store_fact(a,a_clause(Clause)),  % Store factors as input clause
      fail.
factor_clauses(_).

% ---------------------------------------------------------------------------------------
%   Generate factors of the list

% ---------------------------------------------------------------------------------------
generate_factors(List) :-
        factor(List, Factor),
        store_as_factor(Factor),
        fail.
generate_factors(List).

% ---------------------------------------------------------------------------------------
%    Factor a given list

% ---------------------------------------------------------------------------------------
factor(List, Factor) :-
        select(L, Partial, List),
        member(K, Partial),
        unify(L,K),
        factor(Partial, Factor).
```

```
factor(Factor, Factor).

% ----------------------------------------------------------------------------------------------

%           Store the factor if it is not subsumed by any of the factors

% ----------------------------------------------------------------------------------------------
store_as_factor(Factor) :-
      [! length(Factor, N),
      convert_to_predicate(Factor, Term1) !],
      factored(List2),
      length(List2, N),
      convert_to_predicate(List2, Term2),
      subsumes(Term1, Term2),!.
store_as_factor(Factor) :-
      store_fact(a, factored(Factor) ).

% ----------------------------------------------------------------------------------------------
%  Convert the list into a temporary predicate with the element of the list as arguments

% ----------------------------------------------------------------------------------------------
convert_to_predicate(List, Term) :-
    get_combination(List, List2),
    Term =.. [temp|List2].

% ----------------------------------------------------------------------------------------------
%   Get a combination with backtracking allowed to extract all
%   possible combination

% ----------------------------------------------------------------------------------------------
get_combination([E], [E]) :- !.
get_combination(List, [E|Partial]) :-
    select(E, Rest, List),
    get_combination(Rest, Partial).

% ----------------------------------------------------------------------------------------------
%   Check if the asserted file is in the right format

% ----------------------------------------------------------------------------------------------
check_format :-
        a_clause(_), !.
check_format :-
        print(1,[nl,$***The consulted file is not in the right format$]),
        print(1,[nl,$  Format a_clause([Literal1,.... Literaln]). $]), !, fail.

% ----------------------------------------------------------------------------------------------
%           Find the configuration of the set of clauses

% ----------------------------------------------------------------------------------------------
find_configuration :-
          print(1,[nl,$Wait... Configuring the set of Clauses$]),
          a_clause(Clause),
          [! write($.$),
          length(Clause, N),
          min_max(N),      /* determine the minimum and maximum size of clause */
          determine_order(Clause),
          determine_type(Clause),
          determine_equal(Clause) !], fail.
find_configuration.
```

```
% ----------------------------------------------------------------------------------------
%  Determine the order of the set of clauses

% ----------------------------------------------------------------------------------------
determine_order(Clause) :-
        order(1), !.
determine_order(Clause) :-
        member(L, Clause),
        count_var(L, N),
        N > 0,
        abolish(order/1),
        asserta(order(1)), !.
determine_order(Clause) :-
        order(0), !.
determine_order(Clause) :-
        asserta(order(0)), !.


% ----------------------------------------------------------------------------------------
%   Determine the type of the set of clauses

% ----------------------------------------------------------------------------------------
determine_type(Clause) :-
        clause_type(general), !.
determine_type(Clause) :-
        select(++ L, Others, Clause),
        member(++ K, Others),
        abolish(clause_type/1),
        asserta(clause_type(general)), !.
determine_type(Clause) :-
        clause_type(horn), !.
determine_type(Clause) :-
        asserta(clause_type(horn)).

% ----------------------------------------------------------------------------------------
%   Determine if an equal literal exist

% ----------------------------------------------------------------------------------------
determine_equal(Clause) :-
        equal_exist, !.
determine_equal(Clause) :-
        member(Literal, Clause),
        literal_atom(Literal, _, equal(_,_)),
        asserta(equal_exist), !.
determine_equal(_).

% ----------------------------------------------------------------------------------------
%    Eliminate tautologous chain

% ----------------------------------------------------------------------------------------
tautology_elimination :-
          print(1,[nl,$Tautology elimination in action...$]),
          input_chain(N,Chain),
          [! select([b,L], Others, Chain),
          member([b,K], Others),
          tautology(L,K),
          retract(input_chain(N,Chain)),
          print(1,[nl,$***Input chain $,Chain,$ is a tautology.$]) !],
          fail.
tautology_elimination.
```

```
% ----------------------------------------------------------------------------------------
%   Remove a chain which contain a pure literal

% ----------------------------------------------------------------------------------------
pure_literal_elimination :-
        print(1,[nl,$Pure literal elimination in action...$]),
        input_chain(N,Chain),
        [!
        has_pure_literal(Chain),
        retract(input_chain(N,Chain)),
        print(1,[nl,$***Input chain $,Chain,$ has a pure literal.$])
        !], fail.
pure_literal_elimination.

% ----------------------------------------------------------------------------------------
%  Determine if the chain is resolvable

% ----------------------------------------------------------------------------------------
has_pure_literal(Chain) :-
        all_resolvable(Chain), !, fail.
has_pure_literal(Chain).



% ----------------------------------------------------------------------------------------
          Update the fact

% ----------------------------------------------------------------------------------------
update_fact(Fact1, Fact2) :-
        retract(Fact1),
        store_fact(a, Fact2), !.
update_fact( _ , Fact2) :-
        store_fact(a, Fact2).

% ----------------------------------------------------------------------------------------
%           Update the minimum and maximum size of the input clauses

% ----------------------------------------------------------------------------------------
min_max(N) :-
          clause_size(Min,Max),
          update_size(N, Min, Max), !.
min_max(N) :-
          store_fact(a, clause_size(N,N)).
update_size(N, Min, Max) :-
          N < Min,
          update_fact(clause_size(Min, Max), clause_size(N,Max) ), !.

update_size(N, Min, Max) :-
          N > Max,
          update_fact(clause_size(Min, Max), clause_size(Min,N) ), !.
update_size( _, _, _).

% ----------------------------------------------------------------------------------------
%           Concatenate a list of lists into a list

% ----------------------------------------------------------------------------------------
mega_append([], []) :- !.
mega_append([H], H) :- !.
mega_append([H|Rest], Result) :-
```

```
        mega_append(Rest, Last_Result),
        append( H, Last_Result, Result).

% ----------------------------------------------------------------------------------------
%           Generate matrix chains for each clause of the input clauses

% ----------------------------------------------------------------------------------------
generate_matrix_chains(DS) :-
        clause_size(Min, Max),
        chain_ndx_set,
        get_a_clause(Clause, Min, Max),
        write($*$),
        convert_to_chain(DS, Clause), fail.
generate_matrix_chains(_) :-
        add_reflexive_equality.

% ----------------------------------------------------------------------------------------
%           Add the reflexive axiom of equality as a unit input chain if the set

% ----------------------------------------------------------------------------------------
add_reflexive_equality :-
        equal_exist,
        store_as_chain(a, [++ equal(X,X)]), !.
add_reflexive_equality.

% ----------------------------------------------------------------------------------------
%           Convert a clause to chain depending on the derivation strategy
%           requirement.

% ----------------------------------------------------------------------------------------
convert_to_chain(_, Clause) :-
        store_as_chain(z, Clause).



% ----------------------------------------------------------------------------------------

%           Get a clause from the least to maximum number of literals

% ----------------------------------------------------------------------------------------
get_a_clause(Clause,Min, Max) :-
        a_clause(Clause),
        length(Clause, Min).
get_a_clause(Clause,Min, Max) :-
        Min < Max,
        N is Min+1,
        get_a_clause(Clause, N, Max).

% ----------------------------------------------------------------------------------------

%           Store a clause as input chain

% ----------------------------------------------------------------------------------------
store_as_chain(Pos, Clause) :-
        form_a_chain(Clause, Chain),
        get_chain_ndx(N),
        store_fact(Pos, input_chain(N,Chain)).

% ----------------------------------------------------------------------------------------
%           Form a chain
```

```
% ----------------------------------------------------------------------------------------
form_a_chain([], []) :- !.
form_a_chain([Literal|Others], [ [b,Literal]|Chain]) :-
        form_a_chain(Others, Chain).

% ----------------------------------------------------------------------------------------
%           Initialise the chain index

% ----------------------------------------------------------------------------------------
chain_ndx_set :-
      ctr_set(30,1).      % Counter 30 is set as chain index counter

% ----------------------------------------------------------------------------------------
%           Get the current chain index and increment it

% ----------------------------------------------------------------------------------------
get_chain_ndx(N) :-
      ctr_inc(30,N).
```

```
/* ==========================================================
      Module:     INTERPRE.ARI
      Purpose:    Serves as user command interpreter. This used by all the implemented
                  theorem provers
========================================================== */

% ---------------------------------------------------------------------------------------
%           Display the theorem prover Introduction

% ---------------------------------------------------------------------------------------
introduction :-
        display_screen('intro.scr'),
        cls,
        abolish([output_device/1, sos/1, derivation_/0, refutation_/0, statistics_/0]),
        asserta(match_check),
        asserta(occurs_check_),
        asserta(sos(-)),
        asserta(refutation_),
        asserta(statistics_),
        asserta(output_device(1)).

% ---------------------------------------------------------------------------------------
%           Display the help options

% ---------------------------------------------------------------------------------------
display_help :-
        cls,
        display_screen('help.scr').

% ---------------------------------------------------------------------------------------
%           Display the screen file to the screen

% ---------------------------------------------------------------------------------------
display_screen(Scrn_File) :-
        p_open(H,Scrn_File,r),
        repeat,
           read_line(H,String),
           full_screen(String),
           close(H), !.

% ---------------------------------------------------------------------------------------
%           Display the string if it is not an assigned flag such as stop and end

% ---------------------------------------------------------------------------------------
full_screen($stop$) :-
       !, hit_key.
full_screen($end$) :-
       get0(X), cls, !.
full_screen(String) :-
       print(1,[nl,String]), fail.

% ---------------------------------------------------------------------------------------
%           Hit an ESC key

% ---------------------------------------------------------------------------------------
hit_key :-
       get0(27), !.
hit_key :-
       cls, fail.
```

```
% ----------------------------------------------------------------------------------------
%  Pause until a key is pressed

% ----------------------------------------------------------------------------------------
pause_ :-
        print(1,[nl,$Hit any key...$]),
        get0(_).



% ----------------------------------------------------------------------------------------
% Interpret the entered command

% ----------------------------------------------------------------------------------------
interpreter(stop) :- !.
interpreter(dir) :- !,
        directory('*.ari',File,_,_,date(Yr,Mm,Dd),Size),
        print(1,[nl,File,$ | $,Yr,$-$,Mm,$-$,Dd,$ | $,Size,$ bytes$]),
        fail.
interpreter(cls) :- !,
        cls, fail.
interpreter(?) :- !,
        display_help, fail.
interpreter(help) :- !,
        display_help, fail.
interpreter(check) :- !,
        abolish(match_check/0),
        asserta(match_check),
        fail.
interpreter(nocheck) :- !,
        abolish(match_check/0),
        fail.
interpreter(trace) :- !,
        abolish(derivation_/0),
        asserta(derivation_),  fail.
interpreter(notrace) :- !,
        abolish(derivation_/0), fail.
interpreter(proof) :- !,
        abolish(refutation_/0),
        asserta(refutation_), fail.
interpreter(noproof) :-!,
        abolish(refutation_/0),  fail.
interpreter(stat) :- !,
        abolish(statistics_/0),
        asserta(statistics_), fail.
interpreter(nostat) :- !,
        abolish(statistics_/0), fail.
interpreter(list) :- !,
        display_chains, fail.
interpreter(input) :- !,
        display_chains, fail.
interpreter(sos) :- !,
        display_sos, fail.
interpreter(occur) :- !,
        abolish(occurs_check_/0),
        asserta(occurs_check_), fail.
interpreter(nooccur) :- !,
        abolish(occurs_check_/0), !, fail.
interpreter(valid) :- !,
```

```
        show_valid_literals, fail.
interpreter(redundant) :- !,
        show_redundant_literals, fail.
interpreter(default) :- !,
        show_current_flags, fail.
interpreter(prove) :- !,
        prove_theorem(0),
        !, fail.


interpreter(X) :-
     name(X,List),
     extract_pred(List, Pred, Term), !,
     find_command(Pred,Term),
     fail.
interpreter(X) :-
     syntax_error__,
     fail.
syntax_error__ :-
     print(1,[nl,$Wrong syntax or unknown command <<<$,X,$>>>$]),
     fail.

% ----------------------------------------------------------------------------------------------
%  Show valid literals if any

% ----------------------------------------------------------------------------------------------
show_valid_literals :-
     not valid_literal(_),
     printf([nl,$No information of valid literals yet...$]), !.
show_valid_literals :-
     valid_literal(Atom),
     printf([nl,tab(7),Atom]),
     fail.
show_valid_literals.

% ----------------------------------------------------------------------------------------------
%  Show redundant literals if any

% ----------------------------------------------------------------------------------------------
show_redundant_literals :-
     not redundant_literal(_),
     printf([nl,$No information of redundant literals yet...$]), !.
show_redundant_literals :-
     redundant_literal(Atom),
     printf([nl,tab(7),Atom]),
     fail.
show_redundant_literals.

% ----------------------------------------------------------------------------------------------
%  Display all the indicators flag

% ----------------------------------------------------------------------------------------------
show_current_flags :-
     display_flag($trace$,derivation_),
     display_flag($proof$,refutation_),
     display_flag($check$,match_check),
     display_flag($occurs check$, occurs_check_),
     current_sos,
     current_bound.
display_flag(Text, Flag) :-
```

```
    Flag,
    print(1,[nl,$### The $,Text,$ flag is ON.$]), !.
display_flag(Text, Flag) :-
    print(1,[nl,$@@@ The $,Text,$ flag is OFF.$]), !.

% ----------------------------------------------------------------------------------------

%           Display the polarity of the support set

% ----------------------------------------------------------------------------------------
current_sos :-
    sos(Sign),
    print(1,[nl,$The selected polarity of the SOS is [ $,Sign,$ ]$]), !.
current_sos.





% ----------------------------------------------------------------------------------------

%           Display the current search bound

% ----------------------------------------------------------------------------------------
current_bound :-
    bound(N),
    print(1,[nl,$The current search bound used is $,N]), !.
current_bound :-
    print(1,[nl,$The current search bound used is the size of the top clause$]).

% ----------------------------------------------------------------------------------------
% Determine appropriate command

% ----------------------------------------------------------------------------------------
find_command(prove,N) :-
      integer(N),
      prove_theorem(N),!.
find_command(bound,N) :- !,
      integer(N),
      abolish(bound/1),
      asserta(bound(N)), !.
find_command(consult,File) :- !,
      clear_all,
      print(1,[nl,$Consult file $, File]),
      compile(File).
find_command(sos,Sign) :-
      abolish(sos/1),
      asserta(sos(Sign)), !.
find_command(cd,Path) :-
      chdir(Path), !.
find_command(cd,Path) :-
      print(1,[nl,$*** Invalid directory path $]), !.
find_command(output,File) :-
      open_device(File), !.
find_command(input,_) :- !,
      display_chains.
find_command(redundant,Atom_Image) :- !,
        add_redundant(Atom_Image).
```

```
find_command(show, Arg) :- !,
        interpreter(Arg).
find_command(show, Arg) :-
      syntax_error__, fail.

% ----------------------------------------------------------------------------------------------
%  Add a redundant atom

% ----------------------------------------------------------------------------------------------
add_redundant(Atom_Image) :-
        atom_string(Atom_Image, String),
        string_term(String, Atom), asserta(redundant_literal(Atom) ), !.
add_redundant(Atom_Image) :-
        print(1,[nl,$*** Wrong syntax of atom $,Atom_Image]).

% ----------------------------------------------------------------------------------------------
%              Extract the predicate name and the argument
%              '(' - 40   ')' - 41  '['- 91  ']' - 93  '{' - 123  '}' - 125

% ----------------------------------------------------------------------------------------------
extract_pred(List, Pred, Term) :-
      choose(40, Prec, Succ, List),
      reverse(Succ, [], Rev_List),
      pick_suc(41, Rev_Arg, Rev_List),
      reverse(Rev_Arg, [], Arg),
      name(Pred, Prec),
      name(Term, Arg), !.
extract_pred([91|List], consult, File) :-
      pick_pre(93,Arg,List),
      form_a_prolog_file(Arg,File), !.
extract_pred([123|List], output, File) :-
      pick_pre(125,Arg,List),
      name(File, Arg), !.
extract_pred(List, cd, Path) :-
      choose(32, Prec, Succ, List),
      name(Path,Succ), !.

% ----------------------------------------------------------------------------------------------
%  Form a proper arity prolog file if a file extension is not specified
%   '.' - 46   'ari' - 97,114,105

% ----------------------------------------------------------------------------------------------
form_a_prolog_file(Arg,File) :-
          not member(46,Arg),
          append(Arg, ".ari", List),
          name(File, List), !.
form_a_prolog_file(Arg,File) :-
          name(File, Arg).

% ----------------------------------------------------------------------------------------------
%  Display the chains of the set of support

% ----------------------------------------------------------------------------------------------
display_sos :-
        not clause_file(_),
        print(1,[nl,$*** There is nothing to display...$]), !.
display_sos :-
        print(1,[nl,$>>>> List of clauses in the Set of Support$]),
        set_of_support(Chain),
        input_chain(N,Chain),
```

```
        print(1,[nl,$[$,N,$] $,Chain]),
        fail.
display_sos.

% ----------------------------------------------------------------------------------------
% Get the initial search bound

% ----------------------------------------------------------------------------------------
get_search_bound(_, Bound) :-
        bound(Bound), !.
get_search_bound(Chain, Bound) :-
        length(Chain,Bound).

% ----------------------------------------------------------------------------------------
% Prove the theorem

% ----------------------------------------------------------------------------------------
prove_theorem(N) :-
        not clause_file(_),
        print(1,[nl,$*** There is nothing to prove...$]), !.
prove_theorem(N) :-
        init_search_space,
        record_event(derivation_start),
        [! obtain_(N, Chain, Bound),
        search(Chain, Bound,Status),
        record_event(derivation_end),
        event_duration(derivation),
        Status == true,
        print(1,[nl,$Goal : $,Chain]),
        pause_ !],
        display_statistics(Chain), !.
prove_theorem(0) :-
        print(1,[nl,$*** The theorem is unsatisfiable...$]), !.
prove_theorem(N) :-
        print(1,[nl,$*** The chosen top chain is unrefutable...$]), !.

% ----------------------------------------------------------------------------------------
% Initialise the derivation search space

% ----------------------------------------------------------------------------------------
init_search_space :-
        abolish(event/2),
        abolish(exceed/1),
        abolish(op_ctr/2),
        abolish(path/1),
        abolish(err/3),
        store_fact(a, path([])).

% ----------------------------------------------------------------------------------------
%               Obtain the top clause and the search bound

% ----------------------------------------------------------------------------------------
obtain_(N, Chain, Bound) :-
        N > 0,
        input_chain(N, Chain),
        get_search_bound(Chain, Bound), !.
obtain_(0, Chain, Bound) :-
        set_of_support(Chain),
        get_search_bound(Chain, Bound).
```

```
/* ==========================================================
     MODULE:      SUPPORT.ARI
     PURPOSE:     Contains utility procedures used in ME-TP, SL-TP, GC-TP,
                  SLM-TP, and SLM5-TP.

