INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9216249

Methods for an expert system to access an external database

He, Xiaoping, Ph.D.

Case Western Reserve University, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

**METHODS FOR AN EXPERT SYSTEM TO ACCESS
AN EXTERNAL DATABASE**

by
XIAOPING HE

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Advisor: George W. Ernst

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY

August 1991

CASE WESTERN RESERVE UNIVERSITY
GRADUATE STUDIES

We hereby approve the thesis of

XIAOPING HE

candidate for the Ph. D.

degree.*

Signed:

(Chairman)

Date JUNE 11, 1991

*We also certify that written approval has
been obtained for any proprietary material
contained therein.

I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

Xiaoping He

METHODS FOR AN EXPERT SYSTEM TO ACCESS AN EXTERNAL DATABASE

ABSTRACT

by

XIAOPING HE

This dissertation develops several methods by which an expert system can intelligently access an external database. The motivation for this research came from an advanced manufacturing application which required fairly sophisticated reasoning about a relatively large volume of factual knowledge which is stored in an external database. An expert system was designed and implemented to do this in a relatively efficient way which takes advantage of contemporary database and logic programming technologies. An interesting feature of this design is its use of meta-interpreters to engineer the system's software; i.e., a meta-interpreter was used to generate database queries when information from the external database was needed.

Although this expert system provide a practical solution, it also suggested that a more conceptual solution would be advantageous. A more powerful inference engine would remove the need for some of the specialized software that was part of the expert system and at the same time would allow the expert system

rules to be more declarative. The main feature of this new inference engine is that it performs a graph search; i.e., when attempting to solve a subgoal, it uses all possible answers from previously solved subgoals. Since the subgoals are essentially Prolog subgoals, in general there will be a set of answers for each subgoal. The basic problem which must be solved is that two subgoals may have a number of answers in common, but each may also have answers which are not answers to the other because neither subgoal is a substitution instance of the other.

We have developed a solution to this problem and incorporated it into a method which performs a graph search of a space defined by a logic program and a goal. We prove that this method is both sound and complete.

*To Xiaoqing
and to my parents*

ACKNOWLEDGEMENT

I would like to thank Prof. George W. Ernst for his continuous help and encouragement during my Ph.D study. He generously spent time and effort in advising, directing and reviewing this research. He also made many valuable suggestions which are very important to the completion of this dissertation.

I would like also thank Prof. Z. Meral Ozsoyoglu, Prof. Leon Sterling and Prof. Miles Kennedy, who are in my research advisory committee, for their suggestions to the proposal, and for their time and patience in reading and evaluating this dissertation. Prof. Z. Meral Ozsoyoglu also advised and supported me in my early study.

Special thanks are given to Dr. Fred Discenzo and some of his colleagues in Reliance Electric Corp. for their cooperation during the project for MOSES2. Their friendly help and suggestions are greatly appreciated.

Many thanks to all my friend in this department for their friendship and generous help in all regards.

My study was partly supported by the Center for Automation and Intelligent Systems Research at Case Western Reserve University.

Table of Contents

Chapter 1 Introduction	1
Chapter 2 Background	4
Chapter 3 MOSES2	16
3.1 Knowledge Representation	17
3.2 System Design	22
3.3 Query Formulation	26
3.4 Implementation and Empirical Results	31
Chapter 4 Graph Search	36
4.1 Justification	36
4.2 Basic Idea	43
4.2.1 Essential Issues in a Graph Search	44
4.2.2 The Constraints	46
4.3 The Graph Search Algorithms	49
4.3.1 Constraint Generation	49
4.3.2 Application of Substitution to a Constraint	51
4.3.3 Normalization	53
4.3.4 Application of Substitution to a Subgoal	57
4.3.5 Method of Inference	58
4.3.5.1 Rule Derivation Steps	59
4.3.5.2 Answer Derivation Steps	60
4.3.5.3 Complement Derivation Steps	61
4.3.5.4 The Search Tree	64
4.3.6 Data Structure	71
Chapter 5 Empirical Evaluation	74
5.1 Transitive Closure Examples	74

5.2 The Rules of MOSES2	86
5.3 Discussion	91
Chapter 6 Meta-Theory of Graph search	93
6.1 Properties of the Basic Methods	94
6.2 Soundness and Completeness of the new Inference Method	98
Chapter 7 Conclusions	112
7.1 Future Research	115
References	118
Appendix	121
A. The Rules of MOSES2	121
B. The Rules with Graph Search	125

1 Introduction

We have been involved in research on the integration of expert systems (ES) and database management systems (DBMS). Such integration is attractive for applications which employ expert knowledge to reason about a relatively large amount of data that should be managed by a DBMS. This project is a cooperative effort between the Center for Automation and Intelligent Systems at Case Western Reserve University and Reliance Electric Corp., an industrial sponsor of the center.

The initial part of the project was to develop an expert system for an application which involves the use of a DBMS. The reason for starting with a particular application is to insure that the project has a "real world" basis. An equally important part of the project is to understand the issues involved in the integration of ES and DBMS in a more general context because any one application will probably only address part of the problem.

The application that was selected for the project is to select the best motor for an industrial automation project from a database of over 10,000 motors manufactured by the Reliance Electric Corp. An initial version of a motor selection system called MOSES (MOtor Selection Expert System) [Disc88] was developed and used by Reliance personnel. The expertise embedded in MOSES is quite limited and the data is stored in several files instead of in a DBMS. The initial goal of this research was to significantly increase the amount of motor selection expertise in MOSES. The expert system thus developed is called

MOSES2 [HeEr89]. It was desirable to replace the data files with a relational DBMS and to implement the system in Prolog. The DBMS used was Ingres. A meta-interpreter was used in coupling Prolog and Ingres.

Although Prolog is a convenient and powerful language for implementing ES, the design and implementation of MOSES2 was still a sizable task. The work on MOSES2 exposed several issues that complicated the design and implementation. For example, formulating the rules requires an understanding of how they will be executed. Not only is this an extra burden to the rule designer, but it also makes the rule less declarative which cause them to be more difficult to understand and "debug". Another issue is the fact that DBMS gives all the answers to a query at once which is very different than Prolog's evaluation style. In MOSES2 this difference was bridged by a meta-interpreter and the Prolog database. These issues will be discussed in detail later. We mention them here to give the reader some idea of what the issues are.

Based on the experience with MOSES2 we looked for a more conceptual way to deal with these issues as opposed to the implementation techniques used in MOSES2. It is important to realize that these issues are not specific to the particular application that MOSES2 deals with; they are generic to the evaluation mechanism of Prolog and the difference between Prolog and a DBMS. They will appear every time a system of this kind is built and it is better to solve them at a more conceptual level in stead of dealing with them on individual bases. This motivated us to design a new inference engine which keeps the good feature of Prolog and supplements them with additional capabilities.

Our investigation showed that an inference engine that does a graph search seems to cope with these issues in a reasonably good way. That is, the inference engine stores the answers of subgoals and make them available for solving other subgoals in the future. To do this we designed an algorithm so that graph search can be carried out in a general way; i.e., it can deal with the case when two subgoals have some answers in common and each one has some answers which are not answers of the other. We also investigated some of the empirical properties and the theoretical properties of this new inference engine.

This dissertation is organized as follows. Related research is discussed in the next chapter. Chapter 3 contains the detailed description of MOSES2. The new inference engine is described in Chapter 4. It starts with a discussion of the issues which complicated the design and implementation of MOSES2. Then the algorithms which constitute the new inference engine are given, including some examples which illustrate the basic idea and a complete description of the algorithms. In Chapter 5, some empirical properties about the new inference engine are given based upon several typical cases. Chapter 6 contains the proof of soundness and completeness of the new inference engine, and the conclusion follows in Chapter 7.

2 Background

A number of different research efforts have been concerned with allowing a Prolog program to use a DBMS to access an external database. [Bocc86a, Bocc86b, ChWa86, HeEr89, Ioan88, Rama??, ScWa86, VaCJ83, VaCJ85, Zani86].

A system named EDUCE [Bocc86a] [Bocc86b] was developed at the European Community Research Center. It offers two different methods for coupling Prolog and a DBMS: close coupling and loose coupling. The close coupling was referred as integration in [Bocc86a] [Bocc86b].

The loose coupling is implemented by running two intercommunicating processes: one for Prolog and one for the DBMS. Pipes are established between the two processes. The pipe is a communication channel from one process to another. It is a facility offered by Unix. To access the external database, users have to write explicit QUEL queries in the context of Prolog. For example,

```
?- retrieve([employee.name = N, employee.salary = S],  
           (employee.salary > 30,000))
```

produces, for the appropriate database,

```
N = 'John'  
S = 50,000
```

```
N = 'Mike'  
S = 63,000
```

i.e., it retrieves the *employee.name* and *employee.salary* of employees with

employee.salary higher than 30,000. The query is sent to DBMS through a pipe. The answers from DBMS are returned through another pipe. This pipe also serves as a queue to store the answers so that the Prolog interpreter can pick up one tuple each time it needs a new tuple. This mechanism allows for the definition of virtual relations which behave just like a Prolog fact. Although this mechanism capabilities include all of QUELL, a disadvantage of it is that the use of DBMS is not transparent to the users.

This loose coupling of EDUCE works well when there is no recursion. With recursion, it becomes necessary to relate replies from the DBMS to their originating queries. One possible way of doing this is to set up some kind of communication protocol to label queries and replies. Another way is to set up a new pair of pipes for each new query. However, neither of these have been implemented; instead EDUCE uses the close integration mechanism to evaluate queries involved in recursive definitions.

Briefly speaking, the close integration part of EDUCE is Prolog extended to use the access method module of a DBMS. It is only used for retrieval from database; update is not allowed by close integration. An example of the use of retrieval through close integration is *?- retr(employee(john, Salary))*. In this example, the relation *employee* is searched for *john's* salary since *john* is a constant and *Salary* is a variable here. The use of DBMS is more transparent in the close integration than the loose coupling of EDUCE although it is still not completely transparent. No boolean conditions can be used in the query of close forms, which implies that users need to be careful not to retrieve too much

unnecessary data. The evaluation procedure starts by transforming the (only) argument of *retr* into a list. The head of the list is instantiated to the name of relation and the tail is instantiated to a list of attributes, some as variables and the others as constants. Once this is done, the corresponding relation is opened and the first tuple is retrieved. The opening of the relation is actually done by opening the file which contains the relation. A descriptor *D* is also created. If the file for the relation was already open then a new descriptor *D* is created.

Descriptors not only keep static information about a relation, e.g. file name, cardinality, etc, but also maintain information of a dynamic type. In particular, information about the last tuple accessed is kept by the descriptors. This use of descriptors is essential in recursive cases. Without the descriptors recursive query on a given relation would be restricted to as many levels of recursion as the numbers of files that the host operating system allows to keep opened at any particular time.

Recursive definitions which include expressions whose evaluation need loose coupling are evaluated by a hybrid strategy. Loose coupling is used for the non-recursive part of the definition. For the recursive part, intermediate relations are established for the base relations first. This is done by preparing a query in loose form, executing it and saving the results in the intermediate relations. The close integration is then used for retrieval from these intermediate relations for the evaluation of the recursive definitions.

PROSQL [ChWa86] is a system which extends Prolog with SQL/DS. It was

developed at IBM Watson Research Laboratory. Prolog is extended with one special predicate "SQL" which takes any SQL statement as its argument. Users are responsible for SQL statements, i.e. the database access is not transparent to the user. It offers the full functionality of a DBMS, including data definition, data manipulation, concurrency control, etc. The answers from the DBMS are put into Prolog working database using *assert* facility.

BERMUDA [Ioan88] is a system interfacing Prolog to a database machine. It was designed and implemented at the University of Wisconsin. Prolog and the database machine stay unchanged. The central component of the system, called BERMUDA Agent, is responsible for the communication between Prolog and the database machine. It receives database predicates along with appropriate information to form database queries with possible optimization; it caches answers from the database machine for later use and also gives them to Prolog one tuple at a time; it also supports multiple Prolog processes as well as multiple database queries. All of these systems contain a mechanism for giving the Prolog interpreter one tuple at a time, as opposed to a set of tuples, because of the way in which Prolog evaluates goals.

Another issue addressed by our research is how to avoid re-evaluating a subgoal by saving the results of its first evaluation. This issue arises in a number of other research projects in a somewhat different context than in our research. Tamaki and Sato in [TaSa86] presented a new version of resolution called OLDT-resolution. In order to explain what the OLDT-resolution is, it is easier to start from SLD-resolution [Lloy84] assuming that the reader is somewhat familiar

with it.

The *Computation rule* in SLD-resolution is the strategy used to select the subgoal from a resolvent to be expanded next, while *search rule* is the strategy of searching for success path in an SLD-tree. SLD-resolution does not commit to any specific computation rule and search rule; it leave both of them open. Prolog adopts SLD-resolution with left-to-right for the computation rule and left-to-right depth-first for the search rule. OLD-resolution is between these two cases; it is SLD-resolution with left-to-right for the computation rule. The search rule in OLD-resolution is open.

OLDT-resolution is OLD-resolution with "Tabulation". A so called *solution table* is used in OLDT-resolution to store the answers. Users are given flexibility to choose *table predicates*. Tabulation is only done on the nodes whose left most atom is an atom of a table predicate. Let us call these nodes *table nodes*. Some of the table nodes are *solution nodes* which contribute answers while the others are *lookup nodes* which look for answers in the table. An algorithm is given to register a new table node either as a solution node or a lookup node depending on the leftmost atom of the node and the currently existing solution nodes. For any lookup node, its leftmost atom is always an instance of the leftmost atom of the corresponding solution node. i.e., it is the same or subsumed subgoal.

OLDT-resolution creates an *OLDT-structure* as follows. An *immediate-extension* to an OLDT-structure is either an *OLD-extension* or a *lookup-extension*. An OLD-extension is only applied to a non-lookup node. It is similar to one step

of an SLD-derivation. The difference is that all the applicable rules are applied to the node to generate a set of new nodes, while an SLD-derivation only applies one rule each step. A newly produced node from the OLD-extension is registered if it is a table node. A lookup-extension is only applied to a lookup node. Associated with the lookup node there is a pointer which points to a sublist of the solution table. One solution will be picked up from the list. If this solution is resolvable with the leftmost atom of the node, a new child node is then created and labeled with the resolvent.

Successive application of the immediate-extensions to an initial goal node produce an *OLDT-tree*. To bound the number of distinct subgoals to be solved, a more general goal may be created for several similar subgoals. For instance, instead of solving subgoals $p(f(a), Y)$, $p(f(h(a)), b)$ and $p(f(g(X)), Y)$, it might be clearer to create a new goal $p(f(X), Y)$ and make it the root node of a separate tree. The root node of this new tree is registered as a solution node and the nodes corresponding to the above subgoals as lookup nodes. The OLDT-structure created for this case is essentially a forest of OLDT-trees, one for each of the above created goals.

An *OLDT refutation* for a given program P and a negative clause C , is a path in the OLDT structure for (P, C) , from the initial root to a node labeled with the null clause. In [TaSa86], it is proved that for programs defining finite relations only, OLDT-resolution is terminating and complete under an arbitrary search rule, provided that all the predicates are designated as table predicates.

In [Warr89] Warren modified the WAM (Warren Abstract Machine) [Warh83] for Prolog evaluation to extend it to support the *extension table* evaluation strategy [Diet87]. Answers of subgoals are stored in the extension table, so that later invocation of the same or subsumed subgoals use the results in the table instead of recomputing the answers. It also saves the environment so that a suspended call of a subgoal can be resumed later on. (by call or invocation of a subgoal we mean the evaluation of the subgoal by the interpreter) The reason some call might be suspended is that when a call looks into the extension table for answers, the call of a subsuming subgoal which contributes to the answers may not have produced a single answer yet, nor produced a new answer since last call if the current call is a backtrack call. Under such condition the current call will be suspended since there is no new answer for it, but it is not a failure either.

The modified machine is called XWAM and is only designed for datalog program (Prolog without function symbols). The purpose of the XWAM is to both reduce redundant computation and make the evaluation strategy "more complete" than Prolog. More complete here means it will terminate on all the programs on which Prolog terminates, and it will also terminate on some programs on which Prolog does not. e.g., left recursive rules. Both this method and OLDT-resolution use the results of some previous evaluations of subgoals. This implies that these results must be stored in memory in some way so that they can be retrieved when needed. Although this has the desirable properties mentioned above, the amount of memory that is needed may be very large. The graph search algorithm that is given in Chapter 4 uses a similar kind of evaluation stra-

tegy.

Our new inference engine uses *constraints* in order to do a graph search. The name *constraint* comes from the *constraint logic programming* [JaLa87] where a rule is a conventional Prolog rule augmented with several primitive constraints. e.g.,

taxable(Person, Salary, Interest) :- Salary+Interest>3000, of_age(Person).

where *Salary+Interest>3000* is a primitive constraint. The rule states that a person's income is taxable if he is of age and his salary plus interest income is over 3000. The evaluation procedure of the constraint logic programming has special methods for processing constraints. It is different from the resolution which is used by conventional logic programming. The constraints used in our new inference engine is a special kind of constraints whose basic elements are inequations over terms. They are quite similar to the constraints in [Colm84]. Linear inequalities as in the above example are not used by our new inference engine.

Colmerauer designed algorithms to process a special kind of constraint which contains only equations and inequations for logic programming [Colm84]. An algorithm called reduction algorithm was given first to solve a system containing only equations. An *endless system* is a system of equations in which every term which occurs as the right-hand side of an equation also occurs as the left-hand side of an equation. e.g., $(X=Y, Y=Z, Z=X)$. A *reduced system* is a finite system of equations having the following two properties: (1) the left-hand sides of

its equations are distinct variables, (2) it does not contain an endless subsystem. A reduced system is considered solved since an assignment to the variables occurring at right-hand sides of equations determines a solution. Taking a reduced system $(X=f(Y), Z=W)$ as an example, the assignment $(Y=a, W=b)$ leads to a solution $(X=f(a), Y=a, Z=b, W=b)$. Given a system of equations, the reduction algorithm produces a reduced system, and therefore solves the equation system. The correctness of the algorithm was also proven. i.e., the algorithm terminates and produce a correct solution.

A system containing both equations and inequations is processed in the following way in [Colm84]. The equations are grouped together and processed by the reduction algorithm to produce a reduced system, say Se . One inequation, say $s \neq t$, is then picked up and converted to an equation which is, in this case, $s=t$. The reduction algorithm is applied to the new equation system $Se \cup \{s=t\}$. If the algorithm fails, meaning $s=t$ is inconsistent with Se , $s \neq t$ is simply removed since Se implies $s \neq t$. If the algorithm succeeds it will produce a new system of equations S such that $S = Se \cup Te$. i.e., the new reduced system is the old one plus some extra equations, Te . If Te is empty, the reduction fails because it indicates that Se implies $s=t$, and therefore $Se \cup \{s \neq t\}$ has no solution. Otherwise the equations in Te are converted back into inequations which are guaranteed to be in so called "simplified" form. If the reduction fails, then the Se and the inequation are mutually inconsistent and the process terminates. Otherwise, the above process is applied to each of the remaining original inequations, one after another. The final result is the reduced (equation) system Se and a set of simplified inequations if no

failure was encountered. The resulting system is proven to be satisfiable. The processing of constraints in our graph search uses some of these simplification techniques.

In [LaMM87], Lassez, Maher and Marriott described their work on unification, anti-unification and systems of equations and inequations. An *anti-instance* of a term is its generalization. e.g., $f(X)$ is an anti-instance of $f(g(a))$. For a set of terms, there is a unique least common anti-instance up to variable renaming. A *mgs* (*most general solution*) for a system E of equations is the least common anti-instance of the *solution* set for E ; while a *solution* for E is a grounding substitution which makes E true. For a system of equations and inequations, *mgss*, most general solution set, instead of *mgs* is needed. One of the theorems in [LaMM87] states that no finite set of *mgu* can provide an explicit representation for the solutions for a system of equations and inequations unless the system is unsolvable or its collection of inequations is redundant. This theorem is of importance to us because it justifies the use of inequations in our new method for resolution. It is also proved in [LaMM87] that if there is a finite number of function symbols then there exists an algorithm that will produce an *mgss* for a system of equations and inequations. For the case where there is infinite number of function symbols, such system has an *mgss* iff it is unsolvable or its collection of inequations is redundant.