========================================================== */

% ---------------------------------------------------------------------------------------

%          Append two lists into one list
% ---------------------------------------------------------------------------------------

append([], L,L) :- !.

append(L1, L2, L3) :-

      insert(L1,L11, L3,L33),

      append(L11,L2,L33).
% ---------------------------------------------------------------------------------------

%          Insert elements of the first list to the second list or vice versa
% ---------------------------------------------------------------------------------------

insert([H1,H2,H3,H4,H5,H6,H7,H8|T], T, [H1,H2,H3,H4,H5,H6,H7,H8|R],R) :- !.

insert([H1,H2,H3,H4|T], T, [H1,H2,H3,H4|R],R) :- !.

insert([H1,H2|T], T, [H1,H2|R],R) :- !.

insert([H1|T], T, [H1|R],R).
% ---------------------------------------------------------------------------------------

%          Find if the element is member of the list
% ---------------------------------------------------------------------------------------

member(E, [E|_], 1).

member(E,[_|Rest],N) :-

      member(E,Rest,N1),

      N is N1+1.

member(E,List) :-

      member(E,List, _).
% ---------------------------------------------------------------------------------------

%          Recursive selection of element from a list  starting from the first element
% ---------------------------------------------------------------------------------------

select(X, T, [X|T]).

select(X, [Y|T], [Y|R]) :-

      select(X,T,R).
% ---------------------------------------------------------------------------------------

Select starting from the last element of the list
```

```
% ----------------------------------------------------------------------------------------------
select_last(E, [X|Prev], Succ, [X|List]) :-
        select_last(E, Prev, Succ, List).
select_last(E, [], Succ, [E|Succ]).
% ----------------------------------------------------------------------------------------------
%               Pick an element and return the other succeeding elements
% ----------------------------------------------------------------------------------------------
pick_suc(E,Succ, [E|Succ]).
pick_suc(E,Succ, [X|Rest]) :-
        pick_suc(E,Succ, Rest).
% ----------------------------------------------------------------------------------------------
%  Pick an element and return the other preceding elements
% ----------------------------------------------------------------------------------------------
pick_pre(E,[], [E|Succ]).
pick_pre(E,[X|Prec], [X|Rest]) :-
        pick_pre(E,Prec, Rest).
% ----------------------------------------------------------------------------------------------
%               Reverse a list
% ----------------------------------------------------------------------------------------------
reverse([], Result, Result) :- !.
reverse([H|Rest], Initial, Result) :-
        reverse(Rest, [H|Initial], Result).
% ----------------------------------------------------------------------------------------------
%               Partition a list into two as divided by the element E
% ----------------------------------------------------------------------------------------------
choose(E, [], After, [E|After]).
choose(E, [B|Before], After,[B|List]) :-
        choose(E, Before, After, List).
% ----------------------------------------------------------------------------------------------
%               Delete the occurence of element E in the list L1
% ----------------------------------------------------------------------------------------------
delete(E, [X|L1], L1) :-
        X == E, !.
delete(E, [H|L1], [H|L2]) :-
        delete(E, L1, L2).
% ----------------------------------------------------------------------------------------------
```

```
%           Delete all the occurence of the specified element
% ----------------------------------------------------------------------------------------------
delete_all(E,List, Result) :-
        delete(E,List,Rest),
delete_all(E, Rest, Result), !.
        delete_all(E,List, List).
% ----------------------------------------------------------------------------------------------
%           Unification algorithm with occur check
%           Occur check is only done if it is sure that they are PROLOG unifiable.
%           Instantiation will only takes place if occur-check is satisfied
% ----------------------------------------------------------------------------------------------
unify(L,K) :-
        nonvar(L),
        nonvar(K),
        equal_unify(L,K), !. % ------- Try equal symmetry axiom
unify(L,K) :-   % The occurs check is disabled
        not occurs_check_,
        !, L = K.
unify(L, K) :-          % variable vs variable
        var(L),
        var(K),
        L=K, !.
unify(L, K) :-          % atomic vs atomic
        atomic(L),
        atomic(K), !,
        L=K, !.
unify(L, K) :-
        var(L), !,
        occur_check(L,K), !.
unify(L, K) :-
        var(K), !,
        occur_check(K,L), !.
```

```
unify(L, K) :-
        L =.. [Functor|Terms1],
        K =.. [Functor|Terms2],
        unify_list(Terms1,Terms2).
% ----------------------------------------------------------------------------------------
%               Unify each corresponding element of the list
% ----------------------------------------------------------------------------------------

unify_list([], []) :- !.

unify_list([E1|Rest1], [E2|Rest2]) :-
        unify(E1, E2),
        unify_list(Rest1, Rest2).

% ----------------------------------------------------------------------------------------
%               Check the occurence of the variable on the other term
% ----------------------------------------------------------------------------------------

occur_check(L,K) :-
        atom_to_list(K, K_List),
        member(K2, K_List), L == K2, !, fail.

occur_check(L,L).
% ----------------------------------------------------------------------------------------
%               Convert atom to list
% ----------------------------------------------------------------------------------------

atom_to_list(K, [K]) :-
        var(K), !.

atom_to_list(K, [K]) :-
        atomic(K), !.

atom_to_list(K,[P|Args_List]) :-
        K =.. [P|Args],
        flatten_list(Args, Args_List).
% ----------------------------------------------------------------------------------------
%               Flatten a list of atom to list
% ----------------------------------------------------------------------------------------

flatten_list([], []) :- !.

flatten_list([A|Args], Args_List) :-
        atom_to_list(A, List),
        flatten_list(Args, Args_List2),
```

```
        append(List, Args_List2, Args_List).
% ----------------------------------------------------------------------------------------

%    Unification based on the equality symmetry axiom
% ----------------------------------------------------------------------------------------

equal_unify(Equal_L, Equal_K) :-

        literal_atom(Equal_L, Sign, L),

        literal_atom(Equal_K, Sign, K),

        equal_unify(L,K), !.

equal_unify(equal(LS1,RS1), equal(LS2,RS2)) :-

        unify(LS1,RS2),

        unify(RS1,LS2).
% ----------------------------------------------------------------------------------------

%            Count the number of variables in an atom
% ----------------------------------------------------------------------------------------

count_var(Atom, 1) :- var(Atom), !.

count_var(Atom, 0) :-

        atomic(Atom), !.

count_var(Atom, N) :-

        Atom =.. [_|Args],

        count_var_list(Args, N).
% ----------------------------------------------------------------------------------------
%            Count the number of variables in the list
% ----------------------------------------------------------------------------------------

count_var_list([], 0) :- !.

count_var_list([E|Args], N) :-

        count_var(E, M),

        count_var_list(Args, K),

        N is M+K.
% ----------------------------------------------------------------------------------------

%            Check if General subsumes Specific
% ----------------------------------------------------------------------------------------

subsumes(General,Specific) :-

        disagree_pairs(General,Specific, Gen_List, Spec_List),

        is_general(Gen_List, Spec_List).
% ----------------------------------------------------------------------------------------
```

```
%           Determine the disagreeing pairs of the two unifiable atoms
%           at the lowest level.
% ----------------------------------------------------------------------------------------------
disagree_pairs(General, Specific, [],[]) :-
        General == Specific, !.                 % if they are identical (variable or atomic)
disagree_pairs(General, Specific, _,_) :-
        atomic(General),
        atomic(Specific), !, fail.
disagree_pairs(General,Specific, [General],[Specific]) :-
        var(General),!.
disagree_pairs(General,Specific, Gen_List, Spec_List) :-
        nonvar(General),
        nonvar(Specific),
        General =.. [P|Terms_G],
        Specific =.. [P|Terms_S],
        disagree_list(Terms_G, Terms_S, Gen_List, Spec_List).
% ----------------------------------------------------------------------------------------------
%           Extract the corresponding terms which disagree
% ----------------------------------------------------------------------------------------------
disagree_list(Terms_G, Terms_S, [], []) :-
        Terms_G == Terms_S, !.
disagree_list([General|Terms_G], [Specific|Terms_S], Gen_List, Spec_List) :-
        disagree_pairs(General,Specific, Sub_Gen_List, Sub_Spec_List),
        disagree_list(Terms_G, Terms_S, Accu_Gen, Accu_Spec),
        append(Sub_Gen_List, Accu_Gen, Gen_List),
        append(Sub_Spec_List, Accu_Spec, Spec_List).
% ----------------------------------------------------------------------------------------------
%   Check if the first list is more general than the second list by checking if the #
%       of distinct variables in the first is greater than the second, otherwise,
%   check if one of the variables in the first list when substituted by its corresponding
%       non-variable term in the second list, the rest of the first list is still general than the
%       second (This is assuming that in the extraction of the two lists, there is no
```

```
%       instance that  a non-variable element in the first list is paired
%       with a variable term in the second list)
% ----------------------------------------------------------------------------------------------
is_general(Gen_List, Spec_List) :-
        unify_var_nonvar(Gen_List, Spec_List),
        subsume_list(Gen_List, Spec_List).
% ----------------------------------------------------------------------------------------------
%   Check if the first list subsume the second list by checking if there is no pair
%       where the first element is subsumed by the second element, OR
%   checking if the number of distinct variables in the first is not less than the second list
% ----------------------------------------------------------------------------------------------
subsume_list(Gen_List, Spec_List) :-
        more_general(Gen_List, Spec_List),
        count_distinct_var(Spec_List, N),
        rename_vars(Gen_List, Spec_List, [], Gen_Vars),
        count_distinct_var(Gen_Vars,M),
        !, M >= N.
% ----------------------------------------------------------------------------------------------
%   Rename/substitute the general variables by the specific variables
% ----------------------------------------------------------------------------------------------
:rename_vars(Gen_List, Spec_List, Result, Result) :-
        Gen_List == Spec_List, !.
rename_vars([General|Gen_List], [Specific|Spec_List], Initial, Gen_Vars) :-
        check_var(General, Specific, Initial, Result),
        rename_vars(Gen_List, Spec_List, Result, Gen_Vars).
% ----------------------------------------------------------------------------------------------
%           Check if the general variable was already substituted before
% ----------------------------------------------------------------------------------------------
check_var(General, Specific, Initial, [General|Initial]) :-
        var(General),
        var(Specific),
        not exact_element(General, Initial),
        General = Specific, !.
check_var(General, Specific, Initial, Initial) :-
```

```
        var(General),
        var(Specific),
        exact_element(General, Initial),                    % for p(X,X,Y) vs p(A,B,B)
        General = Specific, !.
check_var(_, _, Initial, Initial).
% ---------------------------------------------------------------------------------------
%           Count the number of distinct variables in the list
% ---------------------------------------------------------------------------------------
count_distinct_var(List, N) :-
        distinct_var(List, [], Var_List),
        length(Var_List, N).
% ---------------------------------------------------------------------------------------
%   Extract distinct variables from the list
% ---------------------------------------------------------------------------------------
distinct_var([E|List], Initial, Var_List) :-
        unique_var(E, Initial, Result),
distinct_var(List, Result, Var_List), !.
        distinct_var(_, Result, Result).
% ---------------------------------------------------------------------------------------
%           Find if E is a variable and distinct from the rest
% ---------------------------------------------------------------------------------------
unique_var(E, Initial, [E|Initial]) :-
        var(E),
        not exact_element(E,Initial), !.
        unique_var(_, I,I).
% ---------------------------------------------------------------------------------------
%           An element is identical to one of the elements in the list
% ---------------------------------------------------------------------------------------
exact_element(E,[F|_]) :-
        E == F, !.
exact_element(E,[_|List]) :-
        exact_element(E, List).
% ---------------------------------------------------------------------------------------
%   Check if the first list is more general than the second list
% ---------------------------------------------------------------------------------------
```

```
more_general(Gen_List, Spec_List) :-
        Gen_List == Spec_List, !.
more_general([G|Gen_List], [S|Spec_List]) :-
        subsume_test(G,S),
        more_general(Gen_List, Spec_List).
% ----------------------------------------------------------------------------
%               Test if two terms subsume each othert
% ----------------------------------------------------------------------------

subsume_test(G,S) :-
        nonvar(G),
        var(S), !, fail.
subsume_test(G,S) :-
        not unify(G,S),
        !, fail.
subsume_test(G,S).
% ----------------------------------------------------------------------------

%  Unify a variable of the general list with a non-variable element in
%  the second list
% ----------------------------------------------------------------------------

unify_var_nonvar(Gen_List, Spec_List) :-
        Gen_List == Spec_List, !.
unify_var_nonvar([General|Gen_list], [Specific|Spec_List]) :-
        unify_subsume(General, Specific),
        unify_var_nonvar(Gen_list, Spec_List).
% ----------------------------------------------------------------------------

%               Unify only if the first subsumes the second
% ----------------------------------------------------------------------------

unify_subsume(General, Specific) :-
        var(General),
        nonvar(Specific),
        unify(General,Specific), !.
unify_subsume(General, Specific).
% ----------------------------------------------------------------------------

%               Read an atomic data from the keyboard
% ----------------------------------------------------------------------------
```

```
ratom(Text) :-
        get_string([], List),
        trailing_space(List, List1),
        reverse(List1, [], List2),
        trailing_space(List2, Net),
        name(Text,Net).
% ----------------------------------------------------------------------------------------------
%   Remove any preceding spaces
% ----------------------------------------------------------------------------------------------
trailing_space([32|Rest], Result) :-
        trailing_space(Rest, Result), !.
trailing_space(Result, Result).
% ----------------------------------------------------------------------------------------------
%               Read a series of characters
% ----------------------------------------------------------------------------------------------
get_string(Initial,Result) :-
        get0(X),
        X \== 13,
        valid_char(X,Initial, Update),
        get_string(Update,Result), !.
get_string(Result, Result).
% ----------------------------------------------------------------------------------------------
%               Check if valid character
% ----------------------------------------------------------------------------------------------
valid_char(8, [], []) :-
        put(7), put(32), !.
valid_char(8,[_|Result], Result) :-
        put(32),
        put(8), !.
valid_char(X,Initial, [X|Initial]) :-
        X >= 32,
        X =< 136, !.
valid_char(_, Initial, Initial) :-
```

```
        put(7),
        put(8),
        put(32),
        put(8).
% ----------------------------------------------------------------------------------------------
%           Record an event
% ----------------------------------------------------------------------------------------------
record_event(Event) :-
        get_time(Time),
        display_time(Time, Event),
        asserta(event(Event,Time)).
% ----------------------------------------------------------------------------------------------
%           Determine the duration to obtain a refutation or the whole derivation
% ----------------------------------------------------------------------------------------------
event_duration(refutation) :-
        retract(event($refutation_start$,T1)),
        retract(event($refutation_end$,T2)), !,
        compute_duration(T2,T1,$Refutation$), !.
event_duration(derivation) :-
        retract(event(derivation_start,T1)),
        retract(event(derivation_end,T2)), !,
        compute_duration(T2,T1,$Derivation$).
% ----------------------------------------------------------------------------------------------
%           Compute the duration of an event
% ----------------------------------------------------------------------------------------------
compute_duration(T2,T1,Event) :-
        time_lapse(T2,T1,Lapse),
        concat(Event,$ duration$,Message),
        display_time(Lapse, Message).
% ----------------------------------------------------------------------------------------------
%           Get the system time and display with a prompt When
% ----------------------------------------------------------------------------------------------
get_time(s_time(Hr,Min, Sec, Hundredth)) :-
        time(time(Hr,Min,Sec,Hundredth)).
% ----------------------------------------------------------------------------------------------
```

```
%           Display the time
% ----------------------------------------------------------------------------------------

display_time(s_time(Hr,Min,Sec,Hd), Prompt) :-

        printf([nl, Prompt,$ time = $,Hr,$ : $,Min,$ : $, Sec,$ : $,Hd, nl]).
% ----------------------------------------------------------------------------------------

%           Compute the time lapse Time2 - Time1.

%           Time = s_time(Hr, Mn,Sc,Ht)
% ----------------------------------------------------------------------------------------

time_lapse(Time2, Time1, Lapse) :-

        convert_to_seconds(Time2, Seconds2),

        convert_to_seconds(Time1, Seconds1),

        sec_diff(Seconds2, Seconds1, Lapse_in_Seconds),

        standard_time(Lapse_in_Seconds, Lapse).
% ----------------------------------------------------------------------------------------

%           Compute the time difference in seconds
% ----------------------------------------------------------------------------------------

sec_diff(Seconds2, Seconds1, Lapse_in_Seconds) :-

        Seconds2 < Seconds1,

        Lapse_in_Seconds is 24*60*60 + Seconds2 - Seconds1, !.

sec_diff(Seconds2, Seconds1, Lapse_in_Seconds) :-

        Lapse_in_Seconds is Seconds2 - Seconds1.
% ----------------------------------------------------------------------------------------

%           Convert the time in terms of seconds
% ----------------------------------------------------------------------------------------

convert_to_seconds(s_time(Hr, Mn, Sc, Hd), Seconds) :-

        Seconds is Hr*3600 + Mn*60 + Sc + Hd/100.


% ----------------------------------------------------------------------------------------

%           Convert from seconds to standard time format
% ----------------------------------------------------------------------------------------

standard_time(Seconds, s_time(Hr,Mn,Sc,Hd) ) :-

        Sec is integer(Seconds),

        Hr is Sec//3600,

        Mn is (Sec//60 - Hr*60),

        Sc is Sec - Mn*60 - Hr*3600,
```

```
        Hd is (Seconds - Sec)*100.
% ----------------------------------------------------------------------------------------

%               Print a list of string/values to the device specified
% ----------------------------------------------------------------------------------------

print(Handle, List) :-

        member(Item, List),

        printing(Handle,Item), fail.

print(_, _).

printing(H,nl) :-

        nl(H), !.

printing(H,tab(T)) :-

        tab(H,T), !.

printing(H,Item) :-

        write(H,Item).
% ----------------------------------------------------------------------------------------

%               Get the handler of the default output device and print the contents of the list
% ----------------------------------------------------------------------------------------

printf(List) :-

        output_device(Handle),

        print(Handle, List).
% ----------------------------------------------------------------------------------------

%               Determine if the two atoms are identical
% ----------------------------------------------------------------------------------------

identical_atom(A1,A2) :- A1 == A2, !.

identical_atom(equal(A1,A2), equal(B1,B2)) :-

        A1 == B2, A2 == B1, !.

identical_atom(L, K) :-

        literal_atom(L, _, Atom1),

        literal_atom(K, _, Atom2),

        identical_atom(Atom1, Atom2).
% ----------------------------------------------------------------------------------------

%               Extract the sign and atom of the literal
% ----------------------------------------------------------------------------------------

literal_atom(Literal, Sign, Atom) :-
```

```
        Literal =.. [Sign, Atom],
        opposite(Sign,_), !.
% ----------------------------------------------------------------------------------------------
%               Determine if the two signed atoms are complementary in sign
% ----------------------------------------------------------------------------------------------
complementary(L, K, L_A,K_A) :-
        literal_atom(L, Sign1, L_A),
        literal_atom(K, Sign2, K_A),
        opposite(Sign1, Sign2).
% ----------------------------------------------------------------------------------------------
%    Determine if the two literal are identical
% ----------------------------------------------------------------------------------------------
identical(L,K) :-
        literal_atom(L, Sign, L_A),
        literal_atom(K, Sign, K_A),
        identical_atom(L_A,K_A).
% ----------------------------------------------------------------------------------------------
%               The two literals are tautologous
% ----------------------------------------------------------------------------------------------
tautology(L, K) :-
        complementary(L, K, L_A,K_A),
        identical_atom(L_A, K_A).
% ----------------------------------------------------------------------------------------------
%               Check if two atoms are opposite
% ----------------------------------------------------------------------------------------------
opposite(L,K) :-
        negate(L,K).
negate(-- , ++ ) :- !.
negate(++ , -- ) :- !.
negate(++ A, -- A) :- !.
negate(-- A, ++ A) :- !.
negate(0,1) :- !.
negate(1,0).
% ----------------------------------------------------------------------------------------------
%               Open a device if it is not the console
```

```
% ----------------------------------------------------------------------------------------------
open_device(Device) :-
        output_device(Handle),
        abolish(output_device/1),
        Handle \== 1,
        close(Handle), fail.
open_device(console) :-
        asserta(output_device(1)), !.
open_device(Device) :-
        create(Handle,Device),
        asserta(output_device(Handle)), !.
% ----------------------------------------------------------------------------------------------
%               Store a fact at the bottom or at the top
% ----------------------------------------------------------------------------------------------
store_fact(z, Fact) :-
        assertz(Fact ), !.
store_fact(a, Fact) :-
        asserta(Fact ).
% ----------------------------------------------------------------------------------------------
%               Delete a fact from the database without backtracking
% ----------------------------------------------------------------------------------------------
delete_fact(Fact) :-
        retract(Fact), !.
```

```
/* ==========================================================
      Module :    DRIVER.ARI
      Purpose :   Search control of the derivation for the ME-TP, SL-TP and GC-TP.

========================================================== */

/* ----------------------------------------------------------------------------------------
      Control the search for a refutation using the consecutively bounded depth-first
      search strategy. The bound is the number of A- and B-literals in the center chain.

---------------------------------------------------------------------------------------- */

search(Goal, Limit, true) :-

      increment_bound(Limit,Bound),

      abolish(op_ctr/2),  % erase the operation counter of the previous search tree

      abolish(err/3),      % erase the error counter of the previous search tree

      ctr_set(0,0),        % set the GC C-point counter to 0

      refute(Goal, start, Bound), !.

search(Goal, _, fail) :-  % --- Failure caused by unrefutable goal ---

      printf([nl,$**** Unrefutable Goal : $, Goal, $****$,nl]).

% ----------------------------------------------------------------------------------------

%     Increment the search bound by the minimum excess of the previous search bound

% ----------------------------------------------------------------------------------------

increment_bound(Bound,Bound) :-

      new_search(Bound),

      reset_refutation_start.

increment_bound(Limit,Bound) :-

      delete_fact(exceed(Increment)),

      New_Limit is Limit + Increment,

      print(1,[nl,$New Limit $, New_Limit,nl]),

      increment_bound(New_Limit,Bound).

% ----------------------------------------------------------------------------------------

%           Update the search bound

% ----------------------------------------------------------------------------------------

new_search(Bound) :-

      abolish(search_bound/1),
```

```
        store_fact(a,search_bound(Bound)), !.
% ----------------------------------------------------------------------------------------
%               Reset the start time of refutation
% ----------------------------------------------------------------------------------------
reset_refutation_start :-
        delete_fact(event($refutation_start$, _)),
        record_event($refutation_start$), !.
        reset_refutation_start :-
        record_event($refutation_start$).
% ----------------------------------------------------------------------------------------
%               Refute a center chain
% ----------------------------------------------------------------------------------------
refute([], _, _) :-
        display_success,!.
refute(Chain, Prev_Operation, Depth) :-
        infer(Chain, Resolvant, Prev_Operation, Operation, Side_C, Type),
        assess_depth(Operation, Resolvant),
        [! inc(Depth, New_Depth),
        update_ctr(Operation,1),
        admissibility_check(Type, Operation, Resolvant),
        disp_center_chain(Resolvant, Operation, New_Depth, Side_C, derivation_) !],
        refute(Resolvant, Operation, New_Depth),
        disp_center_chain(Resolvant, Operation, New_Depth, Side_C, refutation_),
        update_path([Operation]).


% ----------------------------------------------------------------------------------------
%               Check the admissibility of a center chain using the syntactic and match checks
% ----------------------------------------------------------------------------------------
admissibility_check(id, _, _) :- !.   % No need to check (no substitution of variables)
```

```
admissibility_check(Type, Operation, Resolvant) :-
        err_ctr(syn,Operation),
        syntax_check(Type, Resolvant), !,
        err_ctr(sem,Operation),
        all_have_matches(Resolvant), !.
% ----------------------------------------------------------------------------------------
%               Infer the center chain using any of the operations used
% ----------------------------------------------------------------------------------------
infer(Chain, Resolvent, _, t, Lemma, id) :-
        truncate(Chain, Lemma, Resolvent), !.
infer(Chain, Resolvent, Op, r, [B_Literal], id) :-
        is_reducible(Op),
        reduce(Chain, Resolvent, B_Literal, id), !. %  Compulsory reduction
infer(Chain, Resolvent, Op, r, [B_Literal], Type) :-
        is_reducible(Op),
        reduce(Chain, Resolvent, B_Literal, Type).
infer(Chain, Resolvent, _, x, Index, Type) :-
        extend(Chain, Resolvent, Index, Type).
% ----------------------------------------------------------------------------------------
%               SL does not allow reduction for center chain obtained by truncation
% ----------------------------------------------------------------------------------------
is_reducible(t) :-
        ds(sl), !, fail.
is_reducible(_).
% ----------------------------------------------------------------------------------------
%               Assess if the search bound is exceeded by the number of A- and B-literals
% ----------------------------------------------------------------------------------------
assess_depth(x, Derived_Chain) :-
        search_bound(Bound),
        count_AB_(Derived_Chain, 0, N),
```

```
        N > Bound,
        Excess is N - Bound,
        exceeded(Excess), !, fail.
assess_depth(_, _).
% ----------------------------------------------------------------------------------------
%               Count the number of A- and B-literals in the center chain
% ----------------------------------------------------------------------------------------
count_AB_(Chain, Initial, Result) :-
        [! pick_suc(L, Succ, Chain),
        class(L,Class, _),
        Class \== c,
        inc(Initial, Partial) !],
        count_AB_(Succ, Partial, Result), !.
count_AB_(_, Result, Result).
% ----------------------------------------------------------------------------------------
%               Record the excess if it is less than the previous excess
% ----------------------------------------------------------------------------------------
exceeded(C) :-
        exceed(P),
        P < C, !.
exceeded(C) :-
        abolish(exceed/1),
        store_fact(a, exceed(C)).

% ----------------------------------------------------------------------------------------
%               Display the center chain and the operation applied
% ----------------------------------------------------------------------------------------
disp_center_chain(Chain, Op, Depth, Side_C, Check) :-
        Check,
        operation(Op,Operation),
```

```
        printf([nl,$Inference rule applied : $,Operation]),
        disp_rule(Op,Side_C),
        printf([nl,$Derivant at depth : $,Depth,nl]),
        print_chain(Chain),
        fail.
disp_center_chain(_, _, _, _,_).
% ----------------------------------------------------------------------------------------
%               Display the input chain OR the reduced B-literal or the lemmas generated
% ----------------------------------------------------------------------------------------
disp_rule(x, Index) :-
        input_chain(Index, Chain),
        printf([nl,$Input Chain $, Chain]), !.
disp_rule(r, B_Literal) :-
        printf([nl,$Selected B-literal $, B_Literal]), !.
disp_rule(t, Lemma) :-
        disp_lemma(Lemma).
% ----------------------------------------------------------------------------------------
%               Display the lemma
% ----------------------------------------------------------------------------------------
disp_lemma(Lemma) :-
        ds(me),
        printf([nl,$Lemma $]),
        member(X, Lemma),
        input_chain(X, Chain),
        printf([nl,$    $, Chain]),
        fail.
disp_lemma(_) :-
        printf([nl]).
% ----------------------------------------------------------------------------------------
%               Display the elements of the center chain
```

```
% ----------------------------------------------------------------------------------------
print_chain([ Lit|Chain]) :-
        print_chain(Chain),
        printf([$ $,Lit]), !.
print_chain(_).
% ----------------------------------------------------------------------------------------
%               Select an input chain from the set of support
% ----------------------------------------------------------------------------------------
set_of_support(Chain) :-
        sos(Sign),          %+++ Obtain the assigned polarity of the support set
        input_chain(_, Chain),
        same_sign(Chain, Sign).
% ----------------------------------------------------------------------------------------
%               Check if all the sign of the literal is the same to polarity of the support set
% ----------------------------------------------------------------------------------------
same_sign([], Polarity) :- !.
same_sign([B|Chain], Polarity) :-
        class(B,b,Literal),
        literal_atom(Literal,Sign,_),
        polarity(Sign, Polarity),
        same_sign(Chain, Polarity).

% ----------------------------------------------------------------------------------------
%               Compare the sign of the literal to the polarity of the support set
% ----------------------------------------------------------------------------------------
polarity(++, +) :- !.
polarity(--, -).
% ----------------------------------------------------------------------------------------
%               Display the success message and record also the end time of refutation
% ----------------------------------------------------------------------------------------
```

```
display_success :-
        record_event($refutation_end$),
        printf([nl,$SUCCESSFULL REFUTATION$,nl]).
% ----------------------------------------------------------------------------------------
%               Update the operation counter
% ----------------------------------------------------------------------------------------
update_ctr(Op, Val) :-
        retract(op_ctr(Op, Ctr)),
        N is Ctr+Val,
        store_fact(a, op_ctr(Op, N)), !.
update_ctr(Op, Val) :-
        store_fact(a, op_ctr(Op, Val)).
% ----------------------------------------------------------------------------------------
%               Clear all the derivation predicates from the database prior to the start of the %
                derivation
% ----------------------------------------------------------------------------------------
clear_all :-
        abolish([exceed/1, op_ctr/2, path/1,err/3, redundant_literal/1, valid_literal/1,
                input_chain/2, order/1, clause_type/1, equal_exist/0, clause_file/1,
                search_bound/1]),
        expunge,
        store_fact(a, path([])).
% ----------------------------------------------------------------------------------------
%               Display the input chains
% ----------------------------------------------------------------------------------------
display_chains :-
        not clause_file(_),
        print(1,[nl,$*** There is nothing to display...$]), !.
display_chains :-
        clause_file(Source),
```

```
        printf([nl,$Theorem source file : <<< $,Source,$ >>>$]),
        printf([nl,$The matrix chains with the generated lemma$,nl]),
        input_chain(X, Chain),
        printf([nl,$[$,X,$] $]),
        print_input_chain(Chain),
        fail.
display_chains :-
        order(L),
        clause_type(Type),
        clause_size(Min,Max),
        member([L,Order], [[0,$propositional$],[1,$first order$]]),
        printf([nl,nl,$The problem is a set of $,Type,$ clauses$]),
        printf([nl,$Written in $,Order,$ logic.$]),
        printf([nl,$Minimum clause size : $,Min]),
        printf([nl,$Maximum clause size : $,Max]), !.
display_chains.


% ----------------------------------------------------------------------------------------
%               Display the literal of the input chain
% ----------------------------------------------------------------------------------------
print_input_chain([]) :- !.
print_input_chain([[b,L]|Rest]) :-
        printf([$ $,L]),
        print_input_chain(Rest).
% ----------------------------------------------------------------------------------------
%               Display the statistics of the derivation
% ----------------------------------------------------------------------------------------
```

```
display_statistics(_) :-
        not statistics_, !.
display_statistics(Goal) :-
        search_bound(Bound),
        printf([nl,$Top clause : $,Goal,nl,$Derivation Search bound : $,Bound]),
        refutation_stat,
        search_tree_stat,
        error_stat.
% ----------------------------------------------------------------------------------------
%               Display the number of inadmissible center chains
% ----------------------------------------------------------------------------------------
error_stat :-
        gather_err(syn,Result1),
        gather_err(sem,Result2),
        printf([nl,nl,$   Failed by Restrictions Statistics :$]),
        disp_err($syntactic check$,Result1),
        disp_err($extendable check$,Result2).
% ----------------------------------------------------------------------------------------
%               Gather the asserted operation counter which inadmissible center chains was
%               inferred
% ----------------------------------------------------------------------------------------
gather_err(Err,[[Op,Ctr]|Rest]) :-
        retract(err(Err,Op, Ctr)),
        gather_err(Err,Rest), !.
gather_err(_,[]).
% ----------------------------------------------------------------------------------------
%               Display the check where inadmissible center chains were detected
% ----------------------------------------------------------------------------------------
disp_err( _, []) :- !.
disp_err( Type, List) :-
```

```
        printf([nl,$On $,Type,$ restrictions$]),
        display_ops(List, 0).
% ----------------------------------------------------------------------------------------
%               Display the first refutation statistics
% ----------------------------------------------------------------------------------------
refutation_stat :-
        event_duration(refutation),
        path(Path),
        count(Path, Results),
        printf([nl,$Refutation path : $,Path,nl,$Refutation Statistics :$]),
        display_ops(Results, 0).
% ----------------------------------------------------------------------------------------
%               Display the search tree statistics
% ----------------------------------------------------------------------------------------
search_tree_stat :-
        printf([nl,nl,$Search tree Statistics : $] ),
        gather_ops([x,r,t], Result),
        display_ops(Result, 0).
% ----------------------------------------------------------------------------------------
%               Gather all the applied operations that constitute the search tree
% ----------------------------------------------------------------------------------------
gather_ops([], []) :- !.
gather_ops([O|Tail], [[O,N]|Rest]) :-
        op_ctr(O,N),
        gather_ops(Tail, Rest), !.
gather_ops([O|Tail], Rest) :-
        gather_ops(Tail, Rest).
% ----------------------------------------------------------------------------------------
%               Count the number of occurence of each distinct element of the list
% ----------------------------------------------------------------------------------------
```

```
count([], []) :- !.
count([H|Rest], [[H,F]|Result]) :-
        length(Rest, M),
        delete_all(H,Rest,Others),
        length(Others, N),
        F is M-N+1,
        count(Others,Result).
% ----------------------------------------------------------------------------------------
%               Display the statistics of each operation of the list
% ----------------------------------------------------------------------------------------
display_ops([[Op,C]|Rest], N) :-
        T is N+C,
        disp_op(Op,C),
        display_ops(Rest, T), !.
display_ops(_,Total) :-
        printf([nl, $Total No. of Inference Steps : $, Total]).
% ----------------------------------------------------------------------------------------
%               Display the number of times an operation is applied
% ----------------------------------------------------------------------------------------
disp_op(Op, C) :-
        operation(Op,Operation),
        printf([nl,$  No. of $,Operation,$ = $,C]), !.
% ----------------------------------------------------------------------------------------
%               Identify the operation code
% ----------------------------------------------------------------------------------------
operation(x,$extension$):- !.
operation(r,$reduction$):- !.
operation(t,$truncation$).
% ----------------------------------------------------------------------------------------
%       Update the operation collector which collect the operation applied in the search path
```

```
% ----------------------------------------------------------------------------------------
update_path(Times) :-
        delete_fact(path(Current)),
        append(Times, Current, New),
        store_fact(a, path(New)).
update_path(Times) :-
        delete_fact(path(Current)),
        append(Times, New, Current),
        store_fact(a, path(New)), !, fail.
% ----------------------------------------------------------------------------------------
%               Count the error detected
% ----------------------------------------------------------------------------------------
err_ctr(_,_).


err_ctr(Type, Op):-
        get_prev(Type, Op, Ctr),
        N is Ctr+1,
        store_fact(a, err(Type, Op, N)), !, fail.
% ----------------------------------------------------------------------------------------
%               Get the current error type counter
% ----------------------------------------------------------------------------------------
get_prev(Type, Op, Ctr) :-
delete_fact(err(Type, Op, Ctr)), !.
get_prev(Type, Op, 0).
% ----------------------------------------------------------------------------------------
%               Resolve on the literal either by
%               (i) a unit input chain (subsumed unit extension)
%               (ii) any input chain, or
%               (iii) paramodulate
```

```
% ----------------------------------------------------------------------------------------
resolve(Literal_L, [], Input, id) :-
        unit_subsume(Literal_L, Input), !.
resolve(Literal_L, Input, N, unify) :-
        binary_resolution(Literal_L, Input, N).
resolve(Literal_L, Input, N, unify) :-
        equal_exist,
        paramodulate(Literal_L, Input, N).
% ----------------------------------------------------------------------------------------
%               Apply a binary resolution to resolve the selected literal
% ----------------------------------------------------------------------------------------
binary_resolution(Literal_L, Input, N) :-
        input_chain(N, Input_Chain),
        select( [b,Literal_K], Input, Input_Chain),
        right_match(Literal_L, Literal_K).
% ----------------------------------------------------------------------------------------
%               Apply paramodulation to the Literal.
% ----------------------------------------------------------------------------------------
paramodulate(Literal, [[b,New_Literal]|Rest_Input], Index) :-
        extract_predicate_symbol(Literal, Symbol, Terms),
        find_equal_chain(LS, RS, Rest_Input, Index),
        substitute(LS, RS, Terms, New_Terms),
        extract_predicate_symbol(New_Literal, Symbol, New_Terms).
% ----------------------------------------------------------------------------------------
%               Find an input chain which contain a positive equal literal. It has
%               to be checked if it did not pick up the reflexive equality axiom.
% ----------------------------------------------------------------------------------------
find_equal_chain(LS, RS, Rest_Input, Index) :-
        input_chain(Index, Chain),
        select([b, ++ equal(LS, RS)], Rest_Input, Chain),
```

```
        LS \== RS.   % make sure that it is not the reflexive axiom
% ----------------------------------------------------------------------------------------
%               Substitute a term which is unifiable to any of the terms of the
%               equal literal LS and RS.
% ----------------------------------------------------------------------------------------
substitute(LS, RS, [E|Terms], [New_E|Terms]) :-
        nonvar(E),          % Do not paramodulate an into variable
        find_unifiable(LS, RS, E, New_E).
substitute(LS, RS, [E|Terms], [E|New_Terms]) :-
        substitute(LS, RS, Terms, New_Terms).
% ----------------------------------------------------------------------------------------
%               Determine which term of the equal literal is unifiable with the
%               given term E of the paramodulated literal
% ----------------------------------------------------------------------------------------
find_unifiable(LS, RS, E, New_E) :-
        replace(LS, RS, E, New_E), !.
find_unifiable(LS, RS, E, New_E) :-
        E =.. [Pred|Terms],
        substitute(LS,RS, Terms, New_Terms),
        New_E =.. [Pred|New_Terms].
% ----------------------------------------------------------------------------------------
%               Replace the term with one of the terms of the equal literal
% ----------------------------------------------------------------------------------------
replace(LS, RS, E, RS) :-
        identical_atom(LS,E), !.
replace(LS, RS, E, LS) :-
        identical_atom(RS,E), !.
replace(LS, RS, E, RS) :-
        var(RS),
        unify(LS,E), !.
```

```
replace(LS, RS, E, RS) :-
        unify(LS,E).
replace(LS, RS, E, LS) :-
        unify(RS,E).
% ----------------------------------------------------------------------------------------
%               Extract the sign, predicate symbol, arity and term of a literal
% ----------------------------------------------------------------------------------------
extract_predicate_symbol(Literal, symbol(Sign, Predicate, Arity), Terms) :-
        literal_atom(Literal, Sign, Atom),
        Atom =.. [Predicate|Terms],
        length(Terms, Arity).

% ----------------------------------------------------------------------------------------
%               Find a unit input chain which is subsumed by the given literal.
% ----------------------------------------------------------------------------------------
unit_subsume(Literal_L, Index ) :-
        input_chain(Index, [ [b,Literal_K] ]),
        complementary(Literal_L, Literal_K, L_A, K_A),
        subsumes(K_A, L_A), !.
% ----------------------------------------------------------------------------------------
%               Match the two literals
% ----------------------------------------------------------------------------------------
right_match( Literal_L,  Literal_K) :-
        complementary(Literal_L, Literal_K, A1,A2),
        unify(A1,A2), !.
% ----------------------------------------------------------------------------------------
%               Selection function based on the computed weight of literal
% ----------------------------------------------------------------------------------------
selection_function(Literal, Left_Cells, Chain) :-
        get_rightmost_cell(Right_Cell, LeftCell, Chain),
```

```
        choose_literal(Literal, Right_Cell, LeftCell, Left_Cells), !.
% ---------------------------------------------------------------------------------------
%               Choose a literal to resolve upon
% ---------------------------------------------------------------------------------------
choose_literal(Literal, [Literal], LeftCell, LeftCell) :- !.
choose_literal(Literal, Right_Cell, LeftCell, Left_Cells) :-
        compute_weights(Right_Cell, Weights, LeftCell),
        select_literal(Literal, Rest, Right_Cell, Weights),
        append(Rest, LeftCell, Left_Cells).