Many existing ES's use graph search, but their search methods are quite different than a graph search in logic programming because their use of variables is more specialized. MYCIN [Shor76] is one such example. MYCIN's process

for applying a rule essentially instantiate any variable in a rule with a constant. Since the resulting rule is ground, its consequent is also ground which corresponds to a single answer in logic programming. In addition, each literal in the premise of the rule will have a particular value, since the rule has been made ground, which is the extent to which the literal is true or false as indicated by a certainty factor. This allows MYCIN to use a slightly modified version of a standard method (see, for example [Nils82]) for searching AND/OR graphs. A more general use of variables in the process of applying a rule allows a single rule to produce multiple answers and some of the answers may themselves contain variables. In this situation, AND nodes may contain common variables which makes the solution of an AND node dependent upon one of its sibling. This prevents the use of the AND/OR search methods in [Nils82] because they assume that each AND node can be solved independently of its siblings. This dependence of AND subgoals and the possibility of multiple solutions for them is described in detail in chapter 4.

Zhou and Ozsoyoglu have developed techniques to avoid producing duplicate answers for multiple queries to a database [ZhOz90]. Multiple database queries are assumed to be submitted at the same time for processing. They are first grouped into three groups according to the implication relationship among the queries so that the queries from one group can be answered by using the results from queries of the previous group. There is also partial implication relationship among the queries in the first as well as the last group. An optimization is done on the order in which the queries are processed in one group to take

advantage of this partial implication order. Although there is no implication relationship among the queries in the second group, these queries may have common subexpressions, and therefore, overlapped answers. The queries are then broken up into smaller queries so that overlapped answers will be retrieved only once. Two new relational algebra operations were also proposed to optimize the multiple query processing at physical level. Of course, the method they used is different than the above methods due to the difference between the evaluation procedures in databases and the ones in logic programming. But the idea of reusing previous results is the same.

Apt, Bol and Klop have done some research on *loop checks* to avoid producing duplicate answers in logic programming [ApBo89, Bol90a, Bol90b]. However, the main effort in their work lies in finding and pruning infinite loops. i.e., finding paths in an SLD-tree which will loop forever and prune them. There is no need for tabulation to store and reuse answers.

3 MOSES2

Detailed description of MOSES2 is given in the following sections of this chapter. The most interesting aspect of MOSES2 is that it uses a large (by expert system standards) external database which it intelligently accesses. Both the application domain and the design of the expert system are described, but its interface to the external database is described in more detail because this is part of conceptual interest.

The application problem is to select the best motor for an industrial automation project from a database of over 10,000 motors manufactured by the Reliance Electric Corp. Although Reliance has a catalogue of their motors that contains general properties and prices, often this is not enough information to select a good motor for an application. For this reason the engineers at Reliance have a printout of a motor database and receive monthly updates of it. The database has about 60 attributes of motors, but some are not included in the printouts. In addition, each motor may have several special property which are listed; there are about 50 such special properties, e.g., special overload duty. The engineers find these printouts awkward to use.

In selecting motors the engineers make use of considerable expertise that is not in the database. Some of this expertise is concerned with a particular kind of application, e.g. winder applications, while other expertise is application independent, e.g., special cooling is required to operate a NEMA motor at more than 3,000 feet above sea level. The problem is sufficiently difficult that the engineer

specifies the use of the motor on the order form and a motor engineer at the manufacturing plant will double check if the motor can be used for that purpose.

An initial version of a motor selection system MOSES has been developed and used by Reliance personnel. It runs on a VAX in Cleveland, but it can be accessed remotely by people across the country. The users of the system find it quite helpful and there is some interest in extending it to other Reliance products such as gears and reducers. But, this will be difficult with the MOSES because the data is stored in several files instead of using a DBMS. Also the expertise embedded in the MOSES is quite limited. MOSES2 removes, at least partially, these limitations.

3.1 Knowledge Representation

A relational DBMS was chosen for managing the motor data. The volume of the data is one of main reason for using a DBMS. Another advantage of using a DBMS is its efficiency for both storage and retrieval of data. It also maintains the consistency and integrity of data and provides for easy of extension of the database with new information.

Conceptual modeling was used in the design of the database which is a standard database design methodology[Ullm88]. This process resulted in a database with 6 relations: *main*, *field*, *armature*, *remarks*, *coremarks* and *frames*. The *main* relation contains 17 attributes which are frequently used in motor selection. Another relation *frames* has 13 attributes to represent different features of motor

frames. The *field* and *armature* relations each consists of about a dozen of attributes of motor fields and motor armatures, while *remarks* and *coremarks* contain special features of motors and only have a few attributes.

This structuring allows for both efficient access and storage of the data. For example, each motor has a frame but the same frame may be used in many different motors. Thus, the frame information is only stored once instead of once for each motor with the same frame. The *frame number* attribute links the frame data with the data in the *main* relation. Another advantage of this structuring is that field and armature relations only contain attributes which are used infrequently. Thus the information in these relations is seldomly accessed.

For efficiency considerations it is important to distinguish between the facts stored in the database of the DBMS which resides in secondary storage and the facts stored in Prolog's database which resides in main memory. To make this distinction explicit in the following text, we use *DB* to denote the database of the DBMS whereas *PDB* denotes the Prolog database.

In selecting motors the engineers make use of considerable expertise about a particular kind of application. For this reason MOSES2 is only designed for one difficult kind of motor application, *winder* applications. To extend it to other applications would require knowledge peculiar to those applications.

A typical winder application is a process in the paper industry. The paper which is input to the process is on a roll controlled by a winder motor. The paper is unwound from it and fed into the processing line. In the last component of the

process, the paper is wound onto another roll controlled by another winder motor. Paper tension and line speed are held constant, which means the motors have to change their speeds, continuously, because of the changing diameters of the paper rolls. For safety reasons, the motors must be able to stop the rolls in a short time, usually about a minute. Due to the high inertia of a full roll of paper, a considerable amount of horsepower is needed for stopping in addition to the power for maintaining the paper tension.

The specification of a paper winding process, which is prescribed by the customer, contains a number of process parameters such as line speed, maximum and minimum diameters of the paper rolls, etc. Other parameters will be inferred by MOSES2; e.g., there is a rule which has the line speed and maximum diameter for inputs and calculates the minimum motor speed (rpm) which is then stored in the PDB for later use.

There are other rules, in addition to these calculation oriented rules, which contain expertise about motor selection. Figure 3.1 shows an example of such a rule. To understand this rule, we need the following knowledge about motors. Figure 3.2 shows a motor performance curve which gives the horsepower a motor can output as a function of its speed. The smallest speed at which the motor reaches full horsepower is called the *base speed*. The maximum output horsepower remains constant when the motor's speed is between the base speed and its maximum speed. This horsepower is called the *nominal horsepower*. Normally a motor operates at a speed higher than its base speed and with its output less than or equal to the nominal horsepower. But a motor can operate at a speed


```

motors1(Design, Hp, Base_rpm, Fw_rpm, Winding, Motor_type, Cost, Mult_field,
        Tsel, Tdec)
:- find(frame_change, yes), find(hp_low, Hp_low),
   find(hp_high, Hh), find(base_rpm_low, Brl),
   find(fw_rpm_low, Frl), find(fw_rpm_high, Frh),
   find(motor_type, Motor_type), find(frame_series_low, Fsl),
   find(frame_series_high, Fsh), find(frame_type, Frame_type),
   find(enclosure, Enclosure), find(rpm_min, Rpm_min),
   find(remark, Remark),
   main(Design, Enclosure, Hp, Avolts, Base_rpm, Fw_rpm, Winding,
        Frame, Duty, Ovld_pct, Ovld_time, Fvolts, Service_factor,
        Altitude, Cost, Motor_type),
   field(Design, Tsel, Tdec, Mult_field),
   remarks(Design, Remark, _),
   frames(Frame, Frame_type, Frame_series, _),
   Hp =< Hh, Brl =< Base_rpm, Frl =< Fw_rpm, Fw_rpm =< Frh,
   Fsl =< Frame_series, Frame_series =< Fsh, Hp_low =< Hp,
   Base_rpm =< Rpm_min*Hp/Hp_low.

```

Figure 3.1: A rule which contains predicates which are DB relations as well as ordinary predicates.

lower than the base speed so long as its operating point is under the curve in Figure 3.2; e.g., the motor can operate at point P but not at point Q.

This knowledge is represented in the rule in Figure 3.1. The head of the rule is a predicate whose arguments are 10 motor attributes which are of interest for later reasoning and thus will be stored in the PDB. The first part of the rule body has a number of *find* predicates which retrieve the values of parameters from the PDB; the names of these parameters are arguments of the predicates. For example, *find(rpm_min, Rpm_min)* is evaluated by retrieving the value of minimum speed and assigning it to the variable *Rpm_min*. The next part of the rule contains the predicates *main*, *field*, *remarks*, and *frame* which are relations

Figure 3.2: The standard relationship between the horsepower of a motor and its speed.

stored in the DB and thus these predicates are evaluated by the DBMS. The remainder of the rule are comparisons which constrain the tuples in these relations. To see how the above expertise is embedded in the rule, consider the last constraint which is $Base_rpm \leq Rpm_min * Hp / Hp_low$. $Base_rpm$ is the base speed of a candidate motor, Hp is its nominal horsepower, Rpm_min is the minimum speed required by the winder application, and Hp_low is the amount of horsepower that the motor must output for the application. The reader can see that $Base_rpm / Hp \leq Rpm_min / Hp_low$ assures the operating point of the motor is below the first segment of the curve in Figure 3.2. The preceding constraint $Hp_low \leq Hp$ insures that the operating point is also below the second part of the curve.

3.2 System Design

The Prolog interpreter was chosen for the inference engine of MOSES2. A major advantage of this is that we do not have to implement an inference engine. The reason we were able to use the Prolog interpreter is due to the characteristics of the knowledge base. The rules can be expressed easily in Prolog as showed above. Actually only a subset of Prolog is used because the rules do not contain cuts, which makes them more declarative. Another reason for adopting Prolog is that the control knowledge is also naturally represented in Prolog. The only control knowledge needed for this application is the order in which to evaluate the goals in the body of a rule and the order in which the rules with the same head should be evaluated.

Explanations are important for this application, but they are not provided by the Prolog interpreter. Fortunately there is a standard method for generating the necessary explanations by the use of a meta-interpreter[StLa86, StYa89], and this is used as the basis of the explanation facility in MOSES2.

A special user interface was designed for this application. It was modeled after the one in MOSES which was quite successful. Its main feature is the use of menus to communicate with users. The user interface in MOSES2 was implemented in the C programming language with the support of the Curses Package in UNIX. C is better suited than Prolog for such screen management and there is a nice interface between C and Prolog.

The user interface of MOSES2 has two major functions. The first is that user interaction does not have to be directly represented in the rules. For example, when *line speed* is needed during reasoning, the rule only needs to say "what is the line speed" to the user interface which then takes care of the rest. It will retrieve the line speed if it is stored in the PDB; otherwise it will ask the user and save the answer so that the user will not be asked the same question later. The second function of the user interface is to provide for friendly interaction with the user and menus are used for this purpose. Figure 3.3 shows a sample menu which tells the user that the system is about to access information in the DB. The user can ask questions about the query shown on the screen by typing the name of a motor attribute or he can select a choice listed at the bottom by moving cursor.

In designing MOSES2, most of our effort was devoted to the interface between the DBMS and the other parts of the system. The remainder of this section describes the overall structure of this interface. The details of how database queries are formulated is relegated to the next section.

As described above, rules are expressed in Prolog (see Figure 3.1) and the database relations appear to be the same as other predicates. This representation not only makes the rules homogeneous but also makes the DB somewhat transparent to a person who is either reading or formulating the rules. In fact, in testing an initial version of MOSES2, a small amount of motor data was stored as facts in the PDB which allowed it to be retrieved directly by the Prolog interpreter.

The following query is formed and will be sent to DBMS to search for super_rpm motors. If you have any question about the query, please type in The name in question :

303.03 =< HP =< 606.06 954.928 =< FW_RPM
 BASE_RPM =< HP*0.875352 MOTOR_TYPE = super_rpm

☐search DB ☐do not use this query ☐quit (motor selection)

Figure 3.3: A typical menu in the user interface.

There are two main issues which were addressed in designing the interface to the DBMS: the DB accesses should be reasonably efficient, e.g., the tuples retrieved should be pertinent to solving the goals in a rule. The other major issue is that the execution strategy of Prolog is very different than the strategy of a DBMS. Prolog finds an instantiation which solves a goal and then executes the next goal. A DBMS finds all tuples (instantiations) which satisfy a goal (query) before moving on to the next goal (query). The design needs a mechanism to interface between these two very different control strategies.

The basic strategy of the interface is to separate the goals in a rule into two parts: one part will be evaluated by the Prolog interpreter while the other part will be turned into a DBMS query. It is important to include all of the goals which are pertinent to the query in the latter. Even though the Prolog interpreter could be used to evaluate some of these goals, the efficiency is greatly improved by incorporating them into the DBMS query as described in the next section.

The result of the query is a set of tuples each of which is essentially an instantiation of some of the variables in a rule. Something has to be done to make these instantiations available to the Prolog interpreter one at a time since this is Prolog's execution strategy. To do this, these instantiations are stored in the PDB, and each time another instantiation is needed, the next one is retrieved from the PDB and passed to the Prolog interpreter. Thus, the DB is only accessed once at some point before the first instantiation is needed by the Prolog interpreter. This simple strategy works well in MOSES2 and it should work well whenever the result of DB retrieval is not too large.

The basic mechanism in MOSES2 for implementing the above strategy is a meta-interpreter. [Ster86, StBe89] It uses the normal execution strategy of Prolog for those rules which do not contain DB predicates. For the other rules, it uses the above strategy which is described in more detail below. It evaluates some goals which are not incorporated into the DBMS query, and after the query the remaining goals are evaluated. An important feature of this method is that instantiations produced by evaluating the first set of goals can be used in formulating the DBMS query which often has a considerable impact on the efficiency of the

retrieval.

Now the structure of MOSES2 can be summarized. It has an external database (DB) managed by a relational DBMS and a rule base which represents the application expertise. The Prolog interpreter is the inference engine and a meta-interpreter controls the reasoning and intelligently accesses the DB when necessary. MOSES2 also has a friendly user interface, and an explanation facility which is based on a separate meta-interpreter that stores information about the system's reasoning and presents a relevant part of it to the user when he requests an explanation. These two meta-interpreters are combined together to form a single meta-interpreter which is the main executive in MOSES2. The method for combining the meta-interpreter is based on the technique described in [StYa89, YaSt89].

3.3 Query Formulation

The meta-interpreter in the interface to the DBMS examines the rules as they are executed and applies special processing to those that contain predicates which denote DB relations. An example of such a rule is the one in Figure 3.1. It contains three kind of goals in its body. One kind is a goal whose predicate is a DB relation name such as *main*. Another kind is a goal whose predicate is a relational operator such as $<$ and $>=$ and whose arguments are terms containing variables, some of which denote the values of attributes in a tuple of a DB relation. Thus, these comparison goals can be viewed as constraints on the tuples which should be retrieved from the DB. The third kind is any other goal in a

rule, such as a goal whose predicate is *find* in Figure 3.1. In general any such rule has the form shown in Figure 3.4, where the *D*'s are the goals whose predicates are DB relation names; the *C*'s are the comparison goals and the *A*'s are the remaining goals.

$$P :- A_{11}, A_{12}, \dots, D_{11}, D_{12}, \dots, C_{11}, C_{12}, \dots, A_{21}, A_{22}, \dots, \dots$$

Figure 3.4: The general form of a rule. The *D*'s are DB goals; the *C*'s are comparison goals and the *A*'s are the remaining goals.

Our strategy for evaluating such rules is to evaluate each group $A_{i1}, A_{i2}, \dots, D_{i1}, D_{i2}, \dots, C_{i1}, C_{i2}, \dots$ in the same way. The Prolog interpreter evaluates the *A*'s. Then the meta-interpreter formulates one DBMS query consisting of the knowledge in the *D*'s and *C*'s and the result of the query will be stored in the PDB. The meta-interpreter will prevent the Prolog interpreter from evaluating the *D*'s and *C*'s; instead it will pass one of the tuples retrieved by the query to the Prolog interpreter whenever it would normally evaluate the *D*'s and *C*'s.

The query formulation process can be nicely illustrated by an example. For the rule in Figure 3.1 the query in Figure 3.5 is produced which is expressed in the SQL query language. The symbols in bold face are SQL key words. The symbols in *italic* denote the instantiated values of the variables in Figure 3.1. Thus the actual query will have constants in these locations. The first part of the


```

select  main.design, main.encl, main.hp, main.avolts, main.baserpm,
        main.fwrpm, main.winding, main.frame, main.duty,
        main.ovld_pct, main.ovld_time, main.fvolts, main.service_fact,
        main.altitude, main.cost, main.motor_type,
        field.tsel, field.tdec, field.mult_field,
        remarks.remark,
        frames.frame_type, frames.frame_series
from    main, field, remarks, frames
where   main.design = field.design and main.design = remarks.design and
        main.frame = frames.frame and main.encl = Enclosure and
        remarks.remark = Remark and frames.frame_type = Frame_type and
        main.hp <= Hh and Brl <= main.baserpm and Frl <= main.fwrpm
        and main.fwrpm <= Frh and Fsl <= frames.frame_series and
        frames.frame_series <= Fsh and Hp_low <= main.hp and
        main.baserpm <= Rpm_min*main.hp/Hp_low

```

Figure 3.5: The SQL query which is formulated for the rule in Figure 3.1.

body of the rule in Figure 3.1 is a dozen or so *find* goals which are the *A*'s for this rule. The *D*'s are the goals whose predicates are *main*, *field*, *remarks* and *frames*. The remaining goals are the *C*'s. The *A*'s are evaluated by Prolog. The *D*'s and *C*'s are used to formulate the query shown in Figure 3.5. The *select* part of the query is the list of all the attributes whose values are to be retrieved. The *from* part of the query gives all the relations referenced in the *D*'s. The *where* part is the search condition which consists of the constraints that the retrieved tuples must satisfy. All the *C*'s in the rule are translated into SQL and incorporated as part of the search condition. The rest of the search condition concerns equal joins and run time instantiations which are explained below.

Two different DB goals, say *D1* and *D2*, in a rule may have some common variables, e.g., the variable *design* in the *main* and *field* goals in Figure 3.1. This indicates that any valid instantiations of *D1* and *D2* must instantiate such a

common variable with the same constant. In a relational database this is an equal join between the two relations referenced by *D1* and *D2*. The query formulating process will identify the equal join attributes and incorporate them into the query. In Figure 3.5, *main.design = field.design* indicates such an equal join. An instantiated variable or a constant appearing in a *D* goal constrains the value of the corresponding attribute. They are also translated into part of the search condition of the query. In Figure 3.5 for example, "*main.enclosure = Enclosure*" is a part of the search condition which is due to the instantiation of the variable *Enclosure*.

The major parts of the query formulation method are the following: The select part of the query is formulated by checking the predicate names of the *D*'s. The relation names in the query are just these predicate names. The list of the attribute names can be found through the reference of the relation names. The where part of the query consists of three subparts. One is the translation of the *C*'s into SQL; another is the equality of join attributes which is formulated by detecting the common variables among the different *D*'s. The last part equates the values of attributes to instantiated variables and the constants in the *D*'s. Note that Prolog requires the argument of comparison goals to be instantiated before they are evaluated. This means that the part of search condition corresponding to *C*'s contains only constants and attribute names because all the variables in the *C*'s must also occur in either an *A* goal or a *D* goal. Actually this is also true for the whole search condition since the other parts of it only involve attributes of relations including equal join attributes and run time instantiations.

When formulating the query the meta-interpreter needs to know which predicates are DB relation names, and also the attribute name and type of each argument position of such a predicate. This information is contained in the data dictionary of the DBMS. Although managed by the DBMS, the data dictionary is also stored as facts in the PDB so that the meta-interpreter has easy access to this information. One such fact is shown in Figure 3.6 which lists all the attribute names of the *main* relation. From this fact you can find the attribute names corresponding to any argument position of the predicate *main*. For simplicity, such facts were hand-coded, but they should have been extracted from the data dictionary of the DBMS.

A key feature of this query formulation method is that it attempts to keep the number of tuples retrieved from the DB reasonably small for the purpose of efficiency. To do this it is essential that more than one goal in the body of a rule is used to formulate a query. In winder application, for example, an attribute as fundamental as horsepower is not known, which is quite surprising. Rather expertise is used to infer reasonable bounds on horsepower before retrieving motors from the DB. These bounds are expressed as comparison goals in Figure 3.1, i.e., the goals $Hp_{low} \leq Hp$ and $Hp \leq Hh$. If they were not included in the query of the relation *main*, there would be no constraints on the horsepower and an extremely large number of motors would probably be retrieved. Of course, if additional comparison goals are also used in formulating the query, the tuples in the result will be more highly constrained and thus fewer tuples will be retrieved.

```
find_column(main, [design, pwrmdl, encl, hp, avolts, baserpm, fwrpm,
                  winding, frame, duty, ovld_pct, ovld_time, fvolts,
                  service_fact, altitude, cost, motor_type]).
```

Figure 3.6: A fact given to Prolog which specifies the names of the attributes in the *main* relation.

This idea also applies to the parts of the query's search condition which are not derived from comparison goals. The equality of attributes from two different relations indicates a join. Although for a tuple in one relation there may be several tuples in the result, the overall effect is to constrain the number of tuples retrieved. Of course, this is only possible because all of the *D* goals, and thus multiple relations are used in formulating a single query. An additional advantage of this is that contemporary DBMS's have sophisticated methods for performing joins which efficiently access the information in secondary memory. There are also techniques for making the join operation efficient. For example, a relation can be stored sorted on the values of a particular attribute. A join on such an attribute is much more efficient than a join which does not use such a special technique.

3.4 Implementation and Empirical Results

MOSES2 has been implemented in Quintus Prolog on a SUN workstation. The entire rule set for this application is given in Appendix A. Ingres was used as the DBMS for MOSES2. Part of the interface to Ingres was written in C as was most of the user interface of MOSES2. MOSES2 and Ingres are executed as

a pair of concurrent processes. Whenever a DBMS query is formulated, it is passed to Ingres through a UNIX pipe. The result is put into a UNIX file which is then read and stored in the PDB.

Figure 3.5 is an example of a DBMS query formulated by our method. But it is only a part of what is needed. Such a query will put the answer on the screen instead of passing it to Prolog. In addition to the query, the actual command sent to Ingres also contains a bookkeeping part which takes care of putting the answer into a UNIX file so that Prolog is able to read it. It includes creating a temporary Ingres table, inserting the answer into that table and copying it to a file.

There is also a rule preprocessor. It is for the purpose of efficiency and convenience. The main function of the preprocessor is to formulate a partial DB query which only depends on the rules, not the run time instantiations. Efficiency is gained by doing this part of work only once which would otherwise be repeated for each application of the rules. One example of such part of query is a *select* list. It lists names of all the attributes to be selected. Attribute names can be found in the data dictionary. Therefore this list of names can be built from the information in the rules and the data dictionary before the running of the rules. The data dictionary, as was said above, is stored in the PDB and available to the preprocessor. The bookkeeping command mentioned above is also formulated by the preprocessor since it does not depend on the run time instantiation either.

The input of the preprocessor is the original rules, while the output of it is a new set of rules which will be run by the meta-interpreter. It is not precisely correct when we said above that the preprocessor formulates bookkeeping commands and part of DB queries. What the preprocessor actually does is inserting Prolog atoms into the original rules. The job of these new atoms is to formulate the bookkeeping commands and part of the DB queries. So when the preprocessed rule set is executed, the bookkeeping commands and the part of the DB queries are actually formulated for the meta-interpreter.

The preprocessor also collects information for the meta-interpreter. Remember that the meta-interpreter applies special processing to those rules that contain predicates which denote DB relations. At run time the meta-interpreter has to know whether a goal will eventually lead to a DB access which deserves such special processing. This information is extracted from the rule set by an analysis of the rules performed by the preprocessor. This increases the efficiency of the meta-interpreter which uses Prolog to evaluate all of the subgoals that can never lead to a DB access.

The implementation of MOSES2 is completed except for the explanation facility. We have run MOSES2 on two different DB's; one with data about 100 motors and another about 7,000 motors. This data was obtained from Reliance Electric and consist only of motors with older designs; newer designs were omitted for proprietary reasons. For each DB we recorded the time taken by the Expert System (ES) part and the time taken by the DBMS part. The sum of the two is the total time taken by one test run. The empirical results are given in

Figure 3.6, Figure 3.7 and Figure 3.8. Due to the nature of the rules and the current stage of testing, it turns out that one DB query always consists of one join and one selection plus some auxiliary SQL command such as creating a temporary relation, copying that relation into an UNIX file, etc.

No. of queries	Motors selected	ES time(sec.)	DBMS time(sec.)
1	2	4.1	5.7
1	10	5.5	6.0
2	3	6.3	10.6
2	16	8.4	11.9
3	3	8.0	15.1
3	22	11.6	15.8

Figure 3.6: Empirical results for a DB with 100 motors.

No of queries	Motors selected	ES time(sec.)	DBMS time(sec.)
1	2	4.1	12.6
1	12	5.7	25.7
2	4	6.1	23.9
2	21	9.1	46.2
3	5	8.2	33.7
3	27	12.0	62.8

Figure 3.7: Empirical results for a DB with 7000 motors.
(with heap file structure)

The empirical results indicate that the more tuples retrieved from the DB, the more time is used by the ES part. The explanation for this is that consider-

No. of queries	Motors selected	ES time(sec.)	DBMS time(sec.)
1	2	4.0	6.0
1	12	5.3	6.2
2	4	6.3	10.6
2	21	8.9	11.2
3	5	8.5	15.5
3	27	12.0	16.2

Figure 3.8: Empirical results for a DB with 7000 motors.
(with ISAM file structure)

able amount of time is spent on storing the answer of a DBMS query into the PDB and processing them. This again emphasizes the importance of making the query selective. Figure 3.7 shows that for a database of 7000 motors with heap file structure, a large amount of time is consumed on the DB retrieval, especially when more tuples are retrieved. Great improvement is gained by adopting the ISAM file structure. Notice that for the database of 7000 motors with the ISAM file structure (Figure 3.8), the time taken by DBMS does not increase significantly when compared with the database of 100 motors (Figure 3.6). The reason is that the ISAM file structure makes the join operation much faster.

The explanation facility is being implemented as a separate project and thus is not part of this thesis. However, an important part of this thesis is that the design of MOSES2 has provided for adding an explanation facility. The remainder of the thesis is concerned with the design of a more flexible inference engine for expert systems like MOSES2.

4 Graph Search

The description of the new inference engine that uses a graph search is given in this chapter. We will call this engine GSI (graph search interpreter). The first section describes the advantages of GSI. The formal description is given in the last section of this chapter. But before that is an informal discussion of several issues so that the reader can see the basic ideas behind the formal picture.

4.1 Justification

The expert system described in the previous chapter appears to be a reasonably good solution for its application. But its implementation raised several issues. As is stated before, these issues are not specific to the particular application that MOSES2 deals with. They are generic to the evaluation mechanism of Prolog and the difference between Prolog and a DBMS. Even though various techniques have been used to deal with these issues in the implementation of MOSES2, it is desirable to deal with them at a more conceptual level. This section describes these issues and how GSI deals with them.

During the execution of MOSES2, when data from the external database is needed the meta-interpreter takes charge to access the database. After the answers from database are returned, meta-interpreter stores them into Prolog database by using "assert". "Assert" is the only way to store the data in the Prolog database. The reason we need to store these answers is that the DBMS returns all the answers at one time but Prolog only uses them one at a time. This

problem is not difficult to solve, but it requires special processing by the meta-interpreter.

GSI reduces this special treatment because saving answers is a built-in feature of it. The answers for any subgoal are saved by the system, and thus the answers from the DBMS are treated in the same way as answers produced by normal evaluation.

The "find-table" is a programming technique used in MOSES2 to store answers to certain subgoals in the Prolog database (PDB). For example, there are several rules which need the value of line speed which is an application parameter specified by the customer. If we do not save the value of it, each of these rules will ask the user about line speed separately. That is, the user is asked the same question many times, which is absolutely unacceptable. Some other values which are not supplied by the customer are also stored in the find-table so that their values do not have to be recomputed.

The main disadvantage of the find-table technique is that it makes rules less declarative because rules must know what is stored in the find-table and must specify when it is accessed and changed. The rules contain special subgoals to do these operations which cause rules to be more imperative. This places an added burden on the rule designer and makes the rules much more difficult to understand.

In Prolog there are two alternatives to the find-table. First is simply re-evaluating the data whenever needed. This is what we want to avoid by using

find-table because it means asking the user the same question many times when the answer is provided by the user. Another alternative is parameter passing, i.e. using variables in rules to pass data from where it is generated to where it is needed. In a practical system like MOSES2, this technique makes the rules very unnatural. Figure 4.1 is a subset of rules from MOSES2 showed in Appendix A. We rewrite this part of the rules using parameter passing instead of the find-table. The result is shown in Figure 4.2. It can be seen that the parameter list of a predicate is so long that the rules become unreadable because essentially every thing that is stored in the find-table is a parameter to all rules.

```

main_goal :- init, getinput, calculate, retrieve, endw.

calculate :- rpm_min, rpm_max, hp_tension, hp_decel, hp_min, fw_rpm_low,
             hp_low, hp_high, motor_type.

retrieve :- finds(substitution, Sub), Sub \== yes, finds(motor_type, Mt),
            report_query(Mt, Reply), user_choose(Mt, Reply).
retrieve :- finds(substitution, Sub), Sub == yes,
            report_query(substitution, Reply), user_choose(substitution, Reply).

rpm_min :- finds(line_speed, Ls), finds(diameter_max, Dx),
           Ri is Ls/(Dx*3.1416), putin(rpm_min, Ri).

motors1(Design, Hp, Base_rpm, Fw_rpm, Winding, Motor_type, Cost, Mult_field,
        Tsel, Tdec)
:-
    ...,
    finds(hp_low, Hl), finds(hp_high, Hh), finds(base_rpm_low, Brl),
    finds(fw_rpm_low, Frl), finds(fw_rpm_high, Frh),
    finds(motor_type, Motor_type), finds(enclosure, Enclosure),
    finds(rpm_min, Ri), finds(remark, Remark),
    main(Design, _, Enclosure, Hp, Avolts, Base_rpm, Fw_rpm, Winding, Frame,
        Duty, Ovld_pct, Ovld_time, Fvolts, Service_factor, Altitude,
        Cost, Motor_type),
    field(Design, _, _, Tsel, Tdec, _, Mult_field),
    Hl=<Hp, Hp=<Hh, Brl=<Base_rpm, Frl=<Fw_rpm, Fw_rpm=<Frh,
    Base_rpm =< Ri*Hp/Hl ,
    impose_remarks(Design, Remark, _ ).

```

Figure 4.1: A Subset of The Rules from MOSES2

```

main_goal :- init, getinput(Motor_type, Line_speed, Diameter_max,
    Diameter_min, Tension, Paper_inert, Steel_inert, Stop_time),
    calculate(Motor_type, Line_speed, Diameter_max, Diameter_min, Tension,
        Paper_inert, Steel_inert, Stop_time, Rpm_min, Rpm_max, Hp_tension,
        Hp_decel, Hp_min, Fw_rpm_low, Hp_low, Hp_high),
    Substitution = no,
    retrieve(Substitution, Motor_type, Rpm_min, Rpm_max, Hp_tension,
        Hp_decel, Hp_min, Fw_rpm_low, Hp_low, Hp_high), endw.

calculate(Motor_type, Line_speed, Diameter_max, Diameter_min, Tension,
    Paper_inert, Steel_inert, Stop_time, Rpm_min, Rpm_max, Hp_tension,
    Hp_decel, Hp_min, Fw_rpm_low, Hp_low, Hp_high)
:- motor_type(Motor_type), rpm_min(Line_speed, Diameter_max, Rpm_min),
    rpm_max(Line_speed, Diameter_min, Rpm_max),
    hp_tension(Line_speed, Tension, Hp_tension),
    hp_decel(Paper_inert, Steel_inert, Line_speed, Stop_time, Diameter_max,
        Rpm_min, Hp_decel),
    hp_min(Hp_tension, Hp_decel, Hp_min),
    fw_rpm_low(Rpm_max, Fw_rpm_low),
    hp_low(Hp_min, Hp_low), hp_high(Hp_min, Hp_high).

retrieve(Substitution, Motor_type, Rpm_min, Rpm_max, Hp_tension, Hp_decel,
    Hp_min, Fw_rpm_low, Hp_low, Hp_high)
:- report_query(Substitution, Motor_type, Rpm_min, Rpm_max, Hp_tension,
    Hp_decel, Hp_min, Fw_rpm_low, Hp_low, Hp_high, Reply),
    user_choose(Substitution, Motor_type, Rpm_min, Rpm_max, Hp_tension,
        Hp_decel, Hp_min, Fw_rpm_low, Hp_low, Hp_high, Reply).

rpm_min(Line_speed, Diameter_max, Rpm_min)
:- Rpm_min is Line_speed/(Diameter_max*3.1416).

motors1(Hp_low, Hp_high, Base_rpm_low, Fw_rpm_low, Fw_rpm_high,
    Motor_type, Enclosure, Rpm_min, Remarks,
    Design, Hp, Base_rpm, Fw_rpm, Winding, Cost, Mult_field, Tsel, Tdec)
:- main(Design, _, Enclosure, Hp, Avolts, Base_rpm, Fw_rpm, Winding,
    Frame, Duty, Ovld_pct, Ovld_time, Fvolts, Service_factor,
    Altitude, Cost, Motor_type),
    field(Design, _____, Tsel, Tdec, _____, Mult_field),
    Hp_low=<Hp, Hp=<Hp_high, Base_rpm_low=<Base_rpm,
    Fw_rpm_low=<Fw_rpm, Fw_rpm=<Fw_rpm_high,
    Base_rpm =< Rpm_min*Hp/Hp_low,
    impose_remarks(Design, Remark, _ ).

```

Figure 4.2: A Subset of Rules with parameter passing

The find-table disappears when a GSI is employed. There is no re-evaluating subgoals because the answers are saved right after they are first generated. Thus, the user will not be asked the same question twice. Appendix B gives the new rules which assume the use of GSI. These rules are a rewritten form of the rules of MOSES2 given in Appendix A. They do not suffer from long parameter list as the rules in Figure 4.2 do, and they do not use special subgoals to store and retrieve data. Take line speed as an example. When it is needed, *line_speed(Ls)* is called. The rule about line speed says to ask the user for the value. The first time this rule is executed the user is asked and the answer is saved. Later calls for line speed will retrieve the saved answer instead of asking the user again. The rules are more natural and logical than the rules with find-table in Appendix A because they contain nothing for accessing or updating the find-table, e.g. instead of *find(line_speed, Ls)*, *line_speed(Ls)* is used when the line speed is needed. Also, the rule for computing rpm_min in Appendix A does not have variables on the left-hand side, which seemed strange. But this anomaly does not occur in the new rules in Appendix B.

There are some additional advantages of a graph search in logic programming. One thing is that it helps the explanation facility. Explanations are a very important aspect of expert systems. But to be able to do it, generally you have to save the history of evaluation. In Prolog, an explanation facility can use a meta-interpreter to save the evaluation history and use it to answer user's questions. Not only does this mean an extra meta-interpreter but also a special caution has to be taken for the find-table because when an old value was replaced by a new

value, the old value should be saved as part of the history. In our system all the answers, past or current, are saved. They are part of the evaluation history. With some modification it will be able to provide all the information needed by the explanation facility. Thus, the graph search greatly simplifies the process of generating explanations.

An additional advantage is the capability of reasoning in some cases in which Prolog would not terminate. One of such cases is left recursion. Given the rules in Figure 4.3 and a goal $?- \text{ancestor}(X,Y)$, Prolog will run into an infinite recursion but our system will not. The point is that Prolog keeps recursively calling $\text{ancestor}(X,Y)$ while our system stops at first recursive call and refers to the original call for $\text{ancestor}(X,Y)$ since they are the same. Another such case is a data cycle. If we add one more fact $p(c,a)$ to Figure 4.3 we then have a data cycle. Prolog will keep producing duplicate answers without stopping. But GSI checks for duplicate subgoals and duplicate answers since both are saved in memory. Therefore such a program will terminate in this case and in many other cases too. In fact GSI terminates on any program on which OLDT-resolution terminates since it can do every thing that OLDT-resolution can. The programs are the "programs defining finite relations only". This is true for an arbitrary search rule.

Obviously all the above advantages do not come free. The major cost is memory: all subgoals and their answers must be saved in memory and some of these may not be needed anywhere else in the evaluation.

```

ancestor(X, Y) ← p(X, Y).
ancestor(X, Y) ← ancestor(X, Z), p(Z, Y).

p(a, b).
p(b, c).

```

Figure 4.3: Left recursive rules

4.2 Basic Idea

In this section we discuss the different aspects of our GSI in an intuitive way to give the reader some idea of how GSI works. But first some terminology is introduced.

We adopt the terminology of logic programming [StSh86, Lloy84]. A *program* is composed of *rules*. A rule has a *left-hand side* (LHS) and a *right-hand side* (RHS). The LHS of a rule contains at most one atom while the RHS of it may contain any number of atoms. An *atom* is a predicate with its argument list, such as $p(X, f(Y))$. A rule with empty RHS is called a *fact* which is a special case of a rule. A set of atoms is called a *goal* which means a query. An atom in a goal is called a *subgoal*. Rules, facts and goals are also called *Horn clauses*, or *clauses* for short. Given a program and a goal, a subgoal in the goal and a rule in the program can be selected to produce a *resolvent* which itself is a new goal. The same thing can be done to this new goal and get a new resolvent. A sequence of resolvent thus obtained is called an *SLD-derivation*. An *SLD-refutation* is a finite SLD-derivation which has the empty clause as the last goal in the derivation. An SLD-refutation is a successful evaluation of the original

goal and produces a non-empty set of answers which are values substituted for variables in the original goal.

4.2.1 Essential Issues in a Graph Search

This discussion makes use of the following definitions. The *ground instances* of an atom q is the set of all instances of q which contain no variables. This is denoted by $G(q)$. Given a program P , the *answers* of atom q are all of its ground instances which are logical consequences of P . This is denoted by $A(q)$. Obviously $A(q) \subseteq G(q)$. If q_1 and q_2 are two atoms, then $G(q_1) \subseteq G(q_2)$ implies $A(q_1) \subseteq A(q_2)$. The argument is as follows. $G(q_1) \subseteq G(q_2)$ implies that q_1 is an (possibly non-ground) instance of q_2 . Therefore what makes q_1 true must also makes q_2 true, which in fact means $A(q_1) \subseteq A(q_2)$.