% ---------------------------------------------------------------------------------------
%               Select a literal which has the minimum weights
% ---------------------------------------------------------------------------------------
select_literal(Literal, Rest, Right_Cell, Weights) :-
        minimum(Val, Weights),
        member(Val, Weights, Pos),
        nth_element(Pos, Literal, Rest, Right_Cell), !.
% ---------------------------------------------------------------------------------------
%               Get an element at the given position returning the rest of the list
% ---------------------------------------------------------------------------------------
nth_element(1, E, Rest, [E|Rest]) :- !.
nth_element(Pos, E, [X|Rest], [X|Tail]) :-
        dec(Pos,Next),
        nth_element(Next, E, Rest, Tail).
% ---------------------------------------------------------------------------------------
%               Find the minimum value of the list
% ---------------------------------------------------------------------------------------
minimum(Val, List) :-
        pick_suc(N, Succ, List),
```

```
        minimum(N, Val, Succ), !.

minimum(Initial, Val, List) :-
        pick_suc(N, Succ, List),
        N < Initial,
        minimum(N, Val, List), !.
minimum(Val, Val, _).
% ----------------------------------------------------------------------------------------
%               Compute the weight of each candidate literal literal
% ----------------------------------------------------------------------------------------
compute_weights([], [], _) :- !.
compute_weights([ [b,L] |Rest], [Co|Others], LC) :-
        match_count(L,M),
        has_identical([b,L], I, LC),
        Co is I + M,
        compute_weights(Rest, Others, LC).
% ----------------------------------------------------------------------------------------
%               Determine the weight of the literal by accumulating the size of the
%               input chain in every possible extension
% ----------------------------------------------------------------------------------------
match_count(L,_) :-
        ctr_set(16,0),   % Initialise the accumulator
        ctr_set(17,0),   % Initialise the unit input flag
        resolve(L, Input, _, _),
        [! length(Input, N),
        sum_weight(N) !], fail.
match_count(L,Weight) :-
        literal_atom(L, _, Atom),
        count_var(Atom,Var),
        ctr_is(16,W),
```

```
        Weight is W+Var.
% ----------------------------------------------------------------------------------------
%               Accumulate the weight of the literal. If the side chain is a unit
%               then do not accumulate for the first time and set the unit flag to 1
% ----------------------------------------------------------------------------------------
sum_weight(0) :-
        ctr_is(17,0),
        ctr_set(17,1),!.

sum_weight(N) :-
        ctr_is(16,Current),
        Weight is Current+N+1,
        ctr_set(16,Weight).
% ----------------------------------------------------------------------------------------
%               Find if the literal is preceded by an identical literal
% ----------------------------------------------------------------------------------------
has_identical(L, 0, LC) :-
        member(K, LC),
        L == K, !.
has_identical(L, 1, LC).
% ----------------------------------------------------------------------------------------
%               Get the rightmost cell if it has, otherwise the entire chain.
% ----------------------------------------------------------------------------------------
get_rightmost_cell( _, _, [[S|_]|_]) :-
        S \== b, !, fail.       /* ----- first literal is non-B-literal ----- */
get_rightmost_cell(Right_Cell, [[S|T]|LeftCell], Chain) :-
        choose([S|T], Right_Cell, LeftCell, Chain),
        S \== b, !.
get_rightmost_cell(Chain, [], Chain).
% ----------------------------------------------------------------------------------------
```

```
%       Match check.  Check if one of the B-literal of the chain cannot be extended upon
% ----------------------------------------------------------------------------------------
all_have_matches(_) :-
        not match_check, !.
all_have_matches(_) :-
        order(0), !.
all_have_matches(Chain) :-
        member([b,L], Chain),
        sufficiently_instantiated(L),
        unextendable(L), !, fail.
all_have_matches(_).
% ----------------------------------------------------------------------------------------
%               Determine if the literal is sufficiently instantiated, that is
%               half of its terms are at least not variables
% ----------------------------------------------------------------------------------------
sufficiently_instantiated(Literal) :-
        literal_atom(Literal, Sign, Atom),
        Atom =.. [_|Terms],
        length(Terms, N),
        M is (N+1)//2,
        check_var(Terms,M).
% ----------------------------------------------------------------------------------------
%               Check if the number of nonvariable terms does not exceed the allowed
% ----------------------------------------------------------------------------------------
check_var(Terms,0) :- !.
check_var(Terms,M) :-
        [! select(E,Rest, Terms),
        nonvar(E),
        dec(M,N) !],
        check_var(Rest,N).
```

```
% ----------------------------------------------------------------------------------------
%           The literal is not extendable within a series of two extensions
% ----------------------------------------------------------------------------------------
unextendable(L) :-
        literal_atom(L,_, A1),
        valid_literal(A2),
        subsumes(A1,A2), !, fail.
unextendable(L) :-
        literal_atom(L,_, L_A),
        redundant_literal(K_A),
        subsumes(K_A, L_A), !.   % K_A subsumes L_A
unextendable(L) :-
        unresolvable(L),
        literal_atom(L,_,A),
        store_fact(a, redundant_literal(A) ), !.
unextendable(L) :-
        literal_atom(L,_,A),
        store_fact(a, valid_literal(A) ), !, fail.
% ----------------------------------------------------------------------------------------
%           Check if the literal is unresolvable after the next level
% ----------------------------------------------------------------------------------------
unresolvable(L) :-
        resolve(L, Input, _,_),
        all_resolvable(Input),!, fail.
unresolvable(L).
% ----------------------------------------------------------------------------------------
%           Check if the introduced literals are resolvable
% ----------------------------------------------------------------------------------------
all_resolvable([]) :- !.
all_resolvable([[b,L]|Input]) :-
```

```
        resolve(L, _, _,_),
        all_resolvable(Input).
% ----------------------------------------------------------------------------------------
%               The literal is unresolvable
% ----------------------------------------------------------------------------------------
unresolvable(L) :-
        resolve(L, _, _,_), !, fail.
unresolvable(_).
% ----------------------------------------------------------------------------------------
%               Extract the rightmost cell of the chain
% ----------------------------------------------------------------------------------------
extract_right_most_cell([], [ L|Chain], [ L|Chain] ) :-
        not class(L,b,_), !.
extract_right_most_cell([L|Right_Most_Cell], Left_Cells, [L|Chain]) :-
        extract_right_most_cell(Right_Most_Cell, Left_Cells, Chain).
% ----------------------------------------------------------------------------------------
%               Determine the classification of the literal
% ----------------------------------------------------------------------------------------
class([C,L], C, L) :- !.
class([C,S,L],C,L) :- !.
class([C,_,_,L],C,L).
```

```
/* ==========================================================
      Module:     SLM_DRV.ARI
      Purpose:    Contains the search control for SLM and SLM-5 derivations
========================================================== */

% ---------------------------------------------------------------------------------------
%           Derivation Search Control (bounded depth-first search)

% ---------------------------------------------------------------------------------------
search(Goal_Chain, Depth, true) :-
          increment_bound(Depth,Bound),
          ctr_set(1,1),                    % This counter is reserved for node link counter
          ctr_set(2,0),                       % This counter is reserved for depth counter
          start_refutation,
          reset_ctr,
          refute([ [0,0,Goal_Chain] ], [0,0,x], 0),
          abolish(unit_subsume_fail/0), !.
search(Goal_Chain, Depth, fail).

% ---------------------------------------------------------------------------------------
%           Reset the refutation start time

% ---------------------------------------------------------------------------------------
start_refutation :-
     retract(event($refutation_start$,_)),
     record_event($refutation_start$), !.
start_refutation :-
     record_event($refutation_start$).

% ---------------------------------------------------------------------------------------
%     Increment the search bound if the failure is caused by reaching the bound limit

% ---------------------------------------------------------------------------------------
increment_bound(Depth,Depth) :-
          new_search(Depth).
increment_bound(Depth,New_Depth) :-
          find_excess(Excess),
          Depth2 is Depth + Excess,
          increment_bound(Depth2,New_Depth).

% ---------------------------------------------------------------------------------------
%  Find the excess by checking first if the failure was caused by  the
%  unit subsume check then there is no excess, otherwise, find the exceed(Excess).

% ---------------------------------------------------------------------------------------
find_excess(Excess) :-
        delete_fact(exceed(Excess)), !.
find_excess(0) :-
          delete_fact(unit_subsume_fail).

% ---------------------------------------------------------------------------------------
%           New Search bound

% ---------------------------------------------------------------------------------------
new_search(Bound) :-
            abolish(search_bound/1),
            abolish(disproved/2),
            print(1,[nl,nl,$New Search bound : $,Bound]),
            asserta(search_bound(Bound)), !.
```

```
% ----------------------------------------------------------------------------------------------
%           Check if the chain is empty
% ----------------------------------------------------------------------------------------------
empty_chain([]) :- !.
empty_chain([[0,0,[]]]).


% ----------------------------------------------------------------------------------------------
%           Find a refutation
% ----------------------------------------------------------------------------------------------
refute(Chain, _, _) :-
        empty_chain(Chain),
        display_success, !.
refute(Chain, Tip, Level) :-
        infer(Chain, Derived_Chain, Tip, New_Tip, Op, Type, Desc),
        [!
        assess_depth(Op, Derived_Chain),
        inc(Level, New_Level),
        update_ctr(Op, 1),
        apply_restrictions(Op, Type, Derived_Chain),
        disp_center_chain(Derived_Chain, New_Tip, Desc, Op, New_Level,derivation_)
        !],
        refute(Derived_Chain, New_Tip, New_Level),
        disp_center_chain(Derived_Chain, New_Tip, Desc, Op, New_Level,refutation_),
        update_path([Op]).
% ----------------------------------------------------------------------------------------------
%           Apply an inference operation to the center chain
% ----------------------------------------------------------------------------------------------
% ############ Reduction after an extension operation
infer(Chain, Derived_Chain, Tip, Tip, r, Type, B_Literal) :-
        [! untruncatable(Chain, Tip),
        after_extension(Tip, L,R),
        select_branch([[L,R,Subchain]|Branch], Other_Branches, Chain) !],
        reduce(x, [[L,R,Subchain]|Branch], Reduced_Branch, B_Literal, Type),
```

```
        append(Other_Branches, Reduced_Branch, Derived_Chain).
% ############ Reduction after a truncation operation
infer(Chain,Derived_Chain, Tip, [L,R,t_1,C_Literal], r, Type, B_Literal):-
        after_truncation(Tip, _, C_Literal),
        select_branch([[L,R,Subchain]|Branch], Other_Branches, Chain),
        reduce(C_Literal,[[L,R,Subchain]|Branch],Reduced_Branch, B_Literal, Type),
        append(Other_Branches, Reduced_Branch, Derived_Chain).
% ############ Spread if the last operation is an extension operation
infer(Chain,Derived_Chain, Tip, [L1,R1,s], s, id, []) :-
        spreadable,
        [! after_extension(Tip, L, R),
        untruncatable(Chain, Tip),
        select([L,R,Subchain], Other_Nodes, Chain) !],
        spread([L,R,Subchain], New_Nodes),
        append(Other_Nodes, New_Nodes, Derived_Chain),
        get_tip_node([L1,R1,_], Derived_Chain), !.
% ############ Truncate non-B-literals
infer(Chain,Derived_Chain,[L,R,Op|_], New_Tip, Tr, Type, []) :-
        [! Op \== s,
        select_branch([[L,R,Subchain]|Branch], Other_Branches, Chain) !],
        truncate([[L,R,Subchain]|Branch], Truncated_Branch, [Tr|C], Type),
        [! append(Truncated_Branch, Other_Branches, Derived_Chain),
        find_tip([L,R], Derived_Chain, New_Tip, [Tr|C]) !].
% ############ Apply the extension operation
infer(Chain, Extended_Chain, Old_Tip, New_Tip, x, Type,Input_Index) :-
        untruncatable(Chain, Old_Tip),
        extend(Chain, Extended_Chain, New_Tip, Type, Input_Index).


% ----------------------------------------------------------------------------------------
%     Determine if spreading can be applied. Spreading should be applied only if the
%       set of clauses is in first order level. If is not in first order and the version of SLM
%       is 5.then spreading is not applied.

% ----------------------------------------------------------------------------------------
spreadable :-
        order(1), !.
spreadable :-
        slm_version(5),
        !, fail.
spreadable.

% ----------------------------------------------------------------------------------------
%  Get a new tip

% ----------------------------------------------------------------------------------------
get_new_tip([L,R,s], Chain) :-
        select([L,R,S], Others, Chain),
        not member([R,X,_], Others), !.

% ----------------------------------------------------------------------------------------
%  Select a new tip node

% ----------------------------------------------------------------------------------------
select_node([L,R,s], [L,R,Subchain], Other_Nodes, Chain) :-
        select([L,R,Subchain], Other_Nodes, Chain), !.
select_node(_, [L,R,Subchain], Other_Nodes, Chain) :-
```

```
        select([L,R,Subchain], Other_Nodes, Chain),
        not member([R,X,_], Other_Nodes), !.

% ----------------------------------------------------------------------------------------
%  Check if the currently inferred branch is not truncatable

% ----------------------------------------------------------------------------------------
untruncatable(Chain, [L,R, s]) :- !.
untruncatable(Chain, [L,R|_]) :-
        member([L,R,[A|_]], Chain),
        member(A, [[a|_], [c|_]]), !, fail.
untruncatable( _, _).

% ----------------------------------------------------------------------------------------
%    Check if the tip was inferred by truncation

% ----------------------------------------------------------------------------------------
after_truncation([L,R,t_1,C_Literal], [L,R], C_Literal).

% ----------------------------------------------------------------------------------------
%    Check if the tip was inferred by extension

% ----------------------------------------------------------------------------------------
after_extension([L,R,x], L,R).

% ----------------------------------------------------------------------------------------
%            Select a branch from the chain

% ----------------------------------------------------------------------------------------
select_branch(Branch, Other_Branches, Chain) :-
          select([0,0,S], Other_Nodes, Chain),
          find_next_nodes([[0,0,S]], Branch, Other_Branches, Other_Nodes).

% ----------------------------------------------------------------------------------------
%            Find next nodes

% ----------------------------------------------------------------------------------------
find_next_nodes([[L,R,S]|Rest], Branch, Other_Branches, Chain) :-
          select([R,RR,SS], Other_Nodes, Chain),
          find_next_nodes([[R,RR,SS],[L,R,S]|Rest], Branch, Other_Branches, Other_Nodes).
find_next_nodes([[L,R,S]|Rest], [[L,R,S]|Rest], Chain, Chain) :-

        not member([R,_,_], Chain).
```

```
/* ==========================================================
      Module:    SLM_SUP.ARI
      Purpose:   Contains most of the utilities procedures used in SLM-TP and SLM5-
                 TP. This include the self configuration facilities, etc.
========================================================== */

% ---------------------------------------------------------------------------------------
%          Update the operation counter

% ---------------------------------------------------------------------------------------
update_ctr(Op, Val) :-
        delete_fact(op_ctr(Op, Ctr)),
        N is Ctr+Val,
        store_fact(a, op_ctr(Op, N)), !.
update_ctr(Op, Val) :-
        store_fact(a, op_ctr(Op, Val)).

% ---------------------------------------------------------------------------------------

%          Reset the operation used counter and the rejected operations counter

% ---------------------------------------------------------------------------------------
reset_ctr :-
        abolish([op_ctr/2,err/3]).

% ---------------------------------------------------------------------------------------

%          Assess the depth bound if it exceeded the search depth bound

% ---------------------------------------------------------------------------------------
assess_depth(x, Center_Chain) :-
      search_bound(Bound),
      count_A_and_B_(Center_Chain, Count),
      Count > Bound,
      New is Count - Bound,
      exceeded(New), !, fail.
assess_depth(_, _).

% ---------------------------------------------------------------------------------------
%    Count the A- and B-literals in the center chains

% ---------------------------------------------------------------------------------------
count_A_and_B_(Center_Chain, New_Level) :-
      collect_subchain(Center_Chain, [],List),
      count_AB_(List, 0, New_Level).

% ---------------------------------------------------------------------------------------
%    Count the A- and B-literals in the list of literals

% ---------------------------------------------------------------------------------------
count_AB_([], N, N) :- !.
count_AB_([E|List], Initial, Count) :-
      is_AB_(E, Initial, Partial),
      count_AB_(List, Partial, Count).

% ---------------------------------------------------------------------------------------
%    Evaluate if the element is an A- or B-literals

% ---------------------------------------------------------------------------------------
is_AB_(E, Initial, Partial) :-
```

```
        class(E, Class,_,_),
        member(Class, [a,b]),
        inc(Initial, Partial), !.
is_AB_(E, Initial, Initial).

% ---------------------------------------------------------------------------------------------
%   Collect the subchains

% ---------------------------------------------------------------------------------------------
collect_subchain([], L, L) :- !.
collect_subchain([[_,_,Sub]|Center_Chain], Initial, List) :-
            append(Sub, Initial, Partial),
            collect_subchain(Center_Chain, Partial, List).


% ---------------------------------------------------------------------------------------------

%           The search bound is exceeded. If the excess is >= to the previous

%           excess then do nothing else store the current excess

% ---------------------------------------------------------------------------------------------
exceeded(Excess) :-
        exceed(Current),
        Current < Excess, !.                        % Keep the minimum excess
exceeded(Excess) :-
        abolish(exceed/1),
        store_fact(a,exceed(Excess)), !.

% ---------------------------------------------------------------------------------------------

%           Update a fact by replacing Fact1 by Fact2 in the memory

% ---------------------------------------------------------------------------------------------
update_fact(Fact1, Fact2) :-
        delete_fact(Fact1),
        store_fact(a, Fact2), !.
update_fact(Fact1, Fact2) :-
        store_fact(a, Fact2).

% ---------------------------------------------------------------------------------------------
%    Update the refutation path which allow backtrackingif somewhere the proof failed

% ---------------------------------------------------------------------------------------------
update_path(Times) :-
        delete_fact( path(Current)),
        append(Times, Current, New),
        store_fact(a, path(New)).
update_path(Times) :-
        delete_fact( path(Current)),
        append(Times, New, Current),
        store_fact(a, path(New)), !, fail.

% ---------------------------------------------------------------------------------------------
%           Select a top chain from the set of support.
%           Choose a chain which contains literals having the same truth Index

% ---------------------------------------------------------------------------------------------
set_of_support(Chain) :-
            sos(Sign),
```

```
        model(Sign, Index),
        input_chain(_, Chain),
        same_truth_value(Index, Chain).

% ----------------------------------------------------------------------------------------
%   The sign indicates the truth index based on the trivial interpretation

% ----------------------------------------------------------------------------------------
model(+, 1) :- !.
model(-, 0).

% ----------------------------------------------------------------------------------------
%           Determine if a chain contains literals of the same truth value

% ----------------------------------------------------------------------------------------
same_truth_value(Index, []) :- !.
same_truth_value(Index, [L|Chain]) :-
        class(L,b,Index,_),
        same_truth_value(Index, Chain).

% ----------------------------------------------------------------------------------------
%Compile the file by asserting the set of clauses and apply the self configuration facility

% ----------------------------------------------------------------------------------------
compile(File) :-
      abolish(valid_literal/1),
      abolish(redundant_literal/1),
      [-File], !,
      abolish([ clause_file/1, equal_exist/2, input_chain/2, clause_size/2, clause_type/1,
                order/1]),
      assertz(clause_file(File) ),
      check_format,
      configurise_clauses,
      convert_clause_to_chain,
      tautology_elimination,
      pure_literal_elimination,
      abolish(a_clause/1),
      expunge.
compile(File) :-
        print(1,[nl,$***The file is not available in the current directory.$]),
        print(1,[nl,$>>> Try another file... $]), fail.

% ----------------------------------------------------------------------------------------
%   Check if the asserted file is in the right format

% ----------------------------------------------------------------------------------------
check_format :-
        a_clause(_), !.
check_format :-
        print(1,[nl,$***The consulted file is not in the right format$]),
        print(1,[nl,$  Format a_clause([Literal1,.... Literaln]). $]),
        !, fail.

% ----------------------------------------------------------------------------------------
%           Find the configuration of the set of clauses

% ----------------------------------------------------------------------------------------
configurise_clauses :-
        print(1,[nl,$Wait... Configuring the set of Clauses$]),
        a_clause(Clause),
```

```
        [! write($.$),
        length(Clause, N),
        min_max(N),      /* determine the minimum and maximum size of clause */
        determine_order(Clause),
        determine_type(Clause),
        determine_equal(Clause)!],
        fail.
configurise_clauses.