The main feature of GSI is saving the answers of a subgoal and using them in solving similar goals. For example, the answers for $p(X, Y)$ can be used in finding answers for $p(a, Z)$. Here we have $G(p(a, Z)) \subseteq G(p(X, Y))$, which implies $A(p(a, Z)) \subseteq A(p(X, Y))$. Therefore any answers for $p(X, Y)$ that unifies with $p(a, Z)$ is an answer for $p(a, Z)$, and all answers for $p(a, Z)$ can be obtained in this way. A more complicated example is when you have the answers for $p(f(X), Y)$ and you need the answers for $p(W, b)$. Now $G(p(W, b))$ is not a subset of $G(p(f(X), Y))$. But they do intersect since they are unifiable. Actually $G(p(f(X), Y)) \cap G(p(W, b)) = G(p(f(X), b))$, where $p(f(X), b)$ is their most general common instance. This means that it is possible that some answers of $p(f(X), Y)$ are also answers for $p(W, b)$. Those answers are actually from the set

$A(p(f(X), b))$. The word "possible" is used because the set $A(p(f(X), b))$ might be empty. In such case there is no answer of $p(f(X), Y)$ that is also the answer for $p(W, b)$. But generally we have no way of knowing beforehand if this set is empty or not. As long as $p(f(X), Y)$ and $p(W, b)$ are unifiable, i.e., the intersection of $G(p(f(X), Y))$ and $G(p(W, b))$ is not empty, we can look through the answers for $p(f(X), Y)$ and pick up those that are answers of $p(W, b)$ by unification as we described above. This will give us all the answers of $p(W, b)$ that are also answers of $p(f(X), Y)$. Effectively, unification of atoms computes the intersection of their sets of ground instances.

Besides this we need to find all the rest answers; i.e. $A(p(W, b)) - A(p(f(X), Y))$, the answers of $p(W, b)$ which are not answers of $p(f(X), Y)$. Here we need the difference between their sets of answers. There is no efficient operation on substitutions, like unification, for computing this set difference. Therefore we use normal subgoals augmented with constraints to represent such set differences. For this example, $p(W, b) \& \{W \neq f(X)\}$ is used, where $\{W \neq f(X)\}$ is a constraint. It can simply read as find all the answers of $p(W, b)$ such that W is not instantiated with an instance of $f(X)$. Such answers are exactly the set $A(p(W, b)) - A(p(f(X), Y))$. Note that the variable W and X play different roles in this constraint. W is a free variable while X is universally quantified since $\{W \neq f(X)\}$ in this case means for any X , W is not equal to $f(X)$. Now what we need is a systematic way of constructing the constraints and deriving the resolvent from a rule and a subgoal with constraint, which is described in the next section.

Here is another example that needs a more complicated constraint. Suppose the answers for $p(f(X), a)$ have been saved and the answers for $p(U, W)$ are needed. The set $A(p(U, W)) - A(p(f(X), a))$ is then represented as $p(U, W) \& \{U \neq f(X), W \neq a\}$. This denotes the answers of $p(U, W)$ such that either U is not instantiated with an instance of $f(X)$ or W is not instantiated with a . Note that the relationship between the inequations $U \neq f(X)$ and $W \neq a$ is disjunction since either of them being true is enough to ensure the answers is in the intended set.

The search rule and the computation rule constitute the control of the search process. We decide to make these rules parameters to the search process. This avoids committing to any particular search strategy; instead it lets it be a function of the application. For example, one may adopt breadth first for the search rule to insure getting all the answers by sacrificing efficiency when there is not much knowledge about the run time characteristics of the rules.

4.2.2 The Constraints

A constraint is obtained by looking into the most general unifier (mgu) of two atoms. Suppose the answers for $p(U, W, W)$ are known and the answers for $p(a, X, Y)$ are needed. After picking up the answers for $p(a, X, Y)$ from the answers of $p(U, W, W)$, the rest of the answers are $A(p(a, X, Y)) - A(p(U, W, W))$ which is represented as $p(a, X, Y) \& \{X \neq Y\}$. The constraint $\{X \neq Y\}$ is obtained from the mgu for $p(a, X, Y)$ and $p(U, W, W)$, which is $\sigma = \{U/a, X/Y, W/Y\}$. The reasoning is as follows. $A(p(a, X, Y)\sigma)$ are those answers of $p(a, X, Y)$ which are also answers of $p(U, W, W)$ since σ unifies $p(a, X, Y)$ and $p(U, W, W)$.

W): $A(p(a, X, Y) \& \{U \neq a, X \neq Y, W \neq Y\})$ are those answers of $p(a, X, Y)$ which are not answers of $p(U, W, W)$. It is exactly the set $A(p(a, X, Y)) - A(p(U, W, W))$. But this can be simplified because $U \neq a$ and $W \neq Y$ do not really constrain $p(a, X, Y)$. The simplified version is $p(a, X, Y) \& \{X \neq Y\}$ because it is logically equivalent to $p(a, X, Y) \& \{U \neq a, X \neq Y, W \neq Y\}$ and thus also represents the set $A(p(a, X, Y)) - A(p(U, W, W))$.

The example above contain only one disjunction of inequations. Let such a disjunction be called a *constraint*. There may be a set of several constraints attached to a subgoal. The relationship among those constraints is conjunction. The general form of a subgoal with constraints is $p \& c_1 \& c_2 \& \dots \& c_n$, where c_i is a constraint, a disjunction of inequations. To show the reason why more than one constraint might be needed, let us take the above example and add one more thing to it: there is another subgoal $p(a, f(Z), V)$ whose answers are also saved. Obviously some answers of $p(a, X, Y)$ are in the set $A(p(a, f(Z), V))$. Thus part of the answers of $p(a, X, Y)$ can be picked up from $A(p(U, W, W))$; and part of the answers of $p(a, X, Y)$ can be picked up from $A(p(a, f(Z), V))$. The rest of the answers of $p(a, X, Y)$ are those which are neither the answers of $p(U, W, W)$ nor the answers of $p(a, f(Z), V)$. The set $A(p(a, X, Y)) - A(p(U, W, W))$ is represented as $p(a, X, Y) \& \{X \neq Y\}$ as described above. The set $A(p(a, X, Y)) - A(p(a, f(Z), V))$ can be represented as $p(a, X, Y) \& \{X \neq f(Z)\}$, where $\{X \neq f(Z)\}$ is obtained from the mgu for $p(a, X, Y)$ and $p(a, f(Z), V)$. Therefore the answers of $p(a, X, Y)$ which are neither answers of $p(U, W, W)$ nor answers of $p(a, f(Z), V)$ can be represented as $p(a, X, Y) \& \{X \neq Y\} \& \{X \neq f(Z)\}$.

A new way of deriving a resolvent is needed since subgoals are now coupled with constraints. The normal resolution process in logic programming can still be used. But some processing has to be added to deal with the constraint. Suppose there is a subgoal with constraint $p(X, a, Z) \& \{X \neq a, Z \neq f(a)\}$ and a rule $p(a, U, f(W)) \leftarrow q1(a, U), q2(U, f(W))$. The normal resolution process without considering the constraint will give out the resolvent $q1(a, a), q2(a, f(W))$. The mgu is $\sigma = \{X/a, U/a, Z/f(W)\}$. To process the constraint, the mgu is applied to the constraint; $\{X \neq a, Z \neq f(a)\}\sigma = \{a \neq a, f(W) \neq f(a)\}$. But $\{a \neq a, f(W) \neq f(a)\}$ is equal to $\{W \neq a\}$ since $a \neq a$ is false and $f(W) \neq f(a)$ is equal to $W \neq a$. The false inequation is removed due to the disjunction relationship between the inequations. So the new constraint $\{W \neq a\}$ is attached to each subgoal in the resolvent, which yields $q1(a, a), q2(a, f(W)) \& \{W \neq a\}$. Note that the constraint $\{W \neq a\}$ need not be attached to the subgoal $q1(a, a)$.

To show the other cases let us take the above example and make some changes. Suppose the subgoal remain unchanged but the rule is changed to $p(b, U, f(W)) \leftarrow q1(b, U), q2(U, f(W))$. The new mgu is $\sigma = \{X/b, U/a, Z/f(W)\}$. Applying it to the constraint gives $\{b \neq a, f(W) \neq f(a)\}$. But $b \neq a$ makes the whole constraint true. Therefore the resolvent will have no constraint attached to it. For the next case, let us change the rule again to $p(a, U, f(a)) \leftarrow q1(a, U), q2(U, f(a))$. This time the mgu is $\sigma = \{X/a, U/a, Z/f(a)\}$. The result of applying σ to the constraint is $\{a \neq a, f(a) \neq f(a)\}$, which is a failure because of $a \neq a$. This means there is no match with this rule because the constraint is violated and thus no resolvent is produced.

For the general case where there is a set of constraints, each of the constraints are processed separately in the way described above. Any such constraint which becomes true is removed from the conjunction. The whole constraints is a failure if any constraint in the conjunction becomes a failure.

4.3 The Graph Search Algorithms

In this section we describe our graph search algorithms of GSI. It uses three different methods which do basic manipulations of constraints. These methods are described in the next 3 subsections. This is followed by a description of the algorithm. We need the following terminology. Let p be a term. The variable set of p , $\text{var}(p)$, is the set of all the variable that appear in p . i.e., $\text{var}(p) = \{X \mid X \text{ is a variable} \wedge X \text{ appears in } p\}$

4.3.1 Constraint Generation

The algorithm for constraint generation creates a constraint which represents the "set difference" of two atoms.

Algorithm for constraint generation $\text{GenConst}(p1, p2)$.

Input: Atom $p1$, atom $p2$ such that $\text{var}(p1) \cap \text{var}(p2) = \emptyset$.

Output: The constraint for $G(p1) - G(p2)$ or false. (Recall that $G(p1)$ denotes the set of ground instances of $p1$.)

1. Compute a most general unifier (mgu), σ , of $p1$ and $p2$.
2. Exit with constraint $C = \{ X \neq t \mid X/t \in \sigma \wedge X \in \text{var}(p1) \}$. If C is empty, exit

with false. $\square$

Note that the order of p_1 and p_2 is important to this algorithm. Also note that for the case where p_1 and p_2 are exactly the same, the mgu for them is considered an empty one, therefore the constraint is also an empty constraint which is false. All the cases which lead to this algorithm to produce false are the cases where $G(p_1)$ is a subset of $G(p_2)$. There are some cases where $G(p_1)$ is a subset of $G(p_2)$ but the constraint produced by this algorithm is not empty as it should be, for instance, the case where p_1 and p_2 are alphabetic variant to each other. Such cases will be taken care of by the normalization algorithm introduced later.

Example: Consider two atoms $p(X, b)$ and $p(f(W), Z)$. One of the mgu's for them is $\{X/f(W), Z/b\}$. The algorithm for constraint generation produces the constraint $\{X \neq f(W)\}$ since X is a member of $\text{var}(p(X, b))$ but Z is not. This constraint represents $G(p(X, b)) - G(p(f(W), Z))$ because any instantiation of $p(X, b)$ which satisfies $X \neq f(W)$ is in this set.

Example: $\text{GenConst}(p(X, b), p(U, V)) = \text{false}$ because $\text{mgu}(p(X, b), p(U, V)) = \{U/X, V/b\}$ and the corresponding constraint is empty since neither U nor V is a member of $\text{var}(p(X, b))$.

Suppose C is the constraint produced by the algorithm for constraint generation with input p_1, p_2 . If p_1 is considered representing all the answers of p_1 (with respect to particular program), then $p_1 \& C$ represents those answers of p_1 which are not answers of p_2 . In the above example, $p(X, b) \& \{X \neq f(W)\}$ represent those answers of $p(X, b)$ which are not answers of $p(f(W), Z)$. Note

that the variable W in the constraint does not appear in $p(X, b)$ which it "constrains". The actual meaning of the constraint is $(\forall W)\{X \neq f(W)\}$. Obviously there is no way to tell which variable is universally quantified from a constraint alone. The universally quantified variables are given explicitly or they are identified when the constraint is associated with a particular atom.

We assume that the mgu used has the following property: For any $X/t, Y/s$ in the mgu, $X \notin \text{var}(s)$; i.e., no left-hand side variable appears in any right-hand side terms. If this were the case, the variable in the right-hand side term can be renamed.

4.3.2 Application of Substitution to a Constraint

The algorithm for application of substitution to a constraint applies a substitution to a constraint and simplifies the result.

Algorithm for application of substitution to a constraint $\text{SubApp}(C, \sigma)$.

Input: A constraint C and a substitution σ .

Output: True, false or a new constraint C' which results from applying σ to C .

1. $C' := C\sigma$.
2. Repeat the following substeps which modify C' until no further modifications are possible.
 - 2.1 select one inequation $t \neq s$ from C' .
 - 2.2 If $t = f(u_1, \dots, u_n)$, and $s = f(v_1, \dots, v_n)$, then remove $t \neq s$ from C' , for $i = 1, \dots, n$; put $u_i \neq v_i$ into C' .

- 2.3 If $t=f(\dots)$, and $s=g(\dots)$, then exit with true.
 - 2.4 If t (or s) is a variable, and t and s are not identical symbolically, then do nothing (leave $t \neq s$ in C').
 - 2.5 If $t=s=X$, where X is a variable, or $t=s=a$, where a is a constant, then remove $t \neq s$ from C' .
 - 2.6 If none of above conditions is true, $t \neq s$ must be of the form $a \neq b$ or $a \neq f(\dots)$, where a, b are constants and f is a functional symbol, then exit with true.
3. If $C' = \emptyset$, then exit with false; otherwise exit with the new constraint C' . $\square$

Example: Given a constraint $C = \{X \neq Y, Z \neq f(b)\}$ and a substitution $\sigma = \{X/a, Z/f(U)\}$, the algorithm process the constraint in the following steps: $C\sigma = \{a \neq Y, f(U) \neq f(b)\} \rightarrow \{a \neq Y, U \neq b\}$. It exits with the new constraint $\{a \neq Y, U \neq b\}$.

Example: Given the same constraint as in above example and a substitution $\{X/a, Z/f(a)\}$, the algorithms goes as: $c\sigma = \{a \neq Y, f(a) \neq f(b)\} \rightarrow \{a \neq Y, a \neq b\} \rightarrow \text{true}$.

The algorithm for application of substitution to a constraint is based on Colmerauer's "reduction algorithm" in [Colm84] because this kind of simplification is useful for our purposes. The other algorithms are different from his work since they are specifically designed for our new resolution method. In SLD-resolution, when the selected subgoal of a goal is unifiable with the head of a rule, the mgu of the two is then applied to the rest subgoals in the goal and the subgoals in the body of the rule. In our case, besides applying the mug to the subgoals, the mgu must also be applied to the constraints. Generally there is a

set of constraints attached to a subgoal. This algorithm is used to apply the substitution to such constraints and simplify the result.

4.3.3 Normalization

A constraint is said to be in a *normal form* if

1. All the left-hand side of the inequations are distinct variables and
2. No left-hand side variable appears in the right-hand side terms of any inequations and
3. No universally quantified variable appears in any inequations as the entire left or right-hand side.

Universally quantified variables of a constraint can be found when the constraint is attached to a subgoal. They are those variables of the constraint which have no occurrence in the subgoal. e.g., W is a universally quantified variable in $p(X, Y) \& \{X \neq W, Y \neq W\}$ since W does not appear in $p(X, Y)$.

The following algorithm is used to put constraints into normal form.

Algorithm for normalization Norm(C, L).

Input: A constraint C and a list L of universally quantified variables in C .

Output: A logically equivalent constraint in a normal form, or, true or false.

1. $C' := C$.
2. Repeat the following substeps which modify C' until nothing can be selected at step 2.3.

- 2.1 $C' := \text{SubApp}(C', \sigma)$, where σ is an empty substitution. (This just simplifies C'). If C' is true, exit with true. If C' is false, exit with false. Otherwise, C' is the resulting constraint.
- 2.2 Modify C' by switching the left-hand side with the right-hand side of the inequations in the C' when necessary so that the left-hand side of all the inequations are variables.
- 2.3 Select one inequation $X \neq t$ from C' such that t is not a variable and X occurs in other inequations in C' . If $X \in \text{var}(t)$, exit with true. Otherwise substitute t for all the other appearances of X in C' .
3. Find all the inequations in C' such that both sides of them are variables. Draw a connection graph among these inequations by connecting any two inequations where there is a common variable between the two. Remove all these inequations from C' .
 - 3.1 For each connected subgraph,

Chose arbitrarily one variable X from the inequations in the subgraph. Free variables have higher priority than universally quantified variables in L .

For each variable Y other than X from the inequations in the subgraph,

Put $Y \neq X$ in the constraint C' .
4. For each newly added inequation $Y \neq X$ in step 3, substitute X for all the other appearances of Y in C' .
5. Remove all the inequations $X \neq t$ from C' if X is an universally quantified variable in L .

6. Exit with new constraint C' if C' is not empty. Otherwise exit with false. $\square$

Example: Given a constraint $\{h(X) \neq h(g(V)), f(X, Z) \neq Y, V \neq W, W \neq U\}$ with W as an universally quantified variable, the algorithm for normalization produce the following intermediate constraints step by step and gives the final constraint upon termination.

1. $C' = \{h(X) \neq h(g(V)), f(X, Z) \neq Y, V \neq W, W \neq U\}$
- 2.1 $C' = \{X \neq g(V), f(X, Z) \neq Y, V \neq W, W \neq U\}$
- 2.2 $C' = \{X \neq g(V), Y \neq f(X, Z), V \neq W, W \neq U\}$
- 2.3 $C' = \{X \neq g(V), Y \neq f(g(V), Z), V \neq W, W \neq U\}$
3. $C' = \{X \neq g(V), Y \neq f(g(V), Z), V \neq U, W \neq U\}$
4. $C' = \{X \neq g(U), Y \neq f(g(U), Z), V \neq U, W \neq U\}$
5. $C' = \{X \neq g(U), Y \neq f(g(U), Z), V \neq U\}$

Example: Consider the second example in the section 4.3.1 which generate the constraint for $p(X, b)$ and $p(U, V)$. The result there was $\text{GenConst}(p(X, b), p(U, V)) = \text{false}$. The mgu used is $\{U/X, V/b\}$. But $\{X/U, V/b\}$ is also a mgu for $p(X, b)$ and $p(U, V)$. The corresponding constraint from this mgu is not empty. It is $\{X \neq U\}$. Fortunately this normalization algorithm helps to get consistent answer since $\text{Norm}(\{X \neq U\}, \{U\}) = \text{false}$. i.e., the normal form of $\{X \neq U\}$ is false if U is universally quantified.

A constraint is a logical expression in which some variables are universally quantified variables. The remaining variables are free. Also recall that disjunction is the logical relationship among the inequations in the constraint. It makes

sense to talk about the logical equivalence of constraints. It is proved later that the algorithm for normalization produces a constraint which is logically equivalent to its input.

Putting a constraint into a normal form has the following benefits: (1)it gives a simplified form of the constraint; (2)it determines if the constraint is already a tautology or unsatisfiable; (3)the normalized constraint can be transformed into a substitution by simply changing $\neq$ to $=$, which is a necessary feature for our new resolution as will be clarified later.

As was pointed out by Prof. Leon Sterling and Prof. Z. Meral Ozsoyoglu, there is similarity between our constraint and the *equation set* in [LaMM87]. The equation set is a conjunction of equations while our constraint is a disjunction of inequations. i.e., one is a negated form of the other. Some variables of a constraint may be universally quantified as we mentioned before, but an equation set has only free variables. The *solved form* of an equation set is just the negation of the normal form of the corresponding constraint except for the part relevant to universally quantified variables. Therefore, the normalization algorithm is similar to the *solved form algorithm* [LaMM87], but the normalization algorithm has to take care of the universally quantified variables. For example the normal form of the constraint $\{W \neq a\}$, where W is universally quantified, is empty, and thus unsatisfiable; but its corresponding equation set $\{W = a\}$ is already in solved form, which means it is solvable, and thus not tautology. As another example, the normal form of the constraint $\{X \neq W, Y \neq W\}$ is $\{X \neq Y\}$ if W is universally quantified; but the solved form of its corresponding equation set is $\{X = W,$

$Y=W\}$.

4.3.4 Application of Substitution to a Subgoal

The following algorithm describes how to apply a substitution to a subgoal. For GSI a subgoal is an atom with a set of constraints. Such a subgoal is reduced to a conventional subgoal, which is simply an atom, when the constraint set is empty. Therefore a subgoal under our definition is a generalization of a conventional subgoal. In the remainder of this dissertation a subgoal normally refer to this new kind of subgoal, but a conventional subgoal may still be called a subgoal where there is no ambiguity in context. Note that an empty constraint represents a false, but an empty set of constraints represents a true. The reason is quite natural: a constraint is a disjunction of inequations while a set of constraint is a conjunction of constraints.

Algorithm for application of substitution to a subgoal.

Input: A substitution σ and a subgoal $A = p \& c_1 \& \dots \& c_n$.

Output: A new subgoal or false.

1. $p' := p\sigma$. (conventional substitution is meant here)

2. For each i :

$c_i' := \text{Norm}(\text{SubApp}(c_i, \sigma), L)$, where $L = \{ X \mid X \in \text{var}(\text{SubApp}(c_i, \sigma)) \wedge X \notin \text{var}(p') \}$, i.e. L is the list of variables appearing in $\text{SubApp}(c_i, \sigma)$ but not appearing in p' .

3 If c_j' is false, for some j , exit with false. Otherwise let

$A' = p' \& c_1' \& \dots \& c_n'$; delete all of c_j' which is true from A' to form the output of the algorithm.

Notice that if the input subgoal is reduced to a conventional subgoal, this algorithm does a conventional substitution application to that conventional subgoal. By this argument, it is reasonable to use $A\sigma$ to denote a substitution application to a subgoal described by this algorithm; this notation will be used in the remainder of this chapter.

Also notice that the algorithm might result in false. The actual meaning of it is that the constraint is violated under the intended substitution and thus it can not be proven.

Example: Given a subgoal $A = p(X,Y,Z) \& \{X \neq Y, Z \neq f(b)\}$ and a substitution $\sigma = \{X/a, Z/f(U)\}$. $p(X,Y,Z)\sigma = p(a,Y,f(U))$, $\text{Norm}(\text{SubApp}(\{X \neq Y, Z \neq f(b)\}, \sigma), L) = \text{Norm}(\{a \neq Y, U \neq b\}, \{\}) = \{Y \neq a, U \neq b\}$, and $A\sigma = p(a,Y,f(U)) \& \{Y \neq a, U \neq b\}$. The algorithm exit with $p(a,Y,f(U)) \& \{Y \neq a, U \neq b\}$.

Example: Given a subgoal $A = p(X,Y) \& \{X \neq Y\}$ and a substitution $\sigma = \{X/a, Y/a\}$. $p(X,Y)\sigma = p(a,a)$, $\text{Norm}(\text{SubApp}(\{X \neq Y\}, \{X/a, Y/a\}), L) = \text{Norm}(\text{false}, L) = \text{false}$. The algorithm exits with false. Actually the constraint is violated because it asks $X \neq Y$ but the substitution instantiates both X and Y with a .

4.3.5 Method of Inference

We will now describe how to do the equivalent of resolution in GSI. Rules

(or clauses) will be the same as in logic programming. A goal consists of a set of subgoals which may have constraints.

A search tree, which will be called a GSI-tree, similar to the search tree in logic programming (called an SLD-tree in [Lloy84]) is used to describe the search space. A node corresponds to a goal. A child node of a parent node corresponds to a goal which is derived by one *derivation step* from its parent goal. A derivation step is somewhat similar to the process of obtaining a resolvent from a goal and a rule in logic programming. A *derivation step* in our schema is either a *rule derivation step*, *answer derivation step* or a *complement derivation step* which are described in the following sub-subsections.

4.3.5.1 Rule Derivation Steps

A rule derivation step produces a new goal (resolvent) from an old goal and a rule.

Let $R = p :- p_1, \dots, p_m$ be a rule, $G_1 = A_1, \dots, A_n$ be a goal consisting of n subgoals, and $A_i = q \& C$ be the subgoal selected by the computation rule, where q is an atom and C is a conjunction of constraints. We say there is a *rule derivation step* from G_1 to G_2 if the following condition hold:

- (a) p and q are unifiable with a mgu σ ;
- (b) $(q \& C)\sigma$ is not false;
- (c)

$$G_2 = A_1\sigma, \dots, A_{i-1}\sigma, (p_1 \& C)\sigma, \dots, (p_m \& C)\sigma, A_{i+1}\sigma, \dots, A_n\sigma;$$

(d) No $A_j\sigma$ nor $(p_k \& C)\sigma$ is false.

G_2 is called a *resolvent* of G_1 and R . We may also say that G_2 is the result of a rule derivation step from G_1 .

Example: Let $p(X,Y)\&\{X\neq a\}, q(Y,Z)\&\{Y\neq Z\}$ be a goal and $p(U,b):-r(U)$ be a rule. A mgu for $p(X,Y)$ and $p(U,b)$ is $\sigma=\{X/U, Y/b\}$. Applying σ to $r(U)\&\{X\neq a\}$ and $q(Y,Z)\&\{Y\neq Z\}$ results $r(U)\&\{U\neq a\}$ and $q(b,Z)\&\{Z\neq b\}$ respectively. Therefore the resolvent is $r(U)\&\{U\neq a\}, q(b,Z)\&\{Z\neq b\}$. i.e., there is a rule derivation step from goal $p(X,Y)\&\{X\neq a\}, q(Y,Z)\&\{Y\neq Z\}$ to goal $r(U)\&\{U\neq a\}, q(b,Z)\&\{Z\neq b\}$.

4.3.5.2 Answer Derivation Steps

An *answer derivation step* is designed for the case where a "similar" subgoal has been evaluated before and an answer computed for it is also an answer for the current subgoal. This process is like a rule derivation step in which the rule has no body. But there is a bit difference because a saved answer may have constraints attached to it. The answer derivation step produces a new goal from an old goal and an answer. The following example explains why an answer may have constraints. Suppose $p(X,Y)\&\{Y\neq b\}$ is a subgoal and $p(a,Y)$ is known to be true. $p(a,Y)$ then is an answer for $p(X,Y)$, but not for $p(X,Y)\&\{Y\neq b\}$. Instead, $p(a,Y)\&\{Y\neq b\}$ is the right answer to be saved for subgoal $p(X,Y)\&\{Y\neq b\}$.

Let $p \& C_p$ be an answer, $G_1 = A_1, \dots, A_n$ be a goal consisting of n subgoals, and let $A_i = q \& C$ be the subgoal selected by the computation rule, where q is an atom and C is a conjunction of constraints. We say there is an *answer derivation step* from G_1 to G_2 if the following condition hold:

- (a) p and q are unifiable with a mgu σ ;
- (b) $(q \& C \& C_p)\sigma$ is not false;
- (c) $G_2 = (A_1 \& C_p)\sigma, \dots, (A_{i-1} \& C_p)\sigma, (A_{i+1} \& C_p)\sigma, \dots, (A_n \& C_p)\sigma$;
- (d) no $(A_j \& C_p)\sigma$ is false.

For $A_k = p_k \& C_k$, the notation $A_k \& C_p$ in the above means $p_k \& C_k \& C_p$. G_2 is called a *resolvent* from goal G_1 and answer $p \& C_p$. We may also say that G_2 is the result of an answer derivation step from G_1 .

Example: Let $p(X,Y) \& \{X \neq a\}, q(Y,Z) \& \{Y \neq Z\}$ be a goal and $p(b,U) \& \{U \neq b\}$ be an answer. A mgu for $p(X,Y)$ and $p(b,U)$ is $\sigma = \{X/b, Y/U\}$. $(p(X,Y) \& \{X \neq a\} \& \{U \neq b\})\sigma = p(b,U) \& \{U \neq b\}$ is not false. Then $(q(Y,Z) \& \{Y \neq Z\} \& \{U \neq b\})\sigma = q(U,Z) \& \{U \neq Z\} \& \{U \neq b\}$ is the new goal resulting from an answer derivation step from the given goal $p(X,Y) \& \{X \neq a\}, q(Y,Z) \& \{Y \neq Z\}$.

4.3.5.3 Complement Derivation Steps

A complement derivation step is designed for the case where part of the answers from an *answer providing subgoal* is missing due to the use of constraints. The accurate definition for *answer providing subgoal* is given in the

next sub-subsection. The following is an example where a complement derivation step is needed. Suppose $p(a,Y)\&\{Y\neq b\}, q(Y,Z)\&\{Y\neq Z\}$ is a goal with $p(a,Y)\&\{Y\neq b\}$ being selected and $p(U,V)\{U\neq V\}$ is an answer providing subgoal for $p(a,Y)\&\{Y\neq b\}$. There is no doubt that some of the answers of $p(U,V)\{U\neq V\}$ are good for subgoal $p(a,Y)\&\{Y\neq b\}$. But they are not enough because $p(a, a)$ is not an answer for $p(U,V)\{U\neq V\}$ due to the constraint $\{U\neq V\}$. $P(a, a)$ is certainly a possible answer for $p(a,Y)\&\{Y\neq b\}$. Therefore, in addition to the answer derivation steps using answers from $p(U,V)\{U\neq V\}$, a new goal $p(a,a)q(a, Z)\&\{Z\neq a\}$ is needed to make up for the missing answer. This is the purpose of a complement derivation step which produces a new goal from an old goal and an answer providing subgoal.

Let $p\&C_p$ be an answer providing subgoal, $G_1 = A_1, \dots, A_n$ be a goal consisting of n subgoals, and $A_i = q\&C$ be the subgoal selected by the computation rule, where q is an atom and C is a conjunction of constraints. Also let μ be the substitution formed from C_p by changing " $\neq$ " into "/". To make the following definition simple, let us assume C_p is one constraint, not a set of constraints. We say there is a *complement derivation step* from G_1 to G_2 if the following condition hold:

- (a) $p\mu$ and q are unifiable with a mgu σ ;
- (b) $G_2 = A_1\sigma, \dots, A_{i-1}\sigma, A_i\sigma, A_{i+1}\sigma, \dots, A_n\sigma$
- (c) no $A_j\sigma$ is false.

We may also say that G_2 is the result of a complement derivation step from G_1 .

Example: Suppose $p(a,Y)\&\{Y\neq b\}, q(Y,Z)\&\{Y\neq Z\}$ is a goal with $p(a,Y)\&\{Y\neq b\}$ being selected and $p(U,V)\{U\neq V\}$ is an answer providing subgoal. $\mu=\{U/V\}$ is the substitution formed from the constraint $\{U\neq V\}$. $p(U,V)\mu = p(V,V)$ and $p(a,Y)$ are unifiable with mgu $\sigma=\{V/a, Y/a\}$. Therefore the new goal as the result of this complement derivation step from G_1 is $(p(a,Y)\&\{Y\neq b\})\sigma, (q(Y,Z)\&\{Y\neq Z\})\sigma = p(a,a), q(a,Z)\&\{Z\neq a\}$.

Notice that in the above definition for the complement derivation step, we assumed that the constraints in the answer providing subgoal is only one constraint, not a set of constraints. In general case, for each constraint in the set of constraints there is a possible complement derivation step from the given goal to a new goal. Also notice that converting a normalized constraint into a substitution is always possible by the definition of the normal form for a constraint. This is one of the reasons why a constraint needs to be kept in normal form.

The rationale of the complement derivation step is as follows: μ being a substitution formed from the constraint C_p in $p\&C_p$ means $p\mu$ represents all the answers of p which are not answers for $p\&C_p$. Then $p\mu\sigma=q\sigma$, where σ is a mgu for $p\mu$ and q , represents all the answers of $p\mu$ that are also answers of q , which are exactly what q needs from the answers of p but does not get from the answers of $p\&C_p$.

In processing a subgoal, it is very important (for completeness) that all possible values for its variables are considered. Figure 4.4 shows schematically how our methods do it. For example, suppose the selected subgoal is $q(X,Y,Z)$ and

the only answer providing subgoal APS is $q(a,Y,Z) \& \{Y \neq b\}$. Then in the rule derivation version of the goal the subgoal RDS will be $q(X,Y,Z) \& \{X \neq a\}$. This leaves $q(a,b,Z)$ for the complement derivation version of the subgoal CDS. This explicitly shows how the three different versions cover all possible values for the variables. The general case is more complex because there may be multiple answer providing subgoals. But the basic principle that CDS and RDS covers the remaining cases still holds true.

Figure 4.4: A Venn diagram for the three versions of a subgoal. APS is an answer providing subgoal, RDS is the rule derivation version of the subgoal and CDS is the complement derivation version of the subgoal.

4.3.5.4 The Search Tree

The search tree for GSI, which is called GSI-tree, is an abstract description of the search space and is not a practical implementation structure. However, it is very useful in explaining how the search proceeds in GSI.

The nodes in a GSI-tree correspond to goals. The root node corresponds to the original goal. During the search procedure, any *open node** can be chosen by the search rule to expand next. The goal of the chosen node is referred to as the *current goal*. The way to expand a node is by taking a derivation step. The new goal resulting from a derivation step will be a child node of the current node. The selection of a subgoal for the derivation step is made by the computation rule. The selected subgoal will be called the *current subgoal*. The newly produced child node is, of course, open. The current node will remain open until no more derivation step can be taken.

A *solution table* is used to register subgoals and their answers. It will be described in detail in the next subsection. Subgoals are registered when they are first selected by the computation rule. The first time an open node is chosen to be expanded, the solution table is checked to find all the registered subgoals which are unifiable with the selected subgoal of the current goal. These subgoals are candidates for *answer providing subgoals* of the current subgoal. Subgoals generated after current subgoal will not be considered for answer providing to the current subgoal. Subgoals may have constraints, but two subgoals being unifiable means the atoms of the two subgoals are unifiable. e.g., $p(X,Y) \& \{X \neq Y\}$ and $p(a,Z) \& \{Z \neq b\}$ are unifiable since $p(X,Y)$ and $p(a,Z)$ are. Suppose $A_1 = p \& C_1$ is the current subgoal and $A_2 = q \& C_2$ is a candidate for an answer providing subgoal. Let σ be a mgu for p and q . Then A_2 is an answer providing subgoal for A_1 if

* *Open nodes* are those nodes in the tree that either have not been selected for expansion or have been selected but still have some possible derivation steps which can be applied to them.

and only if neither $A_1\sigma$ nor $A_2\sigma$ is false.

With all the answer providing subgoals given, it is the time to discuss which derivation step is applicable. Note that whether an applicable derivation step can actually take place is subject to the satisfaction of the conditions of that derivation step. Let $G_1 = A_1, \dots, A_i, \dots, A_n$ be the current goal with $A_i = q \& C$ being selected and let $p \& C_p$ be an answer providing subgoal. For each answer of $p \& C_p$, there may be an answer derivation step from this answer and the current goal G_1 to a new goal. For each constraint C_p^i in C_p (C_p is a set of constraints), the complement derivation step from $p \& C_p^i$ and G_1 to a new goal is also possible. But note that for a new goal from a complement derivation step, the corresponding subgoal should remain selected. It is also required that such a new goal should take no subgoal as its answer providing subgoal except subgoals from its sibling node or its parent's answer providing subgoals. The reason for this is to avoid possible answer providing cycles since the corresponding subgoal in a new goal from a complement derivation step serves like an answer providing subgoal for its parent. There are two reasons for this requirement: First, if a node other than a sibling node of the new goal is generated after the parent node of the new goal, this node should take that parent node as its "answer providing goal" instead of being "an answer providing goal" for the new goal. Second, for a node generated before the parent node of the new goal, if it does not provide answers for the parent node it can not be an "answer providing goal" for the new goal either. The real intention of the second part of this requirement is to prevent the new goal from taking its parent node as its "answer providing goal". It also

prevents the new goal from taking its grand parent node as its "answer providing goal" if its parent is also generated by a complement derivation step from its grand parent. The above two requirement can be accomplished by choosing to expand the new goals from the complement derivation steps right after they are generated, selecting the corresponding subgoal and looking for answer providing subgoals among the candidates specified above.

To apply a rule derivation step, new constraints need to be generated first if there exist some answer providing subgoals. It is done by calling the algorithm for constraint generation on q and p and normalizing the resulting constraint, i.e., $\text{Norm}(\text{ConstGen}(q, p), L)$, where p and q are as assumed above and L is the list of variables appearing in $\text{ConstGen}(q, p)$ but not in q . It will generate one constraint. This process is done to all the answer providing subgoals. Let C_g be the conjunction of all the constraints thus generated. Then $G_1' = A_1 \& C_g, \dots, A_i \& C_g, \dots, A_n \& C_g$ is *rule derivation version* of the goal to which a rule derivation step can be applied. Therefore, this version of the goal should also be associated with the current node. The difference between G_1 and G_1' is that G_1' has more constraints. For each rule, there is a potential rule derivation step from the rule and the goal G_1' . The resulting goal is a child node to the current node.

As a summary of the above description, an answer derivation step is used to make use of a saved answer from an answer providing subgoal. A rule derivation step is used to do reasoning with a rule and, at the same time, avoiding any answers which might be produced by answer providing subgoals. A complement

derivation step is used to supply any answers that may be missed by an answer providing subgoal. Answer derivation steps are subjected to the availability of answers. More answers may be put in the solution table later on for an answer providing subgoals and these answers can then be used in additional answer derivation steps. However, it is important to note that when an open subgoal S is expanded for the first time, only previously expanded subgoals are eligible to become answer providing subgoals for S . Subgoals that have not yet been expanded may have answers in common with S , but they will get answers from S instead of supplying answers to S .

Example: Consider the GSI-tree in Figure 4.5. Let $p(X,Y),q(Y)$ be the current goal with subgoal $p(X,Y)$ selected. Let $p(a,Y)\&\{Y\neq a\}$ be the only answer providing subgoal and $p(a,b)$ the only answer of it and let $p(X,Y) :- r(X)$ be a rule. There is one answer derivation step from the current goal and the answer $p(a,b)$ to a new goal $q(b)$. A complement derivation step is also possible since the answer providing subgoal has a constraint. The constraint $\{Y\neq a\}$ is converted into a substitution $\{Y/a\}$. $p(a,Y)\{Y/a\}$ gives us $p(a,a)$. The mgu for $p(a,a)$ and $p(X,Y)$ is $\sigma=\{X/a, Y/a\}$. Therefore the complement derivation step produces the new goal $p(a,a),q(a)$. For a rule derivation step, the new constraint generated is $\text{Norm}(\text{ConstGen}(p(X,Y), p(a,Y)), L) = \{X\neq a\}$. The rule derivation version of the current goal then is $p(X,Y)\&\{X\neq a\},q(Y)$. A double colon is used to divide the goal and its rule derivation version in Figure 4.5. A rule derivation step from this goal and the given rule produces the new goal $r(X)\&\{X\neq a\},q(Y)$.

current goal:	$p(X,Y),q(Y)$
rule:	$p(X,Y) :- r(X)$
answer providing subgoal:	$p(a,Y) \& \{Y \neq a\}$
answer:	$p(a,b)$

Figure 4.5: A GSI-tree showed different derivation steps

A *GSI-derivation* is the sequence of goals in a path of a GSI-tree initiating from the root. The rule derivation version of a goal will be put right behind the "real" goal in the sequence. A finite GSI-derivation ending with an empty goal is a *GSI-refutation*. A GSI-refutation is a successful deduction which produces an answer in the form of substitution. The composition of the mug's of the derivation steps in a GSI-refutation is a substitution. A *computed answer substitution* of a GSI-refutation is obtained by restricting such a substitution to the variables in the original goal (goal at the root).

This deduction method requires the following restriction on the computation rule: new subgoals introduced in a derivation step must be selected before the other subgoals in a goal. For example, if the goal p,q is expanded by a derivation step and produces the resolvent $p_1.p_2.q'$, where $p_1.p_2$ is derived from p , then p_1 and p_2 must be selected before q' in the later derivation steps since they are introduced after q . Intuitively, this restriction requires that once a subgoal in a

goal is selected, the deduction procedure will concentrate on it. The other subgoals in the goal will not be involved in derivation steps until this subgoal is solved, i.e., when this subgoal and all the descendent subgoals derived from it disappear. The purpose of this restriction is to ensure completeness, i.e., all the answers for any subgoal will be produced sooner or later. It is explained in more detail at the end of this sub-subsection.