% ----------------------------------------------------------------------------
%  Determine the order of the set of clauses

% ----------------------------------------------------------------------------
determine_order(Clause) :-
        order(1), !.
determine_order(Clause) :-
        member(L, Clause),
        count_var(L, N),
        N > 0,
        abolish(order/1),
        asserta(order(1)), !.
determine_order(Clause) :-
        abolish(order/1),
        asserta(order(0)), !.

% ----------------------------------------------------------------------------
%   Determine the type of the set of clauses

% ----------------------------------------------------------------------------
determine_type(Clause) :-
        clause_type(general), !.
determine_type(Clause) :-
        select(++ L, Others, Clause),
        member(++ K, Others),
        abolish(clause_type/1),
        asserta(clause_type(general)), !.
determine_type(Clause) :-
        abolish(clause_type/1),
        asserta(clause_type(horn)).

% ----------------------------------------------------------------------------
%   Determine if an equal literal exist

% ----------------------------------------------------------------------------
determine_equal(Clause) :-
        equal_exist, !.
determine_equal(Clause) :-
        member(Literal, Clause),
        literal_atom(Literal, _, equal(_,_)),
        asserta(equal_exist), !.
determine_equal(_).

% ----------------------------------------------------------------------------
%    Eliminate tautologous chain

% ----------------------------------------------------------------------------
tautology_elimination :-
        print(1,[nl,$Tautology elimination in action...$]),
        input_chain(N,Chain),
        [!
```

```
        select([b,L], Others, Chain),
        member([b,K], Others),
        tautology(L,K),
        retract(input_chain(N,Chain)),
        print(1,[nl,$***Input chain $,Chain,$ is a tautology.$])
        !],
        fail.
tautology_elimination.

% ----------------------------------------------------------------------------------------
%   Remove a chain which contain a pure literal

% ----------------------------------------------------------------------------------------
pure_literal_elimination :-
        print(1,[nl,$Pure literal elimination in action...$]),
        input_chain(N,Chain),
        [!
        has_pure_literal(Chain),
        print(1,[nl,$Input chain that contains pure literal : $,nl, tab(5), Chain]),
        retract(input_chain(N,Chain))
        !],
        fail.
pure_literal_elimination.

% ----------------------------------------------------------------------------------------
%  Determine if the chain is resolvable

% ----------------------------------------------------------------------------------------
has_pure_literal(Chain) :-
        all_resolvable(Chain), !, fail.
has_pure_literal(Chain).

% ----------------------------------------------------------------------------------------

%           Convert a clause to input chain

% ----------------------------------------------------------------------------------------
convert_clause_to_chain :-
        clause_size(Min, Max),
        chain_ndx_set,
        get_a_clause(Clause, Min, Max),
        form_a_chain(Clause, Chain),
        get_chain_ndx(N),
        store_fact(z, input_chain(N, Chain)),
        fail.
convert_clause_to_chain :-
        add_reflexive_axiom.

% ----------------------------------------------------------------------------------------
%  Add the equality reflexive axiom

% ----------------------------------------------------------------------------------------
add_reflexive_axiom :-
        equal_exist,
        get_chain_ndx(N),
        store_fact(a,input_chain(N,[[b,1,++ equal(X,X)]]) ), !.
add_reflexive_axiom.

% ----------------------------------------------------------------------------------------
%           Get a clause starting from the minimum no. of literals
```

```
% ----------------------------------------------------------------------------------------
get_a_clause(Clause, N, Max) :-
        a_clause(Clause),
        length(Clause, N).
get_a_clause(Clause, N, Max) :-
        N < Max,
        inc(N,M),
        get_a_clause(Clause, M, Max).

% ----------------------------------------------------------------------------------------
%           Form a chain from a given clause

% ----------------------------------------------------------------------------------------
form_a_chain([], []) :- !.
form_a_chain([L|Rest], [[b,I,L]|Others]) :-
      interpret(L, I),
      form_a_chain(Rest, Others).

% ----------------------------------------------------------------------------------------
%           Initialise the chain index counter

% ----------------------------------------------------------------------------------------
chain_ndx_set :-
    ctr_set(30,1).

% ----------------------------------------------------------------------------------------
%           Get a chain index and update it

% ----------------------------------------------------------------------------------------
get_chain_ndx(N) :-
    ctr_inc(30,N).

% ----------------------------------------------------------------------------------------
%           Update the current minimum and maximum size of a clause in the set

% ----------------------------------------------------------------------------------------
min_max(N) :-
          clause_size(Min, Max),
          update_size(N, Min, Max), !.
min_max(N) :-
          store_fact(a, clause_size(N,N)).

% ----------------------------------------------------------------------------------------

%           Update the clause size

% ----------------------------------------------------------------------------------------
update_size(N, Min, Max) :-
            N < Min,
            update_fact(clause_size(Min,Max), clause_size(N, Max)), !.
update_size(N, Min, Max) :-
            N > Max,
            update_fact(clause_size(Min,Max), clause_size(Min, N)), !.

update_size(_, _, _).

% ----------------------------------------------------------------------------------------
%           Display successful refutation message

% ----------------------------------------------------------------------------------------
```

```
display_success :-
        record_event($refutation_end$),
        printf([nl,$SUCCESSFULL REFUTATION$,nl]), !.

% ----------------------------------------------------------------------------------------
%           Display the set of input chain and some information about the input chains

% ----------------------------------------------------------------------------------------
display_chains :-
        not input_chain(_,_),
        print(1,[nl,$No input chains yet...$]), !.
display_chains :-
        clause_file(File),
        printf([nl,$Source File ***[$,File,$]$]),
        printf([nl,$The set of input chains :$,nl]),
        input_chain(N, Chain),
        printf([nl,$[$,N,$] :$]),
        print_input_chain(Chain),
        fail.
display_chains :-
        clause_type(Type),
        order(N),
        what_order(N,Logic),
        clause_size(Min,Max),
        printf([nl,$The problem is a set of $,Type,$ clauses$]),
        printf([nl,$Written in $,Logic,$ logic.$]),
        printf([nl,$Minimum number of literals $,Min]),
        printf([nl,$Maximum number of literals $,Max]).

% ----------------------------------------------------------------------------------------

% ----------------------------------------------------------------------------------------
what_order(0,$propositional$) :- !.
what_order(1,$first order$) :- !.

% ----------------------------------------------------------------------------------------
% Display literals of input chain

% ----------------------------------------------------------------------------------------
print_input_chain([]) :- !.
print_input_chain([[b,I,L]|Chain]) :-
        printf([$ $,L]),
        print_input_chain(Chain).

% ----------------------------------------------------------------------------------------
%           Display the center chain

% ----------------------------------------------------------------------------------------
disp_center_chain(Chain, Tip, Desc, Op, Level, Check) :-
        Check,
        [! operation(Op, Operation),
        printf([nl,nl,$Inference Rule Applied : $,Operation,nl]),
        disp_rule(Desc, Op),
        printf([$-- : Center chain at Search Depth $,Level]) !],
        get_branch(Branch, Chain),
        disp_branch(Branch),
        fail.
disp_center_chain(_, _, _, _, _, _).

% ----------------------------------------------------------------------------------------
```

```
%     Display a branch of the chain

% ----------------------------------------------------------------------------------------
disp_branch(Branch) :-
          branch_to_list(Branch, List),
          separate_nodes(Nodes, Literals, List),
          printf([nl,$Branch $]),
          display_nodes(Nodes),
          display_literals(Literals), !.

% ----------------------------------------------------------------------------------------
%     Separate the nodes from the literals

% ----------------------------------------------------------------------------------------
separate_nodes([], [], []) :- !.
separate_nodes(Nodes, Literals, [E|List]) :-
          a_literal(E, Nodes, Nodes_Rest, Literals, Literals_Rest),
          separate_nodes(Nodes_Rest, Literals_Rest, List).

% ----------------------------------------------------------------------------------------
%   Is it a literal or a node

% ----------------------------------------------------------------------------------------
a_literal([L,R], [R|Nodes], Nodes, Literals, Literals) :- !.
a_literal(E, Nodes, Nodes, [E|Literals], Literals).

% ----------------------------------------------------------------------------------------
%     Display the nodes of the branch

% ----------------------------------------------------------------------------------------
display_nodes([]) :- !.
display_nodes([N|Nodes]) :-
        display_nodes(Nodes),
        printf([$->$,N]), !.

% ----------------------------------------------------------------------------------------
%     Display the literals of the branch

% ----------------------------------------------------------------------------------------
display_literals([]) :-
        printf([nl]), !.
display_literals([L|Literals]) :-
        display_literals(Literals),
        printf([$ $,L]), !.

% ----------------------------------------------------------------------------------------

%           Determine the operation code

% ----------------------------------------------------------------------------------------
operation(x,$EXTENSION$) :- !.
operation(r,$REDUCTION$) :- !.
operation(s,$SPREADING$) :- !.
operation(t_0,$TRUNCATION A(0)$) :- !.
operation(t_1,$TRUNCATION A(1)$).

% ----------------------------------------------------------------------------------------
%           Display the inference rule applied

% ----------------------------------------------------------------------------------------
```

```
disp_rule(Index, x) :-
        input_chain(Index, Chain),
        printf([$Input Chain : $,Chain,nl]), !.
disp_rule(B_Literal, r) :-
        printf([$Reducing $, B_Literal,nl]), !.
disp_rule(_, _) :- !.

% ----------------------------------------------------------------------------
%       Display a node

% ----------------------------------------------------------------------------
disp_a_node([L,R,S]) :-
        Ind is L mod 40,
        printf([nl,tab(Ind), $Node($, L,$,$, R,$):$]),
        Tab is Ind+5,
        disp_subchain(Tab, S),
        fail.
disp_a_node(_).

% ----------------------------------------------------------------------------
%       Display a subchain

% ----------------------------------------------------------------------------
disp_subchain(Tab,[]) :-
        printf([nl,tab(Tab)]), !.
disp_subchain(Tab, [Literal|Subchain]) :-
        disp_subchain(Tab, Subchain),
        printf([$ $, Literal]).

% ----------------------------------------------------------------------------
%       Clear all asserted facts and initialise the refutation path

% ----------------------------------------------------------------------------
clear_all :-
    abolish(op_ctr/2),
    abolish(node_ctr/1),
    abolish(path/1),
    abolish(err/3),
    store_fact(a, path([])),
    ctr_set(2,0).

% ----------------------------------------------------------------------------
%       Display the statistics of the derivations

% ----------------------------------------------------------------------------
display_statistics(Chain) :-
    not statistics_, !.
display_statistics(Chain) :-
    search_bound(Bound),
    printf([nl,$Goal : $,Chain]),
    printf([nl,$Derivation search bound $,Bound]),
    event_duration(refutation),
    refutation_stat,
    search_tree_stat,
    error_stat.

% ----------------------------------------------------------------------------
%       Display the statistics on errors

% ----------------------------------------------------------------------------
```

```
error_stat :-
        gather_err(syn,Result1),
        gather_err(sem,Result2),
        printf([nl,nl,$ Failed by Restrictions Statistics :$]),
        disp_err(syntactic,Result1),
        disp_err(semantic,Result2).

% ----------------------------------------------------------------------------------------------

%               Gather the rejected operation counter

% ----------------------------------------------------------------------------------------------
gather_err(Err,[[Op,Ctr]|Rest]) :-
        retract(err(Err,Op, Ctr)),
        gather_err(Err,Rest), !.
gather_err(_,[]).

% ----------------------------------------------------------------------------------------------

%               Display the number of inadmissible operations

% ----------------------------------------------------------------------------------------------
disp_err(_, []) :- !.
disp_err(Type, List) :-
        printf([nl,$On $,Type,$ restrictions$]),
        display_ops(List, 0).





% ----------------------------------------------------------------------------------------------
%               Display the first refutation statistics

% ----------------------------------------------------------------------------------------------
refutation_stat :-
        path(Path),
        count(Path, Results),
        printf([nl,$Refutation path : $,Path]),
        printf([nl,$Refutation Statistics :$,nl]),
        display_ops(Results, 0).

% ----------------------------------------------------------------------------------------------
%               Display the search tree statistics

% ----------------------------------------------------------------------------------------------
search_tree_stat :-
        printf([nl,nl, $Search tree Statistics : $]),
        gather_ops([x,s,r,c,t_0,t_1], Result),
        display_ops(Result, 0).

% ----------------------------------------------------------------------------------------------
%               Gather all the applied operations that constitute the search tree

% ----------------------------------------------------------------------------------------------
gather_ops([], []) :- !.
```

```
gather_ops([O|Tail], [[O,N]|Rest]) :-
        op_ctr(O,N),
        gather_ops(Tail, Rest), !.
gather_ops([O|Tail], Rest) :-
        gather_ops(Tail, Rest).

% ----------------------------------------------------------------------------------------
%               Count the number of occurence of each distinct element of the list

% ----------------------------------------------------------------------------------------
count([], []) :- !.
count([H|Rest], [[H,F]|Result]) :-
          length(Rest, M),
          delete_all(H,Rest,Others),
          length(Others, N),
          F is M-N+1,
          count(Others,Result).

% ----------------------------------------------------------------------------------------
%               Display the statistics of each operation of the list

% ----------------------------------------------------------------------------------------
display_ops([[Op,C]|Rest], N) :-
        T is N+C,
        disp_op(Op,C),
        display_ops(Rest, T), !.
display_ops( _,Total) :-
        printf([nl,$Total No. of Inference Steps : $, Total]).

% ----------------------------------------------------------------------------------------
%               Display the number of times an operation is applied

% ----------------------------------------------------------------------------------------
disp_op(Op, C) :-
        operation(Op, Operation),
        printf([nl,$  No. of $,Operation,$ = $,C]), !.

% ----------------------------------------------------------------------------------------
%               Extract the rightmost cell of the branch

% ----------------------------------------------------------------------------------------
extract_RMC_([], [E|List], [E|List]) :-
            not member( E, [[b,_,_], [_,_], [b,_,_,_]]), !.
extract_RMC_([E|RMC], Left_Cells, [E|List]) :-
            extract_RMC_(RMC, Left_Cells, List).


% ----------------------------------------------------------------------------------------
%               Convert a branch of nodes into a linear list (node indicators are included as a
%               paired element)

% ----------------------------------------------------------------------------------------
branch_to_list([], []) :- !.
branch_to_list([[L,R,Subchain]|Branch], List) :-
          branch_to_list(Branch, Others),
          append([[L,R]|Subchain], Others, List).

% ----------------------------------------------------------------------------------------
%               Convert a linear listed branch of nodes into a branch of nodes
```

```
% ----------------------------------------------------------------------------------------------
list_to_branch([], []) :- !.
list_to_branch([Node1|List], [Node_Subchain|Branch]) :-
            is_node(Node1),
            form_subchain(List, Subchain, Others),
            append(Node1,[Subchain], Node_Subchain),
            list_to_branch(Others, Branch).
is_node([L,R]) :-
          integer(L),
          integer(R).

% ----------------------------------------------------------------------------------------------
%           Form a subchain of a node

% ----------------------------------------------------------------------------------------------
form_subchain([], [], []) :- !.
form_subchain([E|Rest], [], [E|Rest]) :-
          is_node(E), !.
form_subchain([E|Rest], [E|Subchain], Others) :-
          form_subchain(Rest, Subchain, Others).
```

```
/* ==========================================================
        Module : SLM_RULE.ARI
        Purpose : Contains the operations used by SLM-TP.

========================================================== */

% ---------------------------------------------------------------------------
%       EXTENSION OPERATION FOR SLM

% ---------------------------------------------------------------------------
extend(Chain, Extended_Chain, [L,R,x], Type, Input_Index) :-
        selection_function(B_Literal, [L,R,LC], Other_Nodes, Chain),
        resolve(B_Literal, Input_Rest, Input_Index, Type),
        [! convert_B_A(B_Literal, A_Literal, Depth),
        append(Input_Rest, [A_Literal|LC], Extended),
        insert_depth(Depth, [L,R,Extended], Other_Nodes, Extended_Chain) !].

% ---------------------------------------------------------------------------
%       Insert the depth at the root node

% ---------------------------------------------------------------------------
insert_depth(Depth, Node, Other_Nodes, [[0,0,Root_Sub]|Rest_Nodes]) :-
      append(Other_Nodes, [Node], Chain),
      select([0,0,Root], Rest_Nodes, Chain),
      append(Root, Depth, Root_Sub), !.

% ---------------------------------------------------------------------------
% Convert a B-literal to A-literal and return the depth symbol if the literal is indexed
%  by 1, otherwise, return an empty list.

% ---------------------------------------------------------------------------
convert_B_A([b,1,L], [a,1,Symbol,L], [Symbol]) :-
      gen_sym("d_",Symbol), !.
convert_B_A([b,0,L], [a,0, L], []).

% ---------------------------------------------------------------------------
%   Spreading operation

% ---------------------------------------------------------------------------
spread([L,R,Subchain], [[L,R,True_Literals]|New_Nodes]) :-
      classify(False_Literals, True_Literals, Subchain),
      length(False_Literals, N), N > 1,
      create_nodes(R, False_Literals, New_Nodes).

% ---------------------------------------------------------------------------

%       Classify the subchain into two lists : FALSE and TRUE literals lists

% ---------------------------------------------------------------------------
classify([], [], []) :- !.
classify([], [L|Rest], [L|Rest]) :-
      non_B_literal(L), !.
classify(False_Literals, True_Literals, [L|Subchain]) :-
      assess_truth(L, False_Literals, FALSE, True_Literals, TRUE),
      classify(FALSE, TRUE, Subchain).

% ---------------------------------------------------------------------------
%  Assess the truth value of the B-literal. If it is indexed by 0 then it is added to the
%       list of false literals, otherwise, to the list of true literals
```

```
% ----------------------------------------------------------------------------------------
assess_truth([b,0,L], [[b,0,L]|False], False, True, True) :-  !.
assess_truth(L, False, False, [L|True], True).

% ----------------------------------------------------------------------------------------
%    Create new tip nodes

% ----------------------------------------------------------------------------------------
create_nodes(R, [], []) :- !.
create_nodes(R, [L|False_Literals], [[R,C,[L]]|New_Nodes]) :-
        ctr_inc(1, C),
        create_nodes(R, False_Literals, New_Nodes).

% ----------------------------------------------------------------------------------------
%                    SLM reduction
%   case (I)  : Reduction after a truncation operation
%   case (II) : Reduction after an extension operation

% ----------------------------------------------------------------------------------------
reduce(_, Branch, Reduced_Branch, B_Literal, Type) :-
    clause_type(horn), !, fail.
reduce([a,0,L], Branch, Reduced_Branch, B_Literal, Type) :-
    [! branch_to_list(Branch, List),
    choose([a,0,L], Prec, Succ, List) !],
    extract_RMC_(RMC, Left, Prec),
    apply_reduction(Left, [a,0,L], Succ, RMC, Reduced_Branch, B_Literal,Type), !.
reduce(x, Branch, Reduced_Branch, B_Literal, Type) :-
    [! branch_to_list(Branch, List),
    extract_RMC_(RMC, Left_Cells, List) !],
    remove_B(RMC, Left_Cells, Reduced_Branch, B_Literal, Type).

% ----------------------------------------------------------------------------------------
%    Remove a B-literal from the rightmost cell by reduction.  First a B-literal which
%      has identical atom with non-B-literal, otherwise, try by unification

% ----------------------------------------------------------------------------------------
remove_B(RMC, Left_Cells, Reduced_Branch, B_Literal, id) :-
    choose(Non_B, Prec, Succ, Left_Cells),
    non_B_literal(Non_B),
    apply_reduction(Prec, Non_B, Succ, RMC, Reduced_Branch, B_Literal,id), !.
remove_B(RMC, Left_Cells, Reduced_Branch, B_Literal, Type) :-
    order(1),
    choose(Non_B, Prec, Succ, Left_Cells),
    non_B_literal(Non_B),
    apply_reduction(Prec, Non_B, Succ, RMC, Reduced_Branch, B_Literal,Type).