This restriction on the computation rule makes it possible to define a *sub-refutation*, a concept needed to explain how the answers of a subgoal are collected. Let the *length* of a goal be the number of subgoals in it. A *sub-refutation* is a part of a GSI-derivation which satisfies the following condition: If n is the length of the first goal of this part of derivation, then the last goal of this part of derivation is the first goal in the derivation whose length is less than n . Actually, a sub-refutation is equivalent to a GSI-refutation for the subgoal selected in the first goal of the sub-refutation if we ignore the other subgoals in the first goal. Consider an example GSI-derivation $s \rightarrow p, q \rightarrow p_1, p_2, q' \rightarrow p_1', q'' \rightarrow q''' \rightarrow \square$, with selected subgoals bold faced. $p_1, p_2, q' \rightarrow p_1', q''$ is a sub-refutation equivalent to a GSI-refutation for p_2 . $p, q \rightarrow p_1, p_2, q' \rightarrow p_1', q'' \rightarrow q'''$ is also a sub-refutation which is equivalent to a GSI-refutation for p .

We now describe how the answers of subgoals are generated. Each sub-refutation produces an answer for a selected subgoal. The definition of sub-refutation suggests the way to identify these sub-refutations. They all end with the new child node and begin with a node in the path from the root to this new

child node. In the above example, two sub-refutation are completed after the derivation step $p_1', q'' \rightarrow q'''$. They are $p_1', q'' \rightarrow q'''$ and $p, q \rightarrow p_1, p_2, q' \rightarrow p_1', q'' \rightarrow q'''$. The former produces an answer for subgoal p_1' while the latter for subgoal p .

The restriction ensures that no answer is missed for any selected subgoal. In the other words, some answers may be missed if the restriction is violated. Consider a derivation $p, q \rightarrow p_1, p_2, q \rightarrow p_1', p_2' \rightarrow \dots$, with selected subgoals bold faced. The restriction is violated with the selection q in the goal p_1, p_2, q . The answers for p in the first goal of the derivation should be produced by p_1, p_2 in the second goal. But the derivation step applied on the subgoal q may introduce some substitution which instantiates p_1, p_2 into p_1', p_2' . In this case p_1', p_2' are not as general as p_1, p_2 , and some answer for the latter may not be generated in solving the former.

4.3.6 Data Structure

The *solution table* described below is a data structure used to store the answers and other necessary information. The purpose of introducing the solution table is to help understanding what information is needed and how it is related to the resolution procedure. The solution table is one way to implement the above algorithm, but there are also other implementations.

Each entry of the solution table contains information for one subgoal. The table has three columns. The first column contains a subgoal and is used as the

key of the entry. The second column is a pointer to a list stored answers for this subgoal. The third column contains a number of pointers, each pointing to a list of answers of an answer providing subgoal that provides answers for this subgoal. There is one pointer for each such answer providing subgoal. Such a pointer moves from one answer to the next one in the answer list when the next answer derivation step take place for the subgoal which is the key for this table entry.

Figure 4.6: A solution table

Example: In the Figure 4.6, $p(f(W), Z)$ and $p(X, b) \& \{X \neq a\}$ are registered. The former is an answer providing subgoal for the latter. Consequently there is a pointer in the third column of the entry for $p(X, b) \& \{X \neq f(a)\}$ pointing to the answer list for $p(f(W), Z)$. An answer derivation step moves this pointer to get to

the next answer. The first answer of $p(f(W), Z)$ is $p(f(a), Z)$, which unifies with $p(X, b)$ but makes the constraint $\{X \neq f(a)\}$ false and therefore is not an answer for $p(X, b) \& \{X \neq f(a)\}$. The second answer, $p(f(b), b)$, is an answer for $p(X, b) \& \{X \neq f(a)\}$ and is stored in the answer list for $p(X, b) \& \{X \neq f(a)\}$. Notice that answers are stored in the form of substitutions. Applying one such substitution to the registered subgoal will give an answer in the form an answer derivation step wants; e.g., the first answer for $p(f(W), Z)$ in the Figure 4.6 is $p(f(W), Z)\{W/a\} = p(f(a), Z)$.

5 Empirical Evaluation

Some sample Prolog programs are used in this chapter to study the empirical characteristic of our new inference engine, GSI. We start with a program that computes a transitive closure. This case is described in detail to illustrate how GSI works as well as its empirical behavior on this example. This also shows the way in which GSI differs from Prolog. We then modify this program slightly to make its evaluation more difficult for GSI. This illustrates additional features of it and gives some insight into the type of inferences which adversely effect its efficiency. Finally we look at the empirical properties of the rule set of MOSES2.

5.1 Transitive Closure Examples

Figure 5.1 is a typical program computing transitive closure. *ac* stands for the ancestor relation; *pa* stands for the parent relation. Figure 5.2 is a graph representation of the parent relation defined by the *pa* facts in the program in Figure 5.1. A GSI-tree for the evaluation of the goal *ac(A, D)* is given in Figure 5.3. The computation rule assumed for this tree is left-to-right.

$ac(A, D) :- pa(A, D).$	(r1)
$ac(A, D) :- pa(A, B), ac(B, D).$	(r2)
$pa(a, b).$	(r3)
$pa(b, c).$	(r4)
$pa(b, d).$	(r5)
$pa(c, b).$	(r6)
$pa(c, d).$	(r7)

Figure 5.1: A program for transitive closure

Figure 5.2: A graphic representation

Figure 5.3: A GSI-tree for the program in Fig. 5.1

The table in Figure 5.4 shows step by step the evaluation sequence, or the way the GSI-tree is built. Each entry corresponds to one or more derivation steps. To make the table short, some entry may contain several derivation steps. The column "derivation type" tells what derivation step is used for expanding the node. The column "rule/answer used" gives which rule or answer is used for the derivation step. When a derivation step is a rule derivation step, a rule number is given. If it is an answer derivation step, an answer providing subgoal is given. This actually means the next applicable answer in the answer list of the subgoal is used for the derivation step. The "answer list change" column shows the answers newly added to answer lists. Consider the second entry of the table

which pertains to expanding node N1. The goal at node N1 is $pa(A, D)$ which only contains one subgoal; it is therefore selected. The derivation type is rule derivation. The rules $r1, r2, \dots, r5$ are used and child nodes $N11, N12, \dots, N15$ are produced. The last column shows that five answers, namely $ac(a, b), ac(b, c), \dots, ac(c, d)$ are added to the answer list for subgoal $ac(A, D)$. The same five answers are also answers for subgoal $pa(A, D)$ and are added to the corresponding answer list. The notation $\langle a, b \rangle$ indicates that a is substituted for A and b for D .

node expanded	subgoal selected	child node	derivation type	rule/ answer used	answer list change
N	ac(A,D)	N1	rule	r1	
N1	pa(A,D)	N11, N12, N13, N14, N15	rule	r3, r4, r5, r6, r7	<div> $ac(A,D)$ $pa(A,D)$ </div> <div> $\langle a,b \rangle$ $\langle a,b \rangle$ </div> <div> $\langle b,c \rangle$ $\langle b,c \rangle$ </div> <div> $\langle b,d \rangle$ $\langle b,d \rangle$ </div> <div> $\langle c,b \rangle$ $\langle c,b \rangle$ </div> <div> $\langle c,d \rangle$ $\langle c,d \rangle$ </div>
N	ac(A,D)	N2	rule	r2	
N2	pa(A,B)	N21, N22, N23, N24, N25	answer	pa(A,D)	
N21	ac(b,D)	N211, N212	answer	ac(A,D)	<div> $ac(A,D)$ $ac(b,D)$ </div> <div> $\langle a,c \rangle$ $\langle b,c \rangle$ </div> <div> $\langle a,d \rangle$ $\langle b,d \rangle$ </div>
N22	ac(c,D)	N221, N222	answer	ac(A,D)	<div> $ac(A,D)$ $ac(c,D)$ </div> <div> $\langle b,b \rangle$ $\langle c,b \rangle$ </div> <div> $\langle c,d \rangle$ </div>
N23	ac(d,D)		answer	ac(A,D)	
N24	ac(b,D)	N241, N242	answer	ac(b,D)	<div> $ac(A,D)$ </div> <div> $\langle c,c \rangle$ </div>
N25	ac(d,D)		answer	ac(d,D)	
N21	ac(b,D)	N213	answer	ac(A,D)	<div> $ac(b,D)$ </div> <div> $\langle b,b \rangle$ </div>
N22	ac(c,D)	N223	answer	ac(A,D)	<div> $ac(c,D)$ </div> <div> $\langle c,c \rangle$ </div>
N23	ac(d,D)		answer	ac(A,D)	
N24	ac(b,D)	N243	answer	ac(b,D)	
N25	ac(d,D)		answer	ac(d,D)	

Figure 5.4: Evaluation sequence

<u>$pa(A, D)$</u>	<u>$ac(A, D)$</u>	<u>$ac(b, D)$</u>	<u>$ac(c, D)$</u>	<u>$ac(d, D)$</u>
$\langle a, b \rangle$	$\langle a, b \rangle$	$\langle b, c \rangle$	$\langle c, b \rangle$	$\emptyset$
$\langle b, c \rangle$	$\langle b, c \rangle$	$\langle b, d \rangle$	$\langle c, d \rangle$	
$\langle b, d \rangle$	$\langle b, d \rangle$	$\langle b, b \rangle$	$\langle c, c \rangle$	
$\langle c, b \rangle$	$\langle c, b \rangle$			
$\langle c, d \rangle$	$\langle c, d \rangle$			
	$\langle a, c \rangle$			
	$\langle a, d \rangle$			
	$\langle b, b \rangle$			
	$\langle c, c \rangle$			

Figure 5.5: Final content of the answer lists

The nodes N and N1 are expanded using rule derivation steps while the nodes N2, N21, ..., N25 are expanded using answer derivation steps. The goal $ac(b, D)$ at the node N21, for instance, takes $ac(A, D)$ as its answer providing subgoal since $ac(A, D)$ is the selected subgoal at the node N and the node N was expanded before N21. In this case $ac(A, D)$ is more general than $ac(b, D)$. The algorithm for constraint generation, taking these two subgoals as input, produces a "false" as output. A "false" constraint makes the rule derivation version of the goal false. The meaning of this is that no rule derivation step is needed, which is correct because all the answers of $ac(b, D)$ are also answers of $ac(A, D)$. A double colon is used to divide a goal and its rule derivation version in the figure. When the node N21 is expanded for the first time, there are two answers in the answer list for $ac(A, D)$ which are unifiable with $ac(b, D)$. They are $ac(b, c)$ and $ac(b, d)$. Two terminal nodes, N211 and N212, are created accordingly. But the expansion of node N22 adds the answer $ac(b, b)$ to the answer list for $ac(A, D)$, which makes one more child node for the node N21 when it is revisited later. The expansion of the node N22 and N24 is similar to that of N21 except that the subgoal $ac(b, D)$ at the node N24 takes the subgoal $ac(b, D)$ at the node N21 as

its answer providing subgoal.

The rule derivation version of the goal at N1 is the same as the goal at N1, $pa(A, D)$. This is also the case for the goal at N. The remainder of the goals in the GSI-tree have false as their rule derivation version which prevents rules from being applied to these nodes. The reason for this is that all of the answers at these nodes will be provided by some answer providing subgoals. To make the tree clear, only few nodes are marked with rule derivation versions. A double colon is used to divide a goal and its rule derivation version in the tree. Nodes N23 and N25 are failure nodes. GSI saves failure subgoals just like other subgoals. The only difference is that the answer list of a failure subgoal will be marked as failure.

The final set of answers for *ac* subgoals are listed in Figure 5.5. Note that the process terminates even though there is a cycle in Figure 5.2. The reason is that when a duplicate answer is produced, it will not be added to the answer list because only non-duplicate answers are added. In the last round GSI tries to expand the open nodes, but it finds that nothing has been added to any answer list. The process terminates because no more derivation steps are possible.

If we change the second rule in the Figure 5.1 into $ac(A, D) :- ac(B, D), pa(A, B)$. and still follow the left-to-right computation rule, we have a left recursive rule. Figure 5.6 shows a GSI-tree for this modified rule set with the same original goal. The main difference between this and the previous example is that $ac(B, D)$ is the subgoal selected at node N2. Since this is a subset of $ac(A, D)$

(node N), the rule derivation version of the goal at N2 is false which prevents recursive application of rule r2. Thus, the subtree of node N2 in Figure 5.6 only has to do with how answers of the answer providing subgoals are used in answer derivation steps. The next example given presents a closer look on how a goal is evaluated with a left recursive rule set.

Figure 5.6: A GSI-tree for left-recursive version of the program in Fig 5.1

The program in Figure 5.7 is not only left recursive, but the evaluation procedure also starts with a more specific goal, $ac(a, D)$. During the evaluation, a more general subgoal, $ac(B, D)$, is produced and selected. This more general

subgoal takes the more specific subgoal $ac(a, D)$ as its answer providing subgoal, which give us a chance to see the generation of the constraints and how they work. Figure 5.8 is a corresponding GSI-tree and Figure 5.9 gives the goals associated with each node in the tree. Figure 5.9 also lists the rule derivation versions of the goals unless it is false. Only node N2 has a rule derivation version which is different than the goal at the node and not false. Finally Figure 5.10 shows the content of the answer lists.

$ac(A, D) :- pa(A, D).$	(r1)
$ac(A, D) :- ac(B, D), pa(A, B).$	(r2)
$pa(a, b).$	(r3)
$pa(b, c).$	(r4)
$pa(c, a).$	(r5)

Figure 5.7: A left recursive program for computing transitive closure

Figure 5.8: A GSI-tree for the program in Fig. 5.7

N	$ac(a, D) :: ac(a, D)$
N1	$pa(a, D) :: pa(a, D)$
N2	$ac(B, D), pa(a, B) :: ac(B, D) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N21	$pa(a, a)$
N22	$pa(a, a)$
N23	$pa(a, a)$
N24	$pa(B, D) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$ $:: pa(B, D) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N241	$pa(a, b)$
N242	$pa(a, c)$
N25	$ac(C, D), pa(B, C) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N251	$pa(B, a) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N2511	$pa(a, c)$
N252	$pa(B, b) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N253	$pa(B, c) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N2531	$pa(a, b)$
N254	$pa(B, a) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N2541	$pa(a, c)$
N255	$pa(B, c) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N2551	$pa(a, b)$
N256	$pa(B, b) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N257	$pa(B, a) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N2571	$pa(a, c)$
N258	$pa(B, c) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$
N2581	$pa(a, b)$
N259	$pa(B, b) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$

Figure 5.9: Goals associated to the nodes in the tree in the Fig. 5.8

<u>$ac(a, D)$</u>	<u>$ac(B, D)$</u>	<u>$pa(a, D)$</u>	<u>$pa(B, D) \& \{B \neq a\}$</u>
<a, b>	<a, b>	<a, b>	<b, c>
<a, c>	<b, c>		<c, a>
<a, a>	<c, a>		
	<a, c>		
	<c, b>		
	<b, a>		
	<a, a>		
	<c, c>		
	<b, b>		

Figure 5.10: Final content of the answer lists

Consider, for example, node N2 in the GSI-tree. The goal corresponding to this node is given in Figure 5.9, $ac(B, D), pa(a, B)$. The subgoal $ac(B, D)$ is selected, and the subgoal $ac(a, D)$ selected at the node N is the only answer providing subgoal for $ac(B, D)$. The answer derivation steps using the answers from this answer providing subgoal introduced the child nodes N21, N22 and N23. The constraint, $\{B \neq a\}$, is produced by the algorithm for constraint generation using the two subgoals as input. Therefore the rule derivation version for the node N2 is $ac(B, D) \& \{B \neq a\}, pa(a, B) \& \{B \neq a\}$. The derivation steps applied to node N25 are all answer derivation steps. $ac(C, D)$ is the selected subgoal there. Its answer providing subgoal is $ac(B, D)$ which is the selected subgoal at the node N2. Since this answer providing subgoal can provide all the answers needed, no rule derivation is applied to node N25. This fact is found out by the algorithm for constraint generation which output a "false" upon the input of the two subgoals. The left-most child of the node N25, for example, is node N251 which is produced by an answer derivation step using the first answer in the answer list for subgoal $ac(B, D)$. The other answer derivation steps applied to node N25 use the other answers from the answer list. The answer from the answer list of $pa(B, D) \& \{B \neq a\}$ is used for the answer derivation step at node N251, which gives node N2511. Answer derivation steps are also used at the nodes N253, N254, etc.

These examples all assume certain search rule in which the program rules are chosen in the order in which they are listed. Listing (choosing) the rules or choosing open nodes in a different order may change the GSI-tree because the

answers to subgoals may be generated in a different order which may affect both the answer derivation steps of the nodes and the rule derivation steps. The latter may be affected because the rule derivation version of a goal depends upon the selected subgoals of previously generated goals as its answer providing subgoals. This situation is quite different than SLD-resolution in which the SLD-tree does not depend upon the search rule. However, the GSI-trees for any search rule for the above examples are all finite and similar in size to the GSI-tree given above because the variety of subgoals and answers essentially remain unchanged.

5.2 The Rules of MOSES2

This section discusses how GSI interprets the rules of MOSES2 which are given in Appendix A. Figure 5.11 is a table that shows how each subgoal may be evaluated. It has one entry for each predicate instead of each subgoal. Several different subgoals with the same predicate may be generated by the interpreter. If this is the case, some of these subgoals may have common answers. The fourth column of the table pertains to this. The value "no" under this column means all the different subgoals are not unifiable and therefore they have no common answers. The value "s" means all the subgoals are the same (alphabetically identical). In this case, one of the subgoals is the answer providing subgoal for all the other subgoals and no rule derivation steps are needed for those "answer receiving" subgoals.