% ----------------------------------------------------------------------------------------
%    Check if the chosen literal is not a B-literal

% ----------------------------------------------------------------------------------------
non_B_literal([b|_]) :- !, fail.
non_B_literal([_,_]) :- !, fail.
non_B_literal(Non_B).

% ----------------------------------------------------------------------------------------
%    Apply reduction using the chosen non-B-literal

% ----------------------------------------------------------------------------------------
apply_reduction(Prec, Non_B, Succ, RMC, Reduced_Branch, B_Literal,Type) :-
        select(B_Literal, Rest_RMC, RMC),
```

```
        reducible(B_Literal, Non_B, Type),
        move_depths(Prec, Non_B, Succ, New_Left_Cells),
        append(Rest_RMC,  New_Left_Cells, Result),
        list_to_branch(Result, Reduced_Branch), !.

% ---------------------------------------------------------------------------
%          Find if the two literals are reducible

% ---------------------------------------------------------------------------
reducible([b,I,L], Non_B, Type):-
      class(Non_B, a, J, K),
      opposite(J,I),
      complementary(L, K, L_A, K_A),
      match(L_A, K_A, Type).

% ---------------------------------------------------------------------------
%     Check if the two atom are unifiable/identical

% ---------------------------------------------------------------------------
match(L_A, K_A, id) :-
      identical_atom(L_A, K_A), !.
match(L_A, K_A, true) :-
      unify(L_A, K_A).


% ---------------------------------------------------------------------------
%    Check if an A-literal indexed by 1 occurs between

% ---------------------------------------------------------------------------
no_A_1(Between) :-
      member([a,1,_,K], Between), !, fail.
no_A_1(Between).

% ---------------------------------------------------------------------------
%   Move the depths of A-literals indexed by 1 and change the status flags
%   of A-literals indexed by 0 which are to the 'right' of the non-B-literal
%   used in the reduction.

% ---------------------------------------------------------------------------
move_depths(Preceeding, Non_B, Succeding, Result) :-
          collect_depths(Depths, New_Prec, Preceeding),
          remove_depths(Depths, Succeding, Common, Rest_Succ),
          append(New_Prec, Common, Right_Non_B),
          append(Right_Non_B, [Non_B|Rest_Succ], Result).

% ---------------------------------------------------------------------------
%     Delete the depths from the 'left' of the non-B-literal used in the reduction

% ---------------------------------------------------------------------------
remove_depths(Depths, Succeding, [E1|Common], Rest_Succ) :-
        select(E1, Others1, Depths),
        select(E2, Others2, Succeding),
        E1 == E2,
        remove_depths(Others1, Others2, Common, Rest_Succ), !.
remove_depths(_, Rest, [], Rest).

% ---------------------------------------------------------------------------
%  Collects all the depths associated for each A-literal indexed by 1

% ---------------------------------------------------------------------------
```

```
collect_depths([], [], []) :- !.
collect_depths(Depth, [E2|Others], [E1|Rest]) :-
        extract_depth(Depth, Added_Depth, E1, E2),
        collect_depths(Added_Depth, Others, Rest).

% ----------------------------------------------------------------------------------------
%    Extract the depth of an A-literal indexed by 1

% ----------------------------------------------------------------------------------------
extract_depth([D|Depth], Depth, [a,1,D,L], [a,1,D,L]) :- !.
extract_depth(Depth, Depth, E, E).

% ----------------------------------------------------------------------------------------
%            Truncation operation for SLM

% ----------------------------------------------------------------------------------------
truncate([Node|Branch], Truncated_Branch, Truncation, Type) :-
        truncatable(Node, Ptr, Subchain),
        [! strip_A_literal(Subchain, Subchain_Rest, Depth_Atom),
        form_node(Ptr, Subchain_Rest, Formed_Node),
        append(Formed_Node, Branch, Branch_Rest) !],
        insert(Depth_Atom, Branch_Rest, Truncated_Branch, Truncation, Type).

% ----------------------------------------------------------------------------------------
% Determine if node is truncatable and extract the Node Ptr & the Subchain of the node

% ----------------------------------------------------------------------------------------
truncatable([L,R, [E|Subchain]], [L,R], [E|Subchain]) :-
        non_B_literal(E).

% ----------------------------------------------------------------------------------------
%    Form the node pointer and the subchain into node

% ----------------------------------------------------------------------------------------
strip_A_literal([], [], []) :- !.
strip_A_literal([L|Rest], [L|Rest], []) :-
        examine(L, true), !.

strip_A_literal([L|Rest], Rest, [D,A]) :-
        examine(L, [D,A]), !.
strip_A_literal([L|Rest], Subchain, Depth_Atom) :-
        examine(L, []),
        strip_A_literal( Rest, Subchain, Depth_Atom).

% ----------------------------------------------------------------------------------------

%    Examine the literal if is a B-literal, an A-literal indexed by 0 or by 1

% ----------------------------------------------------------------------------------------
examine([b,I,A], true) :- !.
examine([a,0,A], []) :- !.
examine([a,1,D,L], [D,L]).

% ----------------------------------------------------------------------------------------
%    Form the node pointer and the subchain into node

% ----------------------------------------------------------------------------------------
form_node(_, [], []) :- !.
form_node([L,R], Subchain, [ [L,R,Subchain] ]).
```

```
% ----------------------------------------------------------------------------------------
%    Insert a C-literal if the stripped literal is an A-literal indexed by 1

% ----------------------------------------------------------------------------------------
insert([], Branch, Branch, [t_0], !) :- !.
insert([D,L], Branch, Truncated_Branch, [t_1,A_literal], true) :-
        [! convert_to_A(L,A_literal),
        branch_to_list(Branch, List),
        choose(D, Prec, Succ, List) !],
        insert_A_literal(A_literal, Prec, Inserted),
        [! append(Inserted, Succ, Result),
        list_to_branch(Result, Truncated_Branch) !].

% ----------------------------------------------------------------------------------------
%    Convert the literal to A-literal

% ----------------------------------------------------------------------------------------
convert_to_A(L, [a,0,K]) :-
        negate(L,K), !.

% ----------------------------------------------------------------------------------------
%    Insert A-literal at its depth or to a position to the right of an A-literal indexed by 0
% ----------------------------------------------------------------------
insert_A_literal(A_literal, Prec, Inserted) :-
        append(Prec, [A_literal], Inserted).
insert_A_literal(A_literal, Prec, Inserted) :-
        select_last([a,0,Literal], Prec2, Succ2, Prec),
        [! append(Prec2, [A_literal, [a,0,Literal]|Succ2], Inserted) !].
```

```
/* ==========================================================
     Module:    SLM_SEL.ARI
     Purpose:   Contains the resolve and selection function used both by SLM-TP and
                SLM5-TP

========================================================== */

% ---------------------------------------------------------------------------------------
%          Resolve upon a literal by subsumed unit chain, ELSE any input chain OR
%          paramodulate the literal if equal exist

% ---------------------------------------------------------------------------------------
resolve(L, [], N, id) :-
        unit_subsume(L, N), !.
resolve([b|L], RMC, N, true) :-
        input_chain(N, Chain),
        select([b|K], RMC, Chain),
        right_match(L, K).
resolve([b,I,L], RMC, N, true) :-
        equal_exist,
        paramodulate([I,L], RMC, N).

% ---------------------------------------------------------------------------------------
%          Apply paramodulation to the Literal.

% ---------------------------------------------------------------------------------------
paramodulate([Truth,Literal], [[b,Truth,New_Literal]|Rest_Input], Index) :-
          extract_predicate_symbol(Literal, Symbol, Terms),
          find_equal_chain(LS, RS, Rest_Input, Index),
          substitute(LS, RS, Terms, New_Terms),
          extract_predicate_symbol(New_Literal, Symbol, New_Terms).

% ---------------------------------------------------------------------------------------
%          Find an input chain which contain a positive equal literal. It has
%          to be checked if it did not pick up the reflexive equality axiom.

% ---------------------------------------------------------------------------------------
find_equal_chain(LS, RS, Rest_Input, Index) :-
          input_chain(Index, Chain),
          select([b, 1, ++ equal(LS, RS)], Rest_Input, Chain),
          LS \== RS.                    % make sure that it is not the reflexive axiom

% ---------------------------------------------------------------------------------------
%Substitute a term which is unifiable to any of the terms of the equal literal LS and RS.
% This procedure allows substitution first one term at a time until all terms are
%  substituted.

% ---------------------------------------------------------------------------------------
substitute(LS, RS, [E|Terms], [New_E|Terms]) :-
          nonvar(E),                    % Do not paramodulate an into variable
          find_unifiable(LS, RS, E, New_E).
substitute(LS, RS, [E|Terms], [E|New_Terms]) :-
          substitute(LS, RS, Terms, New_Terms).

% ---------------------------------------------------------------------------------------
%          Determine which term of the equal literal is unifiable with the given term E of
%          the paramodulated literal

% ---------------------------------------------------------------------------------------
find_unifiable(LS, RS, E, New_E) :-
```

```
            replace(LS, RS, E, New_E), !.
find_unifiable(LS, RS, E, New_E) :-
            E =.. [Pred|Terms],
            substitute(LS,RS, Terms, New_Terms),
            New_E =.. [Pred|New_Terms].

% ----------------------------------------------------------------------------------------
%           Replacing the literal term with one of the equal term

% ----------------------------------------------------------------------------------------
replace(LS, RS, E, RS) :-
            identical_atom(LS,E), !.
replace(LS, RS, E, RS) :-
            var(RS),
            unify(LS,E), !.
replace(LS, RS, E, RS) :-
            unify(LS,E).
replace(LS, RS, E, LS) :-
            unify(RS,E).

% ----------------------------------------------------------------------------------------
%           Extract the sign, predicate symbol, arity and term of a literal

% ----------------------------------------------------------------------------------------
extract_predicate_symbol(Literal, symbol(Sign, Predicate, Arity), Terms) :-
            literal_atom(Literal, Sign, Atom),
            Atom =.. [Predicate|Terms],
            length(Terms, Arity).

% ----------------------------------------------------------------------------------------
%           Find an input chain which is subsumed by L

% ----------------------------------------------------------------------------------------
unit_subsume([b,I,L], N) :-
          input_chain(N, [[b,J,K]]),
          opposite(I,J),
          complementary(L,K,L_A,K_A),
          subsumes(K_A,L_A), !.      %                    *** K_A subsumes L_A

% ----------------------------------------------------------------------------------------
%           Find out if the two literals are complementary unifiable

% ----------------------------------------------------------------------------------------
right_match([I, L], [J, K]) :-
          opposite(I,J),
          complementary(L, K, L_A, K_A),
          unify(L_A,K_A).

% ----------------------------------------------------------------------------------------
%           Generate symbol with the given prefix

% ----------------------------------------------------------------------------------------
gen_sym(Prefix, Symbol) :-
        ctr_inc(2,Current),
        name(Current, List),
        append(Prefix,List, Symbol_list),
        name(Symbol, Symbol_list).

% ----------------------------------------------------------------------------------------
%           Selection function which selects a literal wleast weight
```

```
% ----------------------------------------------------------------------------------------
selection_function(B_Literal, Tip, Remaining_Nodes, Chain) :-
        extract_all_tips(Tips, [], Other_Nodes, Chain),
        collect_RMCs(Tips, [], B_Lists),
        setof(X, member(X, B_Lists), Distinct_List),
        choose_literal(B_Literal, Tip, Distinct_List, Tips, Other_Tips, Chain),
        append(Other_Tips, Other_Nodes, Remaining_Nodes), !.

% ----------------------------------------------------------------------------------------
%   Collect all the B-literals in all the rightmost cells

% ----------------------------------------------------------------------------------------
collect_RMCs([], B_List, B_List) :- !.
collect_RMCs([[L,R,Sub]|Tips], Initial, B_List) :-
        get_RMC_(RMC, Sub),
        append(RMC, Initial, Accumulate),
        collect_RMCs(Tips, Accumulate, B_List).

% ----------------------------------------------------------------------------------------
%  Get the rightmost cell of the subchain

% ----------------------------------------------------------------------------------------
get_RMC_(RMC, Sub) :-
        pick_pre(Non_B, RMC, Sub),
        not class(Non_B,b,_,_), !.
get_RMC_(RMC, RMC).

% ----------------------------------------------------------------------------------------
%   Choose a B-literal from the candidate list.  B-literal which is identical to an A-literals
%           is removed from the candidate list.

% ----------------------------------------------------------------------------------------
choose_literal(B_Literal, Tip, B_Lists, Tips, Other_Tips, Chain):-
        strip_id_B_A(B_Lists, No_id_B_List, Chain),
        length(No_id_B_List, Length),
        Length > 0,
        select_literal(B_Literal, Tip, No_id_B_List, Tips, Other_Tips), !.
choose_literal(B_Literal, Tip, B_List, Tips, Other_Tips, Chain):-
        select_literal(B_Literal, Tip, B_List, Tips, Other_Tips).

% ----------------------------------------------------------------------------------------
%    Select a B-literal with the least weight if there are more than one B-literal to select.

% ----------------------------------------------------------------------------------------
select_literal(B_Literal, Tip, [B_Literal], Tips, Other_Tips) :-
        find_B(B_Literal, Tip, Other_Tips, Tips), !.
select_literal(B_Literal, Tip, B_List, Tips, Other_Tips) :-
        compute_weights(B_Weights, B_List),
        minimum(Val, B_Weights),
        member(Val, B_Weights, Pos),
        member(B_Literal, B_List, Pos),
        find_B(B_Literal, Tip, Other_Tips, Tips).

% ----------------------------------------------------------------------------------------
%    Do not include a B-literal, which is identical to any A-literal, for selection

% ----------------------------------------------------------------------------------------
strip_id_B_A([], [], _) :- !.
strip_id_B_A([B|Literals], B_List, Chain) :-
            find_id(B,Chain, B_List, B_Rest),
```

```
        strip_id_B_A(Literals, B_Rest, Chain).

% ----------------------------------------------------------------------------
%   Find if the B-literal is unifiable (identical) to an A-literal

% ----------------------------------------------------------------------------
find_id([b,I,B], Chain, B_List, B_List) :-
            member([L,R,Sub], Chain),
            member(A_Literal, Sub),
            class(A_Literal, a,I,A),
            identical(A,B), !.
find_id(B, Chain, [B|B_List], B_List).

% ----------------------------------------------------------------------------
%     Extract all tips from the center chain

% ----------------------------------------------------------------------------
extract_all_tips(Tips, Initial, Other_Nodes, Chain) :-
      select_tip(Node, Initial, Rest_Nodes, Chain),
      extract_all_tips(Tips, [Node|Initial], Other_Nodes, Rest_Nodes), !.
extract_all_tips(Tips, Tips, Chain, Chain).

% ----------------------------------------------------------------------------
%    Select a tip node

% ----------------------------------------------------------------------------
select_tip([L,R,Subchain], Tips, Other_Nodes, Chain) :-
      select([L,R,Subchain], Other_Nodes, Chain),
      not member([R,_,_], Other_Nodes),
      not member([R,_,_], Tips),!.

% ----------------------------------------------------------------------------
%  Find the node where the B_Literal occurs

% ----------------------------------------------------------------------------
find_B(B_Literal, [L,R,LC], Other_Tips, Tips) :-
      select([L,R,Subchain], Other_Tips, Tips),
      choose(Literal, Prec, Succ, Subchain),
      B_Literal == Literal,
      all_B_(Prec),
      append(Prec, Succ, LC), !.

% ----------------------------------------------------------------------------
%  All the elements of the list are B-literals

% ----------------------------------------------------------------------------
all_B_([]) :- !.
all_B_([L|Prec]) :-
        class(L,b,_,_),
        all_B_(Prec).

% ----------------------------------------------------------------------------
%   Find the minimum value of the list

% ----------------------------------------------------------------------------
minimum(Val, List) :-
          [! pick_suc(N, Succ, List) !],
          minimum(N, Val, Succ).
minimum(Initial, Val, List) :-
          pick_suc(N, Succ, List),
```

```
        N < Initial,
        minimum(N, Val, List), !.
minimum(Val, Val, _).

% ----------------------------------------------------------------------------------------
%   Compute the coefficient for unit input factor, number of possible

% ----------------------------------------------------------------------------------------
compute_weights([], []):- !.
compute_weights([W|Others], [ L |Rest] ) :-
        match_count(L,W),
        compute_weights(Others, Rest).

% ----------------------------------------------------------------------------------------
%   Determine the weight of the literal by accumulating the size of the
%   input chain used in every possible extension

% ----------------------------------------------------------------------------------------
match_count(L,_) :-
      ctr_set(16,0),                                  % Initialise the accumulator
      ctr_set(17,0),                                  % Initialise the unit input flag
      resolve(L, Input, _, _),
      [! length(Input, N),
      sum_weight(N) !], fail.
match_count([b,I,L],Weight) :-
      literal_atom(L, _, Atom),
      count_var(Atom,Var),
      ctr_is(16,W),
      Weight is W+Var.

% ----------------------------------------------------------------------------------------
% Accumulate the weight of the literal. If the side chain is a unit
% then do not accumulate for the first time and set the unit flag to 1

% ----------------------------------------------------------------------------------------
sum_weight(0) :-
      ctr_is(17,0),
      ctr_set(17,1),!.
sum_weight(N) :-
      ctr_is(16,Current),
      Weight is Current+N+1,

      ctr_set(16,Weight).

% ----------------------------------------------------------------------------------------
%           Apply the admissibility restrictions of SLM

% ----------------------------------------------------------------------------------------
apply_restrictions(Op, id, _) :- !.
apply_restrictions(Op, _, _) :-
        member(Op, [s,t_0]), !.
apply_restrictions(t_1, _, Chain) :-
        !, syntax_check(t_1, Chain).
apply_restrictions(Op, true, Chain) :-
        syntax_check(Op, Chain), !,
        semantic_checking(Op, Chain).

% ----------------------------------------------------------------------------------------
                        Error counter
```

```
% ----------------------------------------------------------------------------------------
err_ctr(Type, Op):-
    delete_fact(err(Type, Op, Ctr)),
    inc(Ctr,N),
    store_fact(a, err(Type, Op, N)), !.
err_ctr(Type, Op):-
    store_fact(a, err(Type, Op, 1)).

% ----------------------------------------------------------------------------------------
%    Collect all B-literals in the chain

% ----------------------------------------------------------------------------------------
collect_B([], []) :- !.
collect_B([[_,_,Sub]|Chain], List) :-
        collect_B(Chain, List2),
        extract_B(Sub, B_List),
        append(B_List, List2, List).

% ----------------------------------------------------------------------------------------

%           Extract B-literals from the center chain

% ----------------------------------------------------------------------------------------
extract_B([], []) :- !.
extract_B([L|Rest], B_List) :-
        get_only_B(L, B_List, Rest_B),
        extract_B(Rest, Rest_B).

% ----------------------------------------------------------------------------------------

%           Get only the B-literals

% ----------------------------------------------------------------------------------------
get_only_B(L, [L|B_List], B_List) :-
        class(L, b,_,_), !.
get_only_B(_, B_List, B_List).

% ----------------------------------------------------------------------------------------
%           Semantic checking. (In place of semantic check, the match check is used)

% ----------------------------------------------------------------------------------------
semantic_checking(_, Chain) :-
        not match_check, !.
semantic_checking(_, Chain) :-
        order(0), !.
semantic_checking(Op, Chain) :-
        [! collect_B(Chain, List) !],
        not all_have_matches(List), !,
        err_ctr(sem, Op), fail.
semantic_checking(_, _).

% ----------------------------------------------------------------------------------------
%                                   Match Check.
%           Check if a sufficiently instantiated B-literal can be extended upon at least up
%           to the next level

% ----------------------------------------------------------------------------------------
all_have_matches(List) :-
        exist_unextendable(List), !, fail.
all_have_matches(_).
```

```
% ----------------------------------------------------------------------------------------
%   There exist a  literal which is unextendable

% ----------------------------------------------------------------------------------------
exist_unextendable(List) :-
        member([b,I,Literal], List),
        sufficiently_instantiated(Literal),
        unresolvable([b,I,Literal]), !.

% ----------------------------------------------------------------------------------------
%  Check if the literal is sufficiently instantiated. The rule is if the number of
%   non-variable terms is half the total number of terms of the literal.

% ----------------------------------------------------------------------------------------
sufficiently_instantiated(Literal) :-
        literal_atom(Literal, _, Atom),
        Atom =.. [_|Terms],
        length(Terms, N),
        M is (N+1)//2,
        check_nonvar(Terms, M).

% ----------------------------------------------------------------------------------------
%  Check if the number of nonvariable terms is equal to the specified no.

% ----------------------------------------------------------------------------------------
check_nonvar(_, 0) :- !.
check_nonvar(Terms, M) :-
        [! select(E, Rest, Terms),
        nonvar(E),
        dec(M,N) !],
        check_nonvar(Rest,N).

% ----------------------------------------------------------------------------------------
%   Check if the list of B-literal are all resolvable

% ----------------------------------------------------------------------------------------
all_resolvable([]) :- !.
all_resolvable([B|Rest]) :-
       resolve(B,Input,_,_),
       all_resolvable(Rest).

% ----------------------------------------------------------------------------------------

%   Check if the literal is unresolvable

% ----------------------------------------------------------------------------------------
unresolvable([b,_,L]) :-                    %  Check if the atom is a valid literal atom
       literal_atom(L,_,L_A),
       valid_literal(K_A),
       subsumes(L_A,K_A), !, fail.
unresolvable([b,_,L]) :-                    %  Check if the atom is a redundant  literal atom
       literal_atom(L,_,L_A),
       redundant_literal(K_A),
       subsumes(K_A, L_A),!.
unresolvable([b,I,Literal]) :-          % Check if the literal can be possibly extended upon
       unextendable([b,I,Literal]),
       literal_atom(Literal,_,L_A),
       store_fact(a, redundant_literal(L_A)), !.
unresolvable([_,_,L]) :-
       literal_atom(L,_,L_A),
```

```
        store_fact(a, valid_literal(L_A) ), !, fail.



% ----------------------------------------------------------------------------------------
%  Check if the literal cannot be extended upon

% ----------------------------------------------------------------------------------------
unextendable(Literal) :-
        resolve(Literal,Input,_,_),
        all_resolvable(Input), !, fail.
unextendable(Literal).

% ----------------------------------------------------------------------------------------
%   Find a tip of a branch

% ----------------------------------------------------------------------------------------
find_tip([L,R], Chain, New_Tip, Truncation) :-
        member([L,R,S], Chain),
        append([L,R], Truncation, New_Tip), !.
find_tip([L,R], Chain, New_Tip, Truncation) :-
        find_new_tip(Tip, Chain),
        append(Tip, Truncation, New_Tip), !.
find_tip([L,R], [], [], Truncation).

% ----------------------------------------------------------------------------------------
%   Find a new tip from the chain

% ----------------------------------------------------------------------------------------
find_new_tip([L,R], Chain) :-
        get_tip_node([L,R,[Literal|_] ], Chain),
        not class(Literal,b,_,_), !.
find_new_tip([L,R], Chain) :-
        get_tip_node([L,R,_], Chain), !.

% ----------------------------------------------------------------------------------------
%   Get a tip node

% ----------------------------------------------------------------------------------------
get_tip_node([L,R,S], Chain) :-
        select([L,R,S], Others, Chain),
        not member([R,_,_], Others).

% ----------------------------------------------------------------------------------------
%    Get a branch only from the chain

% ----------------------------------------------------------------------------------------
get_branch(Branch, Chain) :-
            select([0,0,S], Others, Chain),
            next_node([[0,0,S]], Branch, Others).

% ----------------------------------------------------------------------------------------
%    Find the next node of the branch

% ----------------------------------------------------------------------------------------
next_node([[L,R,S]|Rest], [[L,R,S]|Rest], Chain) :-
            not member([R,_,_], Chain), !.
next_node([[L,R,S]|Rest], Branch, Chain) :-
```

```
        select([R,RR,SS], Others, Chain),
        next_node([[R,RR,SS],[L,R,S]|Rest], Branch, Others).

% ----------------------------------------------------------------------------
%           Find if the list contains only B-literal

% ----------------------------------------------------------------------------
all_B([]) :- !.
all_B([L|Between]) :-
      class(L,b,_,_),
      all_B(Between).

% ----------------------------------------------------------------------------
%           Find if the list does not contain an A-literal indexed by 1

% ----------------------------------------------------------------------------
no_A_1_([]) :- !.
no_A_1_([L|Between]) :-
      class(L,a,1,_), !, fail.
no_A_1_([_|Between]) :-
      no_A_1_(Between).

% ----------------------------------------------------------------------------
%       Reserve for possible extensions

% ----------------------------------------------------------------------------

%           Interpret the truth index of the atom

% ----------------------------------------------------------------------------
interpret(Signed_Atom, Index) :-
      trivial_I(Signed_Atom, Index).
trivial_I(++ A, 1) :- !.
trivial_I(-- A, 0).

% ----------------------------------------------------------------------------
%     Get the rightmost cell of the branch

% ----------------------------------------------------------------------------
get_RMC(RMC, List) :-
        pick_pre(L, RMC, List),
        non_B_literal(L), !.
get_RMC(RMC, RMC).
```

```
/* ============================================================
      Module:    SLM-REST.ARI
      Purpose:   Contains the procedure that implement the syntactic restrictions of
                 SLM-TP
============================================================ */

% ---------------------------------------------------------------------------------------

%          Sytanctic restrictions used by SLM-TP

% ---------------------------------------------------------------------------------------
syntax_check(Op, Chain) :-
        get_branch(Branch, Chain),
        [! branch_to_list(Branch, Literals_List) !],
        not restrictions(Literals_List),
        err_ctr(syn,Op), !,
        fail.
syntax_check(_, _).

% ---------------------------------------------------------------------------------------
%          Restrictions for SLM

% ---------------------------------------------------------------------------------------
restrictions(Literal_List) :-
       pick_suc(Right_Literal, Rest, Literal_List),
       class(Right_Literal, C1, I, L),
       pick_pre(Left_Literal, In_Between, Rest),
       class(Left_Literal, C2, J, K),
       inadmissible([C1,I,L], [C2,J,K], In_Between),!, fail.
restrictions(_).

% ---------------------------------------------------------------------------------------
             /* Find if the combination of the two literal are inadmissible
1)  No A-literal which is identical to any A-literal indexed by 0 unless
       an A-literal indexed by 1 is between them.
2)  No two tautologous A literals unless an A-literal indexed by 1 exists between them
      (to enforce compulsory reduction)
3)  No B-literal which is identical to a preceeding A-literal unless
       an A-literal indexed by 1 is between them (preemptive of 1) */

% ---------------------------------------------------------------------------------------
inadmissible([a,I,L], [a,I,K], Between) :-          % to prevent inserting a recycled
       identical(L,K),                              % A-literal to an equivalent position
       no_A_1_(Between), !.
inadmissible([a,I,L], [a,J,K], Between) :-          % to prevent extension when it
       opposite(I,J),                               % must be reduced
       tautology(L,K),
       no_A_1_(Between), !.
inadmissible([b,I,L], [a,I,K], Between) :-          % to prevent loop by not allowing
       identical(L,K),
       no_A_1_(Between).                            % identical B- and A-literals

% ---------------------------------------------------------------------------------------
%          Identify the classification, index and atom (with sign) of the literal

% ---------------------------------------------------------------------------------------
class([C,I,L], C, I, L) :- !.
class([C,I,_, L], C, I, L) :- !.
class([C,I,_,_, L], C, I, L).
```

```
/* ============================================================
      Module:     SLM5_RUL.ARI
      Purpose:    Contains the procedures of the inference operations of SLM5-TP
============================================================ */

% ---------------------------------------------------------------------------------------
%              EXTENSION OPERATION FOR SLM-5

% ---------------------------------------------------------------------------------------
extend(Chain, Extended_Chain, [L,R,x], Type, Input_Index) :-
        selection_function(B_Literal, [L,R,LC], Other_Nodes, Chain),
        resolve(B_Literal, Input_Rest, Input_Index, Type),
        [! convert_B_A(B_Literal, A_Literal, Depth),
        append(Input_Rest, [A_Literal|LC], Extended),
        insert_depth(Depth, [L,R,Extended], Other_Nodes, Extended_Chain) !].

% ---------------------------------------------------------------------------------------
%    Insert the depth at the root node

% ---------------------------------------------------------------------------------------
insert_depth(Depth, Node, Other_Nodes, [[0,0,Root_Sub]|Rest_Nodes]) :-
      append(Other_Nodes, [Node], Chain),
      select([0,0,Root], Rest_Nodes, Chain),
      append(Root, Depth, Root_Sub), !.

% ---------------------------------------------------------------------------------------
%  Convert a B-literal to A-literal and return the depth symbol if the
%    literal is indexed by 1, otherwise, return an empty list.

% ---------------------------------------------------------------------------------------
convert_B_A([b,1,L], [a,1,Symbol,L], [Symbol]) :-
      gen_sym("d_",Symbol), !.
convert_B_A([b,0,L], [a,0,1,L], []).

% ---------------------------------------------------------------------------------------
%    SPREADING OPERATION of SLM5-TP

% ---------------------------------------------------------------------------------------
spread([L,R,Subchain], [[L,R,True_Literals]|New_Nodes]) :-
      classify(False_Literals, True_Literals, Subchain),
      length(False_Literals, N), N > 1,
      create_nodes(R, False_Literals, New_Nodes).

% ---------------------------------------------------------------------------------------
%    Classify the subchain into two lists : FALSE and TRUE literals list

% ---------------------------------------------------------------------------------------
classify([], [], []) :- !.
classify([], [L|Rest], [L|Rest]) :-
      non_B_literal(L), !.
classify(False_Literals, True_Literals, [L|Subchain]) :-
      assess_truth(L, False_Literals, FALSE, True_Literals, TRUE),
      classify(FALSE, TRUE, Subchain).

% ---------------------------------------------------------------------------------------
%   Assess the truth value of the B-literal. If it is indexed by 0
%     then it is added to the list of false literals, otherwise, to the list of true literals

% ---------------------------------------------------------------------------------------
assess_truth([b,0,L], [[b,0,L]|False], False, True, True) :-  !.
```

```
assess_truth(L, False, False, [L|True], True).

% ----------------------------------------------------------------------------------------
%    Create new tip nodes

% ----------------------------------------------------------------------------------------
create_nodes(R, [], []) :- !.
create_nodes(R, [L|False_Literals], [[R,C,[L]]|New_Nodes]) :-
        ctr_inc(1, C),
        create_nodes(R, False_Literals, New_Nodes).




% ----------------------------------------------------------------------------------------
%                    REDUCTION OPERATION OF SLM5-TP
%   case (I)  : Reduction after a truncation operation
%   case (II) : Reduction after an extension operation

% ----------------------------------------------------------------------------------------
reduce(_, Branch, Reduced_Branch, B_Literal, Type) :-
     clause_type(horn), !, fail.
reduce([c,0,L], Branch, Reduced_Branch, B_Literal, Type) :-
     [! branch_to_list(Branch, List),
     choose([c,0,L], Prec, Succ, List) !],
     extract_RMC_(RMC, Left, Prec),
     apply_reduction(Left, [c,0,L], Succ, RMC, Reduced_Branch, B_Literal,Type), !.
reduce(x, Branch, Reduced_Branch, B_Literal, Type) :-
     [! branch_to_list(Branch, List),
     extract_RMC_(RMC, Left_Cells, List) !],
     remove_B(RMC, Left_Cells, Reduced_Branch, B_Literal, Type).

% ----------------------------------------------------------------------------------------
%    Remove a B-literal from the rightmost cell by reduction
% (i)  Compulsory reduction
% (ii) reduce by unification

% ----------------------------------------------------------------------------------------
remove_B(RMC, Left_Cells, Reduced_Branch, B_Literal, id) :-
     choose(Non_B, Prec, Succ, Left_Cells),
     non_B_literal(Non_B),
     apply_reduction(Prec, Non_B, Succ, RMC, Reduced_Branch, B_Literal,id), !.
remove_B(RMC, Left_Cells, Reduced_Branch, B_Literal, Type) :-
     order(1),
     choose(Non_B, Prec, Succ, Left_Cells),
     non_B_literal(Non_B),
     apply_reduction(Prec, Non_B, Succ, RMC, Reduced_Branch, B_Literal,Type).

% ----------------------------------------------------------------------------------------
%    Check if the chosen literal is not a B-literal

% ----------------------------------------------------------------------------------------
non_B_literal([b|_]) :- !, fail.
non_B_literal([_,_]) :- !, fail.
non_B_literal(Non_B).

% ----------------------------------------------------------------------------------------
%    Apply reduction using the chosen non-B-literal
```

```
% ----------------------------------------------------------------------------------------
apply_reduction(Prec, Non_B, Succ, RMC, Reduced_Branch, B_Literal,Type) :-
        select(B_Literal, Rest_RMC, RMC),
        reducible(B_Literal, Non_B, Type),
        move_depths(Prec, Non_B, Succ, New_Left_Cells),
        append(Rest_RMC,  New_Left_Cells, Result),
        list_to_branch(Result, Reduced_Branch), !.

% ----------------------------------------------------------------------------------------
%          Find if the two literals are reducible

% ----------------------------------------------------------------------------------------
reducible([b,I,L], Non_B, Type):-
      class(Non_B, C, J, K), C\==b,
      opposite(J,I),
      complementary(L, K, L_A, K_A),
      match(L_A, K_A, Type).

% ----------------------------------------------------------------------------------------
%     Check if the two atom are unifiable/identical

% ----------------------------------------------------------------------------------------
match(L_A, K_A, id) :-
      identical_atom(L_A, K_A), !.
match(L_A, K_A, true) :-
      unify(L_A, K_A).

% ----------------------------------------------------------------------------------------
%    Check if an A-literal indexed by 1 occurs between

% ----------------------------------------------------------------------------------------
no_A_1(Between) :-
      member([a,1,_,K], Between),
      !, fail.
no_A_1(Between).

% ----------------------------------------------------------------------------------------
%   Move the depths of A-literals indexed by 1 and change the status flags
%   of A-literals indexed by 0 which are to the 'right' of the non-B-literal
%   used in the reduction.

% ----------------------------------------------------------------------------------------
move_depths(Preceeding, Non_B, Succeding, Result) :-
          collect_depths(Depths, New_Prec, Preceeding),
          remove_depths(Depths, Succeding, Common, Rest_Succ),
          append(New_Prec, Common, Right_Non_B),
          append(Right_Non_B, [Non_B|Rest_Succ], Result).

% ----------------------------------------------------------------------------------------
%     Delete the depths from the 'left' of the non-B-literal used in
%     the reduction

% ----------------------------------------------------------------------------------------
remove_depths(Depths, Succeding, [E1|Common], Rest_Succ) :-
        select(E1, Others1, Depths),
        select(E2, Others2, Succeding),
        E1 == E2,
        remove_depths(Others1, Others2, Common, Rest_Succ), !.
remove_depths(_, Rest, [], Rest).
```

```
% ----------------------------------------------------------------------------------------
%           Collects all the depths associated for each A-literal indexed by 1 and at the
%           same time change the status flag of A-literals indexed by 0 to 0.

% ----------------------------------------------------------------------------------------
collect_depths([], [], []) :- !.
collect_depths(Depth, [E2|Others], [E1|Rest]) :-
        extract_depth(Depth, Added_Depth, E1, E2),
        collect_depths(Added_Depth, Others, Rest).

% ----------------------------------------------------------------------------------------
%   Extract the depth of an A-literal indexed by 1 or change the status flag of an
%           A-literal indexed by 0.

% ----------------------------------------------------------------------------------------
extract_depth([D|Depth], Depth, [a,1,D,L], [a,1,D,L]) :- !.
extract_depth(Depth, Depth, [a,0,1,L], [a,0,0,L]) :- !.
extract_depth(Depth, Depth, E, E).

% ----------------------------------------------------------------------------------------
%           TRUNCATION OPERATION for SLM5-TP

% ----------------------------------------------------------------------------------------
truncate([Node|Branch], Truncated_Branch, Truncation, Type) :-
        truncatable(Node, Ptr, Subchain),
        [! strip_A_literal(Subchain, Subchain_Rest, Depth_Atom),
        form_node(Ptr, Subchain_Rest, Formed_Node),
        append(Formed_Node, Branch, Branch_Rest) !],
        insert(Depth_Atom, Branch_Rest, Truncated_Branch, Truncation, Type).

% ----------------------------------------------------------------------------------------
%           Determine if the node is truncatable and extract the Node Ptr & the Subchain of
%           the node

% ----------------------------------------------------------------------------------------
truncatable([L,R, [E|Subchain]], [L,R], [E|Subchain]) :-
        non_B_literal(E).

% ----------------------------------------------------------------------------------------
%   Form the node pointer and the subchain into node

% ----------------------------------------------------------------------------------------
strip_A_literal([], [], []) :- !.
strip_A_literal([L|Rest], [L|Rest], []) :-
        examine(L, true), !.
strip_A_literal([L|Rest], Rest, [D,A]) :-
        examine(L, [D,A]), !.
strip_A_literal([L|Rest], Subchain, Depth_Atom) :-
        examine(L, []),
        strip_A_literal( Rest, Subchain, Depth_Atom).

% ----------------------------------------------------------------------------------------
%   Examine if the non-B-literal is a C-literal or an A-literal indexed by 0 and
%   status flag is 0 (conditionally proved literal, an A-literal indexed by 0 and
%   status flag is 1 (proved literal), or an A-literal indexed by 1
%   (get the associated depth and the atom)

% ----------------------------------------------------------------------------------------
examine([b,I,A], true) :- !.
examine([c,0,A], []) :- !.
```

```
examine([a,0,0,A], []) :- !.
examine([a,0,1,L], []) :-
        add_as_unit_chain(L), !.
examine([a,1,D,L], [D,L]).

% ---------------------------------------------------------------------------
%   Add the proved literal as a unit chain if it is not subsumed by
%   any of the unit input chain

% ---------------------------------------------------------------------------
add_as_unit_chain(L) :-
        opposite(L,K),
        add_to_set_([b,1,K]).

% ---------------------------------------------------------------------------
%   Add the B-literal as unit chain if it does not subsume a unit chain
%   or if the set is a set of general clauses

% ---------------------------------------------------------------------------
add_to_set_([b,I,Literal]) :-
        clause_type(general),
        opposite(I,J),
        opposite(Literal, K),
        not unit_subsume([b,J,K], _),
        get_chain_ndx(N),
        store_fact(a, input_chain(N, [[b,I,Literal]]) ), !.
add_to_set_(_).

% ---------------------------------------------------------------------------
%   Form the node pointer and the subchain into node

% ---------------------------------------------------------------------------
form_node(_, [], []) :- !.
form_node([L,R], Subchain, [ [L,R,Subchain] ]).

% ---------------------------------------------------------------------------
%   Insert a C-literal if the stripped literal is an A-literal indexed by 1

% ---------------------------------------------------------------------------
insert([], Branch, Branch, [t_0], !) :- !.
insert([D,L], Branch, Truncated_Branch, [t_1,C_literal], true) :-
        [! convert_to_C(L,C_literal),
        branch_to_list(Branch, List),
        choose(D, Prec, Succ, List),
        proved(C_literal, Succ, Status)!],
        insert_C_literal(Status, C_literal, Prec, Inserted),
        [! append(Inserted, Succ, Result),
        list_to_branch(Result, Truncated_Branch) !].

% ---------------------------------------------------------------------------
%   Find if the C-literal is a proved literal

% ---------------------------------------------------------------------------
proved([c,0,Literal], Succ, true) :-
        no_A_or_B_(Succ),
        add_to_set_([b,0,Literal]), !.
proved(_, _, false).