For example, the subgoals with the predicate *retrieve* are called by the *main_goal* and *next_search*. The value "m" in the third column means it is called

many times. If it were called only once, a value "1" would have showed in this column. The value "no" in the fourth column says no two subgoals have common answers. The reason for that is that each call for *retrieve* has different instantiations of its only parameter "Motor_type". Another example entry is the one for *field_cost*. The value "s/no" means that any two subgoals are either have the same answer set or have no answer in common at all. For any calls to the *field_cost* by the *motors1*, the first parameter is always instantiated into a constant and the second one is always a variable. Therefore two subgoals have the same answer set if the first parameter is instantiated into a same constant; and the two subgoals have no answer in common otherwise.

predicate	called by	times called	type of inters- ection	note
main_goal	(root)	1		
init	main_goal	1		
motor_type	main_goal	1		
endow	main_goal	1		
retrieve	main_goal, next_search	m	no	distinct Motor-type
report_query	retrieve	m	no	...
user_choose	retrieve	m	no	...
report_and_ask	user_choose	m	no	...
maybe_more	user_choose	m	no	...
next_search	user_choose, maybe_more	m	no	...
substitute	next_search, motors1	m	s	
motors	report_and_ask	m	no	distinct Motor_type
report_motor	report_and_ask	m	no	...
motors1	motors	m	no	...
get_reply	report_and_ask	m	s	big trouble
hp_tension	motors, hp_min	m	s	
hp_decel	motors, hp_min	m	s	
rpm_max	motors, fw_rpm_low	m	s	
low_inert_cost	motors	m	s/no	Tsel, Tdec always constant; F1
winding_cost	motors	m	s/no	always variable Winding always constant; F2
motor_type_cost	motors	m	s/no	always variable Motor_type always constant; F3
field_cost	motors	m	s/no	always variable Mult_field always constant; F4
frame_change	motors1	m	s	always variable
hp_low	motors1, motor_type	m	s	

hp_high	motors1	m	s	
s_base_rpm_low	motors1	1		
s_base_rpm_high	motors1	1		
s_enclosure	motors1	1		
fw_rpm_low	motors1	m	s	
fw_rpm_high	motors1	m	s	
frame_series_low	motors1	1		
frame_series_high	motors1	1		
frame_type	motors1	1		
rpm_min	motors1,	m	s	
	hp_decel			
remark	motors1	m	s	
main	motors1	m	s/no	certain parameters always constant
field	motors1	m	s/no	...
frames	motors1	m	s/no	...
impose_remarks	motors1	m	s/no	...
base_rpm_low	motors1,sub1, sub2,sub3	m	s	
base_rpm_high	motors1,sub1, sub2,sub3	m	s	
enclosure	motors1,sub1, sub2,sub3	m	s	
remarks	impose_remarks	m	s/no	
if_low_inertia	low_inert_cost	m	s	
app	if_low_inertia	m	s	
low_inertia	if_low_inertia	m	s/no	the only parameter always constant
sub1	substitute, s_enclosure, frame_change, s_base_rpm_low, s_base_rpm_high	m	s	
sub2	substitute, s_enclosure, frame_change, sub3, s_base_rpm_low, s_base_rpm_high	m	s	
sub3	substitute, s_enclosure, frame_change, frame_type, s_base_rpm_low, s_base_rpm_high,	m	s	

	frame_series_low, frame_series_high		
line_speed	rpm_min, hp_tension, rpm_max, hp_decel	m	s
diameter_max	rpm_min, hp_decel	m	s
diameter_min	rpm_max	1	
tension	hp_tension	1	
paper_inert	hp_decel	1	
steel_inert	hp_decel	1	
stop_time	hp_decel	1	
max	hp_min	1	
hp_min	hp_low, hp_high	m	s

Figure 5.11: Evaluation pattern of the subgoals in MOSES2

In the execution of the rules of MOSES2, there is no case where two subgoal have some answers in common and also have some answers not in common. Two subgoals are either identical or have nothing to do with each other at all. This greatly simplified the execution under GSI because no constraints are generated. That is, the rule derivation version of a goal is always the goal itself of false. The reader can see from this example that the problem of asking the user the same question more than once is solved naturally. For example, the value of line speed is needed many times during the execution, but user is only asked once. The value is stored in the answer list for line_speed(Ls), and all calls for line speed except the first one will get the value from the answer list instead of from the user. This shows up in Figure 5.11 as an *m* in column 3 which means that line_speed is called many times and a *s* in column 4 because all calls are the same.

The only thing that gives GSI any trouble is the subgoal *get_reply(More)*. It is invoked at the several different stage of the execution of the MOSES2. The invoked subgoals are all the same. But according to the context, the user may give different answers; e.g., after showing all suitable RPMIII motors, the user is asked if he want to see more (other) motors. He may answer "yes" at this point. Then MOSES2 will search for the suitable "SUPER-RPM" motors and show them to the user. After showing the "SUPER-RPM" motors the user is asked the same question again and he might answer "no" this time. The difficulty is that GSI saves the answer *get_reply(yes)* after first time the question is asked. The next call of this subgoal will be directed to this answer. Thus, the user would not be asked a second time and he has lost a chance to say "no" after seeing all the selected "SUPER-RPM" motors. Questions like "what is the line speed" do not have such a problem because the answer is always the same. In the other words, the question is not context sensitive. There is a simple way to solve the problem, which is adding one more parameter to the predicate. The subgoal being called now is *get_reply(Motor_type, More)*. It make the subgoals different for different stage of execution. Therefore, the answers saved in the first stage is no good for the other stages.

5.3 Discussion

The examples given in this chapter are very different than the examples given in chapter 4. The reason is that algorithms in chapter 4 are designed to work in the general case and the substitutions and constraints which illustrate the

intricacies of algorithms are very different than those in this chapter. The most prevalent constraint in this chapter is *false* and all of the others are of the form *variable $\neq$ constant*. We believe that this will also be the case for many other examples too because the characteristics of the above examples appear to be very common.

Most expert systems will probably not need much of generality provided by GSI. In particular, the rules of an expert system usually are not nearly as complex as those found in a typical Prolog program. For example, they probably will have limited use of recursion and probably will not manipulate lists.

In larger transitive closure problems the GSI-tree would be much larger but would contain the same kinds of goals. The lowest level answers would come from the facts that correspond to individual arcs in the graph. The constraints would be sets of inequations of the form *variable $\neq$ constant* as in the above example. The reason is that at each stage the interpreter processes all of the arcs leaving a particular vertex and the constraints essentially say this and the previous vertices should not be processed again.

None of the examples in this chapter contain a complement derivation step. These appear to be very rare in practice. Thus, it should be possible to develop a more efficient inference engine which is optimized for the kind of goals that occur in practice. We have not considered this because we wanted to develop an inference method which is logically complete. This is proven in the next chapter.

6. Meta-Theory of Graph Search

This chapter serves as the theoretic foundation for our new inference engine, GSI. Several propositions and theorems are presented and formally proven. We first investigate certain properties of the methods described in section 4.3. Since GSI uses these methods for processing constraints, it depends upon their correctly performing the functions for which they are designed. For example, if the normalization method would not terminate for certain constraint, then GSI would not perform properly. Basically, we intend GSI to be a method for recursively enumerating refutations and this is only the case if normalization terminates for any possible inputs. The first section of this chapter establishes such properties of the basic method.

GSI also requires the methods to be correct. For example, one correctness condition for normalization is that it preserves logical equivalence, i.e., the output constraint must be logically equivalent to the input constraint. In the first section we prove this and we also show that the normalization method is a decision procedure for the validity and the unsatisfiability of a constraint.

The second section of this chapter is primarily concerned with the logical completeness of GSI. Of course, completeness requires that basic methods are correct. But, the main issue of the section is not the correctness of the basic methods. Rather it investigates whether GSI might miss some refutations even though all of the basic methods are correct. We prove that the set of answers computed by GSI is in some sense equivalent to the set of answers produced by

SLD-resolution. It is in this sense that GSI is complete.

6.1 Properties of the Basic Methods

In the following we explore certain properties, such as termination and correctness, of the algorithm for normalization since these properties are not obvious.

The first proposition concerns the kind of constraints that result from applying a substitution to a constraint. This is important because the normalization method is only designed for such constraints.

Proposition 6.1 The algorithm for application of substitution to a constraint (see 4.3.2) produces constraints in which each inequation has a variable on either the left or the right of the $\neq$.

Proof Consider all the possible cases in which neither side of an inequation is a variable. They are: constant versus constant, constant versus function, function versus function. An inequation of the first case will be removed or lead to a true exit during the execution of the algorithm. An inequation of the second case will be removed by the algorithm. An inequation of the last case will be processed further by that algorithm. So, none of the three forms of inequation would appear in an output constraint from the algorithm. $\square$

Proposition 6.2 The algorithm for normalization terminates.

Proof Consider step 2.3 of the algorithm. Whenever an inequation $X \neq t$ is

selected, there will be no other appearance of X in the constraint after the substitution is done. The algorithm will eventually reach a point where no inequation can be selected at step 2.3 since there is only finite number of variables. $\square$

Proposition 6.3 The result constraint of the algorithm for normalization is logically equivalent to the original constraint.

Proof Since a constraint is a logical expression, the logical equivalence of constraints is defined as the equivalence of logical expressions. i.e., two constraints are said logically equivalent if, for any ground assignment to the free variables in the constraints, the first constraint is true if and only if the second one is true.

The algorithm produces a constraint logically equivalent to the input constraint if each step of the algorithm keeps the logical equivalence of the constraint.

For the step 2.1 and 2.2, it is obvious that the logical equivalence is kept.

Whether the step 2.3 keeps the logical equivalence depends on what the substitution operation in this step does. Let $X \neq t$ be the one selected. Consider a ground assignment to all the variables in the old constraint which makes the old constraint true. If this assignment makes $X \neq t$ true, it also makes the new constraint true since $X \neq t$ is also in the new constraint. If this assignment makes $X \neq t$ false, then $X = t$ under the assignment. It means that the substitution of t for the other appearances of X does not change the truth value of those affected inequa-

tions under this assignment; i.e., for the inequation in the old constraint which is true under the assignment, the corresponding inequation in the new constraint is also true under the same assignment. Therefore under any circumstance this assignment makes the new constraint true. As for the assignment which makes the old constraint false, it must makes $X \neq t$ false. By the same argument as above the assignment makes the new constraint false too. Notice that we did not consider the universally quantified variables in the above proof. But since the equivalence is proved assuming all the variables are free, the equivalence stands if several universal quantifier are added before the constraints.

Step 3 can be considered a series of substitutions like those in the step 2.3. The proof then is the same as above.

The proof for step 4 is the same to the one for step 2.3.

To prove that the step 5 preserves the logical equivalence of the constraint it suffices to prove $(\forall W)(W \neq t \vee p) \equiv p$, where t is a term not containing W and p is any logical expression not containing W . Notice that $(\forall W)(W \neq t \vee p) \equiv (\forall W)(W \neq t) \vee p$. It follows that $(\forall W)(W \neq t \vee p)$ being false makes p false. Any assignment to the free variables that makes $(\forall W)(W \neq t \vee p)$ true makes either $(\forall W)(W \neq t)$ or p true. But it is impossible for the assignment to make $(\forall W)(W \neq t)$ true because when the value of W is t , $W \neq t$ is false. Therefore the assignment makes p true, and $(\forall W)(W \neq t \vee p) \equiv p$ is proved. $\square$

Incidentally, the method for applying a substitution to a constraint has same property. The algorithm does simplification after applying the substitution to the

constraint, and the simplification part of the algorithm preserves the logical equivalence of the constraint.

There is no question that a constraint produced by normalization is in normal form. The above proposition insures that this normalized form is logically equivalent to the original constraint. But it is not obvious that the algorithm will return with true (false) if the original constraint is logically true (false). This is a desirable feature of the algorithm because it insures that there is no delay in finding the truth value of a constraint. To prove that the algorithm has this feature, the following proposition needs to be proved first.

Proposition 6.4 A constraint in normal form is not a tautology nor is it unsatisfiable.

Proof This proof assumes that there are at least two elements in the Herbrand universe. Consider a ground assignment to all those free variables that appear in the right-hand side of some inequation. We will expand this partial assignment to a complete assignment to all the free variables which makes the constraint true. Take one inequation $X \neq t$ after the partial assignment. If t has no universally quantified variables, it is ground already; assign a different ground term to X . If t has some universally quantified variables, it must be a function since such a variable can not appear as an entire one side of any inequation in a normalized constraint. X is then assigned to a constant. The rest of the left-hand side variables can be assigned arbitrary values. The complete assignment thus obtained makes the constraint true since that particular inequation is true. We

can also expand this partial assignment to a complete one as follows which makes the constraint false. Arbitrarily take one constant and consider all the universally quantified variables with this constant as their value. For each inequation $X \neq t$, where t is ground now, X is assigned to the value of t . The complete assignment obtained in this way makes the constraint false. Since the constraint can be true or false under different assignment, the constraint is not a tautology nor is it unsatisfiable. $\square$

Corollary The algorithm for normalization will return with true (false) if the original constraint is logically true (false).

6.2 Soundness and Completeness of the New Inference Method

In this section we discuss the soundness and completeness of GSI described in chapter 4. As mentioned above, the completeness of GSI requires the basic methods to be correct. This section investigates whether GSI might be incomplete in some other way, and thus assumes that the basic methods are correct. It also assumes that the methods for generating subgoals are correct. For example, Figure 4.4 shows why all possible values for variables are considered by GSI. Obviously if APS, RDS and CDS in Figure 4.4 do not cover all possible values for the variables of the subgoal, GSI would not be complete. However, even if they do cover all the values, it still must be proven that GSI is complete and this proof is quite involved.

The soundness is not a problem for GSI because the rule derivation version

G' of a goal is just an ordinary goal G with (perhaps) some constraints. G' is a logical consequence of G since the constraints only restricts the values of variables in the G . For the case of a complement derivation step, the resolvent is an instance of its parent, which raises no soundness problem either. The same is true for an answer derivation step because it uses a proven answer. Therefore we have the following theorem for soundness.

Theorem 6.1 (soundness theorem) Let P and G be a program and a goal, α be the computed answer substitution of an GSI-refutation for P and G , then $G\alpha$ is a logical consequence of P .

The remainder of this section is devoted to establishing that GSI is logically complete. However, the concept of completeness is slightly different for GSI than for SLD-resolution because the purpose of GSI is to use a previously generated answer whenever possible instead of deriving it a second time. This has an effect on the kind of answers produced by GSI. For example, an SLD answer for $q(X, Y)$ may be $\{X/a\}$ which represents a refutation of $q(a, Y)$. This may corresponds to two GSI-refutation: one for $q(a, b)$ and one for $q(a, Y) \& \{Y \neq b\}$. Each of these refutations are less general than the one for SLD-resolution and GSI would find the more general one if the subgoal $q(X, Y)$ was selected before $q(a, b)$. However, if $q(a, b)$ was selected first then GSI is committed to use its answers and afterwards it will only consider the subgoal $q(a, Y) \& \{Y \neq b\}$ which is also more specialized than $q(X, Y)$.

Completeness requires that all possible answers of a subgoal are generated.

However, the above discussion shows that the GSI answers may be more specialized than SLD answers because of the order in which subgoals are selected. Below we prove that for any ground answer to a goal, there is a GSI-refutation which has this ground answer as an instance of the refutation's answer. Roughly speaking, we consider any given SLD-refutation, and for any ground instance of the answer corresponding to this refutation we find a GSI-refutation whose answer contains this instance. Since SLD-resolution is complete, this shows that all possible ground answers are an instance of some GSI-refutation.

During the proof of the completeness below, the following conventions are used in order to make the whole thing easier to describe and understand. Let P be the program, G be the original goal of concern. Suppose there is an SLD-refutation for P and G with computed answer substitution ω . Let $G_g = G\omega$ if $G\omega$ is ground, $G_g = G\omega\omega'$ otherwise, where ω' is a substitution which makes $G\omega$ ground. Then there is an SLD-refutation for P and G_g . Let $G_0 = G_g$, $G_1, \dots, G_n = \square$ be that refutation, $r_1, r_2, \dots, r_n$ and $\theta_1, \theta_2, \dots, \theta_n$ be the rules and mgu's used along the refutation.

The description of the algorithm also needs a concept called *ground companion* of a goal. G_i' is used to denote the ground companion of G_i . Using the notations given above, the ground companions of $G_0 = G_g$, $G_1, \dots, G_n$ can be defined recursively as follows:

- (1) $G_0' = G_0 = G_0\theta_1\theta_2 \dots \theta_n$;
- (2) If $G_i' = G_i\theta_{i+1} \dots \theta_n\tau$, where τ may be empty, then

$$G'_{i+1} = G_{i+1}\theta_{i+2} \dots \theta_n \tau \tau';$$

where τ' is a substitution obtained from a *ground substitution* for $G_{i+1}\theta_{i+2} \dots \theta_n \tau$ by restricting it to the variables appearing in $G_{i+1}\theta_{i+2} \dots \theta_n \tau$. A *ground substitution* for a goal G_a is an arbitrary substitution τ_a such that $G_a \tau_a$ is ground. Obviously τ' is empty if $G_{i+1}\theta_{i+2} \dots \theta_n \tau$ is already ground.

Intuitively, if we look at the SLD-refutation as it starts from G_i , the ground companion of G_i , which is denoted by G'_i , is one ground instance of the "final answer" of this refutation. Variables may appear in the "final answer" which means it is actually a set of "ground answers". We arbitrarily pick up one of the "ground answers", G'_i .

Given an SLD-refutation for P and G_g , the algorithm described below will find a GSI-refutation which corresponds to this given SLD-refutation in a GSI-tree for P and G . The algorithm finds for each step in the SLD-refutation the analogous step in the GSI-tree. It calls itself recursively upon an answer derivation step, which will find a sub-GSI-refutation corresponding to the answer providing subgoal. In order to make the following description and proof clear, we will simply say a call to the algorithm, whether it is the main call or a recursive one, finds a GSI-refutation without distinguishing a sub-GSI-refutation from a GSI-refutation.

The computation rule adopted by the GSI-tree may not be the same as the one adopted by the SLD-refutation. But the switching lemma [Lloy84] can be applied to the SLD-refutation to change the computation rule. Therefore, without

loss of generality, we can assume that the SLD-refutation always selects the subgoal we want to; i.e., it will follow the computation rule adopted by the GSI-tree. This assumption is made for the remainder of this section.

Correspondence algorithm $\text{Cor}(R, T)$.

Input: An SLD-refutation R for a program P and a goal G_g ; and a GSI-tree T for P and G with one node marked as the starting node.* The computation rule used to produce R is consistent with the computation rule and the search rule used to create T . By this we means that the subgoal selected at any point in R will always correspond to the analogous subgoal which is selected in the analogous goal in T .

Output: A finite GSI-refutation in T from the starting node such that G_g is an instance of $G\alpha$, where α is the computed answer substitution of the GSI-refutation.

1. Let the first goal in R be the current goal and the starting node be the current node.
2. Repeat the following.
 - 2.1 If the current goal is $\square$, exit. Otherwise, let q_i be the selected subgoal in the current node G_i and q be the selected subgoal in the current goal G_i . Also let G_i' be the ground companion of G_i and q' be the subgoal in G_i' corresponding to q .

* The root node is the starting node for the original call to the algorithm. But the recursive calls may have non-root node as the starting node.

2.2 q' , which is ground, is used to decide how to proceed in the following way.

- 2.2.1 If q' is an instance[†] of an answer providing subgoal for q_i , then call $\text{Cor}(R', T)$ with the node containing the answer providing subgoal as the starting node, where R' is a refutation obtained from the sub-refutation of R starting from G_i by ignoring the subgoals which are irrelevant to the refutation of q . Upon the return of this recursive call, let the new current goal be the first goal in R after the sub-refutation and the new current node be the node derived from the old current node by an answer derivation step such that q' is an instance of the answer used for this answer derivation step.
- 2.2.2 If q' is an instance of the selected subgoal of a child node derived from the current node by a complement derivation step, let the new current node be that child node and the new current goal be the old current goal.
- 2.2.3 If neither case 2.2.1 nor 2.2.2, then there is a rule derivation step from the current node to a child node using the same rule as R does from G_i to G_{i+1} . Let the new current node be that child node and the new current goal be G_{i+1} .

□

[†] q' being an instance of a subgoal, say $p \& C$, means q' is an instance of p and $C\sigma$ is not false, where σ is a mgu for q' and p .

Proposition 6.5 The correspondence algorithm terminates.

Proof During the proof we will look at the recursive calls to the algorithm itself as a macro expansion. Notice that every time the algorithm branches at case 2.2.3, the next goal in the SLD-refutation R becomes the new current goal. Since R is a sequence of finite goals, the algorithm terminates if we can prove that there is no infinite number of consecutive times the algorithm branches at either case 2.2.1 or 2.2.2.

First, there is no infinite number of consecutive case 2.2.1 because a node always looks for the answer providing subgoals from the nodes generated before it and there are only a finite number nodes generated before a particular node.

Second, there can only be a finite number of consecutive case 2.2.2 because each complement derivation step makes the selected subgoal of the new current node a "proper instance" of the corresponding subgoal of the old current node. Therefore there can only be a finite number of such instantiations.

Third, there can only be a finite number of consecutive either case 2.2.1 or 2.2.2. Let us first assume that in such a sequence an answer derivation step from a node only takes its siblings as its "answer providing goal". Since its siblings are also "proper instance" of their parent node, a similar argument to the one for case 2.2.2 applies and such a special sequence must be finite. Now for a general sequence including either case 2.2.1 or 2.2.2, we first identify all the special sub-sequence that have the property described above. For any step that does not belong to any such special sub-sequence, it can only be an answer derivation step

which takes an answer providing subgoal from a node other than its siblings. Right after each special sub-sequence, there must be such a step. Due to the restriction that a child node from a complement derivation step can only take answer providing subgoals from its sibling nodes or its parent's answer providing subgoals (see section 4.3.5.4), the node from which this answer derivation step takes answer providing subgoal must be a node generated before the node corresponding to the first step of that special sub-sequence. In a general sequence including either case 2.2.1 or 2.2.2, if we think each special sub-sequence as one step, then each "step" in the sequence looks for answer providing subgoals from the nodes generated before the node corresponding this current step. Therefore such sequence is finite.