% ---------------------------------------------------------------------------
%   Determine if the list does not contain an A- or B-literals
```

```
% ----------------------------------------------------------------------------------------
no_A_or_B_(Succ) :-
        member(E, Succ),
        class(E,Class,_,_),
        member(Class,[a,b]), !, fail.
no_A_or_B_(_).

% ----------------------------------------------------------------------------------------
%   Convert the literal to C-literal

% ----------------------------------------------------------------------------------------
convert_to_C(L, [c,0,K]) :-
        opposite(L, K).

% ----------------------------------------------------------------------------------------
%   Insert the C-literal at its depth or to a position not equivalent to the position where
%   C-literal was already inserted. Two positions are said to be equivalent if the literals
%   occurring between two positions are all C-literals or A-literals indexed by 0

% ----------------------------------------------------------------------------------------
insert_C_literal(true, C_literal, Prec, Prec) :- !.
insert_C_literal(_, C_literal, Prec, Inserted) :-
        append(Prec, [C_literal], Inserted).
insert_C_literal(_, C_literal, Prec, Inserted) :-
        select_last(Literal, Prec2, Succ2, Prec),
        [! inequivalent_pos(Literal, Succ2),
        append(Prec2, [C_literal, Literal|Succ2], Inserted) !].

% ----------------------------------------------------------------------------------------
%    Check if the preceeding literal of A-literal indexed by 0 is
%    1) a B-literal   (2)  an A-literal indexed by 1 (3) a node ptr

% ----------------------------------------------------------------------------------------
inequivalent_pos([a,0,_,L], [Succeeding_Literal|Succ2]) :-
        not class(Succeeding_Literal,a,0,_).
check_prec([b|_]) :- !.
check_prec([a,1|_]) :- !.
check_prec([_,_]).
```

```
/* ============================================================
      Module:     SLM5-RES.ARI
      Purpose:    Contains the procedures that implement the syntactic restrictions of
                  SLM5-TP
============================================================ */

% ---------------------------------------------------------------------------------------

%           Syntactic Check

% ---------------------------------------------------------------------------------------
syntax_check(Op, Chain) :-
        get_branch(Branch, Chain),
        [! branch_to_list(Branch, Literals_List) !],
        not restrictions(Literals_List),
        err_ctr(syn,Op),
        !, fail.
syntax_check(x, Chain) :-          % Check if an A-literal is subsumed by a unit chain
        clause_type(general),
        member(Node, Chain),
        get_subchain(Node, Chain, Subchain),
        member(Literal, Subchain),
        class(Literal, a, I, Atom),
        unit_subsume([b, I, Atom], _),
        toggle_subsume_flag,
        err_ctr(syn,x),
        !, fail.
syntax_check(_, _).

% ---------------------------------------------------------------------------------------
%  Toggle the unit subsume flag error

% ---------------------------------------------------------------------------------------
toggle_subsume_flag :-
      unit_subsume_fail, !.
toggle_subsume_flag :-
      store_fact(a,unit_subsume_fail).

% ---------------------------------------------------------------------------------------
%    Get a subchain but remove the rightmost literal if it is a subchain of a tip node

% ---------------------------------------------------------------------------------------
get_subchain([L,R,[_|Subchain]], Chain, Subchain) :-
        not member([R,_,_], Chain), !.
get_subchain([L,R,Subchain], Chain, Subchain).

% ---------------------------------------------------------------------------------------
%           Get a pair of literals and check if they are admissible

% ---------------------------------------------------------------------------------------
restrictions(Literal_List) :-
        pick_suc(Right_Literal, Rest, Literal_List),
        class(Right_Literal, C1, I, L),
        pick_pre(Left_Literal, In_Between, Rest),
        class(Left_Literal, C2, J, K),
        inadmissible([C1,I,L], [C2,J,K], In_Between),
        !, fail.
restrictions(Literals).

% ---------------------------------------------------------------------------------------
```

```
                /*Find if the combination of the two literal are inadmissible
1)  No B- or A-literal is identical to any of the preceeding  A-literals
2a) No C-literal which is identical to any A-literal indexed by 0 unless
        an A-literal indexed by 1 is in between them
2b) No B-literal which is identical to a preceeding C-literal unless
        an A-literal indexed by 1 is in between them (preemptive of 1)
3)  No A-literal which is tautologous to a preceeding C-literal unless an
        A-literal indexed by 1 occurs between them.
4)  No two tautologous B-literals unless a non-B-literal occurs
        between them  */

% ----------------------------------------------------------------------------------------------
inadmissible([C1,I,L], [C2,I,K], _) :-
        pair(1,C1,C2),
        literal_atom(L,Sign, A1),
        literal_atom(K,Sign, A2),
        identical_atom(A1,A2), !.
inadmissible([C1,I,L], [C2,I,K], Between) :-
        pair(2,C1,C2),
        identical_atom(L,K),
        no_A_1_(Between), !.
inadmissible([C1,I,L], [C2,J,K], Between) :-
        pair(3,C1,C2),
        1 is I+J,
        tautology(L,K),
        no_A_1_(Between), !.
inadmissible([b,I,L], [b,J,K], Between) :-
        1 is I+J,
        tautology(L,K),
        all_B(Between).

% ----------------------------------------------------------------------------------------------

%           Possible pair of literals classification

% ----------------------------------------------------------------------------------------------
pair(1,b,a).
pair(1,a,a).
pair(2,b,c).
pair(2,a,c).
pair(2,c,a).
pair(3,a,a).
pair(3,a,c).
```

Appendix C

THEOREM PROVER DESCRIPTION

C.1. Running the theorem prover

Any of the five theorem provers can be run by typing the filename of the executable file at the DOS prompt. Once invoked, the program looks for the program description file (`INTRO.SCR`) and displays its contents on the screen. Pressing the `ESC` key at any point skips the display of the entire file. After the display of the program description, the theorem prover prompt (the theorem prover name followed by the symbol ':-') appears. Typing the command '?' or '`help`' displays the syntax of commands and their descriptions. This list of commands is stored in the HELP.SCR file. Each command has to be terminated by a `carriage return` key.

C.2. Loading a theorem

Each theorem to be proved should be stored into a text file and each clause of a theorem should be in the correct format. A clause should be terminated with a dot. For example, problem PEL-10 of Appendix D should be written to a file in the following format :

```
a_clause([++ r, -- q]).
a_clause([-- r, ++ p]).
a_clause([-- r, ++ q]).
a_clause([-- p, ++ q, ++ r]).
a_clause([-- p, -- q]).
a_clause([++ p, ++ q]).
```

A problem file can be loaded into the database by typing the command `[Filename]` where the `Filename` should include the extension, otherwise, the theorem prover will append the extension `'.ari'` to the specified filename before searching for the file in the current directory. If the file is not in the current directory, an error message is displayed. Typing the command `dir` displays all the files in the current directory with the extension `'.ari'`, on the screen. To transfer to another subdirectory, the command `cd PATH` should be typed at the prompt. After a successful loading of the problem file, the input clauses can be viewed by typing the command `list` or `input` at the prompt. The input clauses are displayed on the screen with their corresponding index number. The clauses in the set of support can be viewed by typing the command `show(sos)` or `sos` at the prompt. The default set of support is the set of negative clauses. The set of support can be changed to the set of positive clauses by typing the command `sos(+)`.

C.3. Starting the derivation

The derivation is started by typing the command `prove` or `prove(Index)` where `Index` is the index number of a top clause. If the command `prove` is used, the theorem prover chooses a top clause from the set of support. If the selected top clause does not produce a refutation, another clause from the set of support is selected. If desired, the initial search bound can be specified by typing the command `bound(N)` where N is an integer greater than 0. During the derivation, some of the B-literals in the center chains will be checked if they can be extended upon. Because this check may consume a lot of time, this check can be disabled by typing the command `nocheck` before starting the derivation. Typing the command `check` will make the check operational again. During the unification of two terms, an occurs check is done before the two terms are unified. The occurs check can be disabled by typing the command `nooccur`. However, it should be pointed out that disabling the occurs check may cause some serious problems, hence, it must be done with caution. Typing the command `occur` will make the occurs check operational again.

C.4. Derivation Output

During the derivation, the five theorem provers generate output and send it to the output device. As shown in Figure 29, an output device can be the console, printer or a text file. The default output device is the console. The command `{prn}` directs the output to the printer, and the command `{Filename}` to direct the output to the text file `Filename`. The command `{console}` directs the output to the console. The five theorem provers generate three types of output during and after the derivation. They are as follows :

1. Trace of the derivation. Each inference step of the derivation (excluding inadmissible inference steps) is written to the output device. This includes the center chain, the operation applied, the input chain if the operation applied is an extension, the B-literal removed by a reduction operation, the lemmas generated after truncation in the case of the ME-TP, and the current search depth. This trace can be disabled by typing the command `notrace` before the start of the derivation. The commands `trace` and `notrace` toggle on and off the trace of the derivation respectively.

2. Trace of the proof. The inference steps that lead to the refutation are written to the output device. The commands `proof` and `noproof` toggle on and off the trace of the proof respectively.

3. Statistics of the derivation. After a refutation is obtained, the statistics of the derivation are written to the output device. The statistics include: (i) the search duration, (ii) the refutation duration (the time from the start of the search for the search bound that produce the refutation was obtained until the time when the empty chain is obtained), (iii) the sequence of operations applied to obtain

the refutation, (iv) the number of each of the operations in the refutation, (v) the number of each of the operations in the successful search bound, and (vi) the number of each of the operations pruned by the syntactic restrictions and the match check. The commands `stat` and `nostat` toggle on and off the writing of the statistics to the output device respectively.

During the derivation, some information is generated and stored into the database. The `valid_literal(Atom)` facts can be viewed by typing the command `valid` and the command `redundant` displays the `redundant_literal(Atom)` facts. The command `show(default)` or `default` displays the current values of the trace derivation flag, trace proof flag, match check flag, occurs check flag, the polarity of the set of support and the initial search bound.

Appendix D

PROBLEMS USED TO TEST THE IMPLEMENTED THEOREM PROVERS

The following are problems used to test the implemented theorem provers. They are grouped according to the source from which they were taken. They are presented in clausal format. The sign '~' means negation and 'v' means OR.

D.1. Selected problems from Pelletier (1986)

Pel-10. A problem to test whether 'natural' systems correctly manipulate premises.

```
1.  R v ~Q
2.  ~R v P
3.  ~R v Q
4.  ~P v Q v R
5.  ~P v ~Q
6.  P v Q
```

Pel-12. The 'hardest' propositional problem found in Kalish and Montague (1964), according to Pelletier (1982).

1. P v Q v R
2. ~P v ~Q v R
3. ~P v Q v ~R
4. P v ~Q v ~R
5. P v ~Q v R
6. P v Q ~R
7. ~P v Q v R
8. ~P v ~Q v ~R

Pel-14. A problem not solvable by unit resolution nor by 'pure' input resolution.

1. ~P v Q
2. ~Q v P
3. ~Q v ~P
4. Q v P

Pel-17. A problem which appears not to be provable by Bledsoe et al (1972).

1. ~P v Q v S
2. ~P v ~R v S
3. P
4. ~Q v R
5. ~S

Pel-20.

```
1.  ~p(Y) v ~q(Z) v r(f(Y,Z))
2.  ~p(Y) v ~q(Z) v s(X)
3.  p(a)
4.  q(b)
5.  ~r(W)
```

Pel-21. A moderately tricky problem, especially for 'natural' systems with 'strong' restrictions on variables generated from existential quantifiers.

```
1.  ~P v f(a)
2.  ~f(b) v P
3.  P v f(X)
4.  ~f(X) v ~P
```

Pel-23.

```
1.  P v f(X) v f(Y)
2.  f(X) v P v f(b)
3.  ~f(a) v P v f(Y)
4.  ~f(a) v ~f(b)
5.  ~P
```

Pel-24.

```
1.  ~s(X) v ~q(X)
2.  ~p(X) v q(X) v r(X)
3.  p(a) v q(b)
4.  ~q(X) v s(X)
5.  ~r(X) v s(X)
6.  ~p(X) v ~r(X)
```

Pel-25.

```
1.  p(a)
2.  ~f(X) v ~g(X) v ~r(X)
3.  ~p(X) v f(X)
4.  ~p(X) v g(X)
5.  ~p(X) v q(X) v p(b)
6.  ~p(X) v q(X) v r(b)
7.  ~q(X) v ~p(X)
```

Pel-27.

```
1.  f(a)
2.  ~g(a)
3.  ~f(X) v h(X)
4.  ~j(X) v ~i(X) v f(X)
5.  ~h(X) v g(X) v ~i(Y) v ~h(Y)
6.  j(b)
7.  i(b)
```

Pel-30.

```
1.  ~f(X) v ~h(X)
2.  g(X) v f(X)
3.  ~g(X) v ~h(X)
4.  g(X) v h(X)
5.  i(X) v f(X)
6.  i(X) v h(X)
7.  ~i(a)
```

Pel-31.

```
1.  ~f(X) v ~g(X)
2.  ~f(X) v ~h(X)
3.  i(a)
4.  f(a)
5.  j(X) v h(X)
6.  ~i(X) v ~j(X)
```

Pel-32.

```
1.  ~f(X) v ~g(X) v i(X)
2.  ~f(X) v ~h(X) v i(X)
3.  ~i(X) v ~h(X) v j(X)
4.  ~k(X) v h(X)
5.  f(a).
6.  k(a)
7.  ~j(a)
```

D.2. Problems from Chang (1970)

Chang-1. In an associative system with left and right solutions, there is a right identity element.

```
1.  p(g(X,Y),X,Y)
2.  p(X, h(X,Y),Y)
3.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(X,V,W) v p(U,Z,W)
4.  ~p(k(X),X,k(X))
```

Chang-2. In an associative system with an identity element,if the square of every element is the identity, the system is commutative.

```
1.  p(X,e,X)
2.  p(e,X,X)
3.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(U,Z,W) v p(X,V,W)
4.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(X,V,W) v p(U,Z,W)
5.  p(X,X,e)
6.  p(a,b,c)
7.  ~p(b,a,c)
```

Chang-3. In a group, the left identity element is also a right identity.

```
1.  p(i(X),X,e)
2.  p(e,X,X)
3.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(U,Z,W) v p(X,V,W)
4.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(X,V,W) v p(U,Z,W)
5.  ~p(a,e,a)
```

Chang-4. In a group with left inverses and left identity every element has a right inverse

```
1.  p(i(X),X,e)
2.  p(e,X,X)
3.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(U,Z,W) v p(X,V,W)
4.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(X,V,W) v p(U,Z,W)
5.  ~p(a,X,e)
```

Chang-5. If S is a nonempty subset of a group such that if X, Y belongs to S then $X.Y^{-1}$ belongs to S, then the identity e belongs to S.

```
1.  p(e,X,X)
2.  p(X,e,X)
3.  p(X,i(X),e)
4.  p(i(X),X,e)
5.  s(a)
6.  ~s(X) v ~s(Y) v ~p(X,i(Y),Z) v s(Z)
7.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(U,Z,W) v p(X,V,W)
8.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(X,V,W) v p(U,Z,W)
9.  ~s(e)
```

Chang-6. If S is a nonempty subset of a group such that if X, Y belongs to S then $X.Y^{-1}$ belongs to S, then S contains X^{-1} whenever it contains X.

```
1.  p(e,X,X)
2.  p(X,e,X)
3.  p(X,i(X),e)
4.  p(i(X),X,e)
5.  s(b)
6.  ~s(X) v ~s(Y) v ~p(X,i(Y),Z) v s(Z)
7.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(U,Z,W) v p(X,V,W)
8.  ~p(X,Y,U) v ~p(Y,Z,V) v ~p(X,V,W) v p(U,Z,W)
9.  ~s(i(b))
```

Chang-7. If a is a prime and $a = b^2/c^2$ then a divides b.

```
1.  p(a)
2.  m(a,s(c),s(b))
3.  m(X,X,s(X))
4.  ~m(X,Y,Z) v m(Y,X,Z)
5.  ~m(X,Y,Z) v d(X,Z)
6.  ~p(X) v ~m(Y,Z,U) v ~d(X,U) v d(X,Y) v d(X,Z)
7.  ~d(a,b)
```

Chang-8. Any number greater than 1 has a prime divisor.

```
1.  d(X,X)
2.  ~d(X,Y) v ~d(Y,Z) v d(X,Z)
3.  p(X) v d(g(X),X)
4.  p(X) v l(1,g(X))
5.  p(X) v l(g(X), X)
6.  l(1,a)
7.  ~p(X) v ~d(X,a)
8.  ~l(1,X) v ~l(X,a) v p(f(X))
9.  ~l(1,X) v ~l(X,a) v d(f(X),X)
```

Chang-9. There exist infinitely many primes.

```
1.  l(X,f(X))
2.  ~l(X,X)
3.  ~l(X,Y) v ~l(Y,X)
4.  ~d(X,f(Y)) v l(Y,X)
5.  p(X) v d(h(X),X)
6.  p(X) v p(h(X))
7.  p(X) v l(h(X),X)
8.  ~p(X) v ~l(a,X) v l(f(a),X)
```

Chang-10. Given the rewriting rules :

A + B = B + A

A + (B+C) = (A+B) + C

(A+B) - B = A

A = (A+B) - B

(A-B) + C = (A+C) - B

(A+B) - C = (A-C) + B

Show that

(A+B) + C = A + (B+C)

(A-B) + C = A + (C-B)

A + (B-C) = (A-C) + B

(A+B) - C = A + (B-C)

```
1.  r( p(X,Y),p(Y,X) )
2.  r( p(X,p(Y,Z)), p(p(X,Y),Z) )
3.  r( s(p(X,Y),Y), X )
4.  r( X, s(p(X,Y),Y) )
5.  r( p(s(X,Y),Z), s(p(X,Z),Y) )
6.  r( s(p(X,Y),Z), p(s(X,Z),Y) )
7.  ~r(X,Y) v ~r(Y,Z) v r(X,Z)
8.  r(X,X)
9.  ~r(X,Y) v ~r(U,p(X,V)) v r(U,p(Y,V))
10. ~r(X,Y) v ~r(U,s(X,V)) v r(U,s(Y,V))
11. ~r(X,Y) v ~r(U,s(V,X)) v r(U,s(V,Y))
12. ~r( p(p(a,b),c), p(a,p(b,c)) )
13. ~r( p(s(a,b),c), p(a,s(c,b)) )
14. ~r( p(a, s(b,c)), p(s(a,c), b) )
15. ~r( s(p(a,b),c), p(a,s(b,c)) )
```

D.3. The Schubert's Steamroller problem as described by Stickel (1986).

Problem Statement :

Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them. Also there are some grains, and grains are plants. Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves. Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Therefore, there is an animal that likes to eat a grain eating-animal.

```
1.  ~f(X) v a(X)
2.  ~b(X) v a(X)
3.  ~w(X) v a(X)
4.  ~c(X) v a(X)
5.  ~s(X) v a(X)
6.  f(f)
7.  b(b)
8.  w(w)
9.  c(c)
10. s(s)
11. g(g)
12. ~g(X) v p(X)
13. ~a(X) v ~p(Y) v ~a(Z) v ~p(V) v e(X,Y) v ~m(Z,X) v
    ~e(Z,V) v e(X,Z)
14. ~c(X) v ~b(Y) v m(X,Y)
15. ~s(X) v ~b(Y) v m(X,Y)
16. ~b(X) v ~f(Y) v m(X,Y)
```

17. ~f(X) v ~w(Y) v m(X,Y)

18. ~w(X) v ~f(Y) v ~e(X,Y)

19. ~w(X) v ~g(Y) v ~e(X,Y)

20. ~b(X) v ~c(Y) v e(X,Y)

21. ~b(X) v ~s(Y) v ~e(X,Y)

22. ~c(X) v p(h(X))

23. ~c(X) v e(X, h(X))

24. ~s(X) v p(i(X))

25. ~s(X) v e(X, i(X))

26. ~a(X) v ~a(Y) v ~g(Z) v ~e(X,Y) v ~e(Y,Z)