The conclusion is that after a finite number of branches to either case 2.2.1 or 2.2.2 the algorithm must branch to case 2.2.3, which moves the current goal ahead in the SLD-refutation R . Thus the algorithm terminates since R is finite. $\square$

The only place in the algorithm that raises concern about whether a step can be carried out is at case 2.2.3 where it is assumed that the rule used in the SLD-refutation is also applicable to the corresponding selected subgoal of the current node. The purpose of the following three propositions is to justify this assumption.

Proposition 6.6 If H_2 is the resolvent of a goal $H_1 = A_1, \dots, A_j, \dots, A_k$ and a rule $C_0 :- C_1, \dots, C_m$ with A_j selected, and if K_2 is the resolvent of the goal $K_1 = H_1\sigma$ and the same rule with $A_j\sigma$

selected, where σ is a substitution, then there is a substitution σ' such that $K_2 = H_2\sigma'$.

Roughly speaking, the proposition alleges that if one goal is an instance of another, then its resolvent is also an instance of the corresponding resolvent of the other.

Proof Assume that $C_0 := C_1, \dots, C_m$ has no variables in common with $A_1, \dots, A_j, \dots, A_k$. Let ϕ and ψ be the mgu's used in the two derivations respectively; i.e., $H_2 = (A_1, \dots, C_1, \dots, C_m, \dots, A_k)\phi$ and $K_2 = (A_1\sigma, \dots, C_1, \dots, C_m, \dots, A_k\sigma)\psi$. Obviously $A_j\sigma\psi = C_0\psi$ since ψ is the mgu for $A_j\sigma$ and C_0 . Thus $A_j\sigma\psi = C_0\sigma\psi$ because σ has no effect on C_0 . i.e., $\sigma\psi$ is a unifier for A_j and C_0 . But ϕ is the mgu for A_j and C_0 , which means that $\phi\sigma' = \sigma\psi$ for some substitution σ' . Therefore,

$$\begin{aligned} K_2 &= (A_1\sigma, \dots, C_1, \dots, C_m, \dots, A_k\sigma)\psi \\ &= (A_1\sigma, \dots, C_1\sigma, \dots, C_m\sigma, \dots, A_k\sigma)\psi \\ &= (A_1, \dots, C_1, \dots, C_m, \dots, A_k)\sigma\psi \\ &= (A_1, \dots, C_1, \dots, C_m, \dots, A_k)\phi\sigma' \\ &= H_2\sigma' \end{aligned}$$

□

Proposition 6.7 During the execution of the correspondence algorithm, the ground companion of the current goal is always an instance of the conventional part of the current node in the GSI-tree.

Proof As in the proof for proposition 6.5, we will look at the recursive calls to the algorithm itself as a macro expansion. The proof is by induction on the number of times step 2 is executed in both the main loop and the recursive calls. The inductive assumption is that the ground companion of the current goal is an instance of the conventional part of the current node.

The first execution of step 2 is trivial since $G_0 = G_g$ is a ground instance of G .

If the algorithm branches at case 2.2.1, q' must be an instance of the answer providing subgoal since it is the condition for the algorithm to branch at case 2.2.1. For the next execution of step 2, the old current goal will remain as the new current goal and the node containing the answer providing subgoal will be the new current node. Note that the SLD-refutation used in the recursive call is the one obtained from the sub-refutation of R starting from the current goal by ignoring the subgoals which are irrelevant to the evaluation of q . For ground companions and nodes in the GSI-tree, the irrelevant subgoals are also ignored. Therefore q' being an instance of the answer providing subgoal already makes the inductive assumption true for the next execution of step 2.

If the algorithm branches at case 2.2.3, let $G_i = A_1, \dots, A_j, \dots, A_k$, $G_i' = G_i \theta_{i+1} \dots \theta_n \tau = A_1', \dots, A_j', \dots, A_k'$ and $G_i^f = A_1^f, \dots, A_j^f, \dots, A_k^f$ be the current goal, the ground companion of the current goal and the current node respectively. Note that the computation rule for the SLD-refutation follows the needs of this algorithm; therefore it is reasonable

to let A_j^i and A_j be the selected subgoals. Also let $C_0 := C_1, \dots, C_m$ be the rule used. G_{i+1} then is the resolvent from G_i and the rule. $G'_{i+1} = (A_1, \dots, C_1, \dots, C_m, \dots, A_k)\theta_{i+1}\theta_{i+2} \dots \theta_n \tau \tau'$ is the ground companion of G_{i+1} . A_j' and C_0 are unifiable since $A_j' = A_j\theta_{i+1} \dots \theta_n \tau$ and θ_{i+1} is the mgu for A_j and C_0 . Let G'' be the resolvent from G_i' and the rule. A_j^i and C_0 are unifiable since A_j' and C_0 are unifiable and A_j' is an instance of A_j^i by inductive assumption. Let G_{i+1}^i be the resolvent of G_i^i and the rule. By the proposition 6.6, G'' is an instance of G_{i+1}^i . If we can prove that G'_{i+1} is an instance of G'' , then G'_{i+1} is an instance of G_{i+1}^i , which is exactly the inductive assumption for the next execution of step 2. To prove that G'_{i+1} is an instance of G'' , let ξ be the mgu used in deriving G'' ; i.e., ξ is the mgu for C_0 and A_j' and $G'' = (A_1', \dots, C_1, \dots, C_m, \dots, A_k)\xi$. Since θ_{i+1} is a mgu for C_0 and A_j , $C_0\theta_{i+1} \dots \theta_n \tau = A_j\theta_{i+1} \dots \theta_n \tau = A_j'$. Therefore $\theta_{i+1} \dots \theta_n \tau$ is a unifier for C_0 and A_j' because A_j' is ground and $A_j'\theta_{i+1} \dots \theta_n \tau = A_j'$. But ξ is a mgu for C_0 and A_j' , there is a substitution ϕ such that $\xi\phi = \theta_{i+1} \dots \theta_n \tau$. Thus

$$\begin{aligned}
 G''\phi\tau' &= (A_1', \dots, C_1, \dots, C_m, \dots, A_k)\xi\phi\tau' \\
 &= (A_1', \dots, C_1, \dots, C_m, \dots, A_k)\theta_{i+1} \dots \theta_n \tau \tau' \\
 &= (A_1, \dots, C_1, \dots, C_m, \dots, A_k)\theta_{i+1} \dots \theta_n \tau \tau' \\
 &= G'_{i+1}
 \end{aligned}$$

In the following we will continue to use the same notation introduced above. If the algorithm branches at case 2.2.2, G_{i+1}^i can be considered a resolvent from G_i^i and a pseudo rule $A_j^i\sigma :- A_j^i\sigma$, where σ is the substitution used for that complement derivation step. $G'_{i+1} = G_i'$ can be considered a resolvent from G_i' and

the same pseudo rule because $A_j' = A_j \sigma \phi$, for some substitution ϕ , which is the condition for the algorithm to branch at case 2.2.2. According to the proposition 6.6, G'_{i+1} is an instance of G'_{i+1} . It is exactly the inductive assumption for the next round.

There is one more case we need to consider; i.e., when the algorithm returns from a recursive call. The old current node, the old current goal and its ground companion before the recursive call are G_i^t , G_i and G_i' respectively. Let G_r^t , G_r and G_r' be the new current node, the new current goal and its ground companion right after the returning from the recursive call. G_r^t is the resolvent of G_i^t and an answer, say A_a , which is used as a rule without the right hand side. Note that A_j' is an instance of A_a and $G_r' = A_1', \dots, A'_{j-1}, A'_{j+1}, \dots, A_k'$ assuming $G_i' = A_1', \dots, A'_{j-1}, A_j', A'_{j+1}, \dots, A_k'$. Therefore G_r' can be considered a resolvent from G_i' and the rule A_a . By the proposition 6.6, G_r' is an instance of G_r^t , which is the inductive assumption for the next round. $\square$

Proposition 6.8 In the correspondence algorithm, the rule applied to the current goal is also applicable to the goal of the current node in the GSI-tree.

Proof During the execution of the algorithm, the rule applied to a goal in the SLD-refutation is also applicable to the ground companion of the goal. This is already been mentioned and proven in the proof for the proposition 6.7. According to the proposition 6.7, a ground companion is always an instance of the corresponding node in the GSI-tree. Therefore, the rule is also applicable to that node. Also note that, by the switching lemma, we assumed that the SLD-

refutation would select the subgoal corresponding to the subgoal selected on the node. $\square$

The completeness theorem below proves that for a GSI-refutation found by the correspondence algorithm, the computed answer substitution α of the GSI-refutation has the property that G_g is an instance of $G\alpha$. Now, for a given SLD-refutation for P and G , there is a similar SLD-refutation for P and G_g ; the correspondence algorithm will terminate with a GSI-refutation starting from the root node in the GSI-tree. The completeness theorem insures that the "computed answer" corresponding to that refutation is equal to or more general than G_g .

Theorem 6.2 (Completeness theorem) Let P , G and G_g be as defined above; let α be the computed answer substitution of the GSI-refutation in the GSI-tree for P and G which is found by the correspondence algorithm. Then there is a substitution σ such that $G_g = G\alpha\sigma$.

Proof We only need to consider the outmost call to the algorithm. Let $X_1, \dots, X_v$ be all the variables appearing in G . Imagine that a special subgoal $s(X_1, \dots, X_v)$ were added to the goal G , correspondingly $s(X_1, \dots, X_v)\omega\omega'$ were added to G_g , where $\omega\omega'$ are the substitutions such that $G_g = G\omega\omega'$ as defined above. The predicate for this subgoal is specially chosen so that it is different from any predicates in G as well as in P . The goal of the terminal node of the GSI-refutation found by the correspondence algorithm which originally would have been an empty goal, is now $s(t_1, \dots, t_v)$. Obviously, $\alpha = \{X_1/t_1, \dots, X_v/t_v\}$. By the proposition 6.7, $s(X_1, \dots, X_v)\omega\omega'$ is an

instance of $s(t_1, \dots, t_v)$, i.e., there is a substitution σ such that $s(X_1, \dots, X_v)\omega\omega' = s(t_1, \dots, t_v)\sigma = s(X_1, \dots, X_v)\alpha\sigma$. Since $X_1, \dots, X_v$ are all the variables appearing in G , $G\omega\omega' = G\alpha\sigma$. Since $G\omega\omega' = G_g$, the theorem is proved. $\square$

7. Conclusions

The expert system described in chapter 3 appears to be a reasonably good solution for its application problem. The key feature of this application is that it requires fairly sophisticated reasoning about a relatively large volume of factual knowledge which is stored in an external database. The expert system is designed to do this in a relatively efficient way which takes advantage of contemporary database and logic programming technologies. An interesting feature of this design is its use of meta-interpreters to engineer the system's software; i.e., several different meta-interpreters are used for different purposes and they are combined together using meta-interpreter technology.

In one sense our query formulation method is a deviation from the normal Prolog execution strategy because certain goals are isolated, massaged by the meta-interpreter and evaluated by the DBMS instead of Prolog. The main reason for doing this is to make the DBMS queries selective. However, it should be possible to get more selectivity by additional deviation from the Prolog execution strategy. Storing query results in main memory imposes a limitation on our DBMS interface because the results may be too large. To a large extent this is offset by formulating queries so that they are very selective about the tuples retrieved. For our application the queries are sufficiently selective that the results easily fit in main memory. However, this limitation underscores the importance of formulating selective queries, at least for the kind of DBMS interface described in chapter 3.

The work on MOSES2 exposed several issues that complicated the design and implementation of it. For example, the user can not be asked the same question more than once; an interface is needed between two different evaluation strategies: Prolog's one tuple at a time and DBMS's all the tuples at a time. A solution at a more conceptual level to such problems would be desirable. A new inference engine featuring a graph search for logic programming was then designed. That is, the inference engine stores the answers of subgoals and make use of them for solving other subgoals later. It turns out that such an inference engine not only removes the need for special software to deal with the above issues, it also allows the expert system rules to be more declarative.

A graph search in logic programming raises new problems because a subgoal may have multiple answers, some of which may contain variables. This prevents the use of standard search algorithms for AND/OR graphs (e.g., [Nils82]) because they assume that each AND node can be solved independent of its siblings. This is not true when several AND subgoals contain common variables which makes the solution of one dependent on the solution of the others. Another complication is that two subgoals which are independent may have common answers but each may have answers which are different than the others. For example, subgoal $p(f(X), Y)$ and subgoal $p(W, b)$ may have some common answers, but $p(W, b)$ may also have some other answers that are not common with the answers for $p(f(X), Y)$. Therefore, to evaluate $p(W, b)$, we can use the answers from $p(f(X), Y)$, but we still must compute $A(p(W, b)) - A(p(f(X), Y))$, where $A(q)$ stands for answer set of q . This turns out to be the major challenge.

The first thing needed is a way to represent $G(p(W, b)) - G(p(f(X), Y))$, where $G(q)$ stands for the set of all ground instances of q . A result from [LaMM87] shows that there is no finite set of mgu's that can provide an explicit representation for this in the general case. This means some extension has to be made to logic programming. We decided to use a normal subgoal augmented with constraints to represent such a set difference. In the above example, $G(p(W, b)) - G(p(f(X), Y))$ is represented as $p(W, b) \& \{W \neq f(X)\}$.

The next thing needed is basic methods to manipulate constraints. We developed an algorithm for constraint generation, an algorithm for application of substitution (to constraints) and an algorithm for normalization (of constraints). These methods form a basis for our graph search interpreter, GSI. In addition to basic processing of constraints, these algorithms constitute a decision procedure for whether a constraint is a tautology or unsatisfiable. These methods are based on the work in [Colm84].

Using these basic methods for constraint processing, we developed a new resolution provided by GSI. There are three different derivation steps, namely, answer derivation steps, rule derivation steps and complement derivation steps. An answer derivation step makes use of a previous computed answer of another subgoal. Such a subgoal is called an answer providing subgoal. An answer providing subgoal may have some constraints that will disqualify some answer which may be needed by this "answer receiving goal". Those missed answers are covered by complement derivation steps. The rest of the answers are computed by rule derivation steps which correspond computing a resolvent in SLD-

resolution, but constraints are also involved in a rule derivation step. GSI needs some bookkeeping methods to register subgoals, keep the information about who provides answers to whom and to store answers for each registered subgoal in an answer list. We proved that GSI is sound and complete, which serves as the theoretic foundation of GSI. This means that GSI will not miss any answer if the search rule is fair and all the answers computed by GSI are logically correct.

The demand for space is a potential limitation of GSI since it requires all the answers be saved. But it is not a problem for many practical ES's because many existing ES's and ES shells use a graph search. Such systems must also save all answers to all subgoals and memory does not appear to be a problem for these applications. In addition, the graph search appears to have advantages since it is part of the design of many systems. However, for some application the memory requirement may be a difficulty.

7.1 Future Research

There are many places where some optimization can be done to make GSI more efficient. The reason we did not investigate them in an aggressive way is that our focus has been on giving a clear picture of the new inference engine so that the basic idea behind it can be understood easily. The following are illustrations of some possible future research work.

Consider two registered subgoals $p(X, Y)$ and $p(a, Z)$. One must be another's answer providing subgoal. As illustrated by the examples in section

5.1, it would be better if $p(X, Y)$ happen to be the answer providing subgoal for $p(a, Z)$ because this reduces the processing of constraints. Therefore, a possible optimization is to make the computation rule and the search rule "smarter" so that the selection of more general subgoals has a higher priority than less general ones.

Checking the answer list for duplicate answers and removing them has been suggested for GSI. Unfortunately it is not straightforward since answers are in a form of substitutions and two different answers may have common ground instances; e.g., answers $\{X/a\}$ and $\{Y/b\}$ for subgoal $p(X, Y)$. But a simple check for only alphabetic variants should be easy and beneficial.

Duplicate answers can even be introduced by GSI. Consider the case where $p(a, Y)$ and $p(X, b)$ are answer providing subgoals for $p(X, Y)$. $p(a, b)$ can be an answer for both $p(a, Y)$ and $p(X, b)$. $p(X, Y)$ then gets $p(a, b)$ twice since it picks answers from the answer lists of $p(a, Y)$ as well as of $p(X, b)$. This will not hurt the soundness and completeness of GSI. It will not cause repeated evaluation of "similar" subgoals either. But it may make GSI less efficient. It may also confuse the user if the duplicate answers showed up at the root level. There is a way to avoid this. Actually, it must be the case in the example that either $p(a, Y)$ is an answer providing subgoal for $p(X, b)$ or vice versa. Assume that $p(a, Y)$ is an answer providing subgoal for $p(X, b)$. The answers of $p(X, b)$ which come from $p(a, Y)$ can be marked so that when $p(X, Y)$ picks answers from the answer list of $p(X, b)$, those marked answers will be ignored. Therefore the answer $p(a, b)$ will only be picked up once by $p(X, Y)$, which is from the

answer list of $p(a, Y)$, where it is originally produced.

The computation rule is a strategy to select a subgoal in a goal to perform derivation steps. Once a subgoal is selected, GSI is committed to solving it before evaluating other subgoals. However, as search proceeds, it may appear advantageous to switch to another subgoal. It is not difficult to do this if completeness is not a concern. One way to keep the completeness is to discard all the work done on the previous selected subgoal and start all over again for the new subgoal. This is very inefficient, but the design of more efficient methods for less restrictive computation rules appears to be difficult.

Perhaps the most fruitful direction for future research is in specializing the methods of GSI for certain kinds of subgoals. Most of the examples in chapter 4 and 6 contains fairly general subgoals because they are necessary to illustrate the problems faced by GSI. However, the subgoals in chapter 5 only contained variables and constants (no function symbols) because it is difficult to find meaningful examples that use more complex terms. Thus, a practical version of GSI may use efficient method for processing a special class of subgoals, such as those that contain no function symbols, and other subgoals will be processed by some other methods, perhaps interactive methods. This thesis avoided such specialized methods because we wanted GSI to handle the general case. But, some kind of specialization is probably more practical; this is what many current ES's do in order to use an AND/OR graph search method. It should be possible to adapt the methods of GSI to efficiently search a considerably more general subgoal space than the kind found in contemporary ES's.

References

- [ApBo89] Apt, K. R., Bol, R. N. and Klop, J. W., On The Safe Termination of Prolog Programs, *Proc. of The Sixth International Conference on Logic Programming*, 1989, 353-368.
- [Bird80] Bird, R. S., Tabulation Techniques for Recursive Programs, *Computing Surveys*, Vol 12, No. 4, Dec. 1980, 403-417.
- [Bocc86a] Bocca, J., EDUCE - A Marriage of Convenience: Prolog and a Relational DBMS, *Proc. of the 1986 Symposium on Logic Programming*, 1986, 36-45.
- [Bocc86b] Bocca, J., On the Evaluation Strategy of EDUCE. *Proc. of the 1986 ACM-SIGMOD Conference on the Management of Data*, 1986, 368-378.
- [Bol90a] Bol, R. N., Generalizing Completeness Results for Loop Checks Report No. CS-R9025, Center for Mathematics and Computer Science, 1990.
- [Bol90b] Bol, R. N., Towards More Efficient Loop Checks, Report No. CS-R9026, Center for Mathematics and Computer Science, 1990.
- [BrWa84] Brough, D. R. and Walker, A., Some Practical Properties of Logic Programming Interpreters, *Proc. of The International Conference on Fifth Generation Computer Systems*, 1984, 149-156.
- [ChWa86] Chang, C. L. and Walker, A., PROSQL: A Prolog Programming Interface with SQL/DS, *Expert Database Systems*, L. Kerschberg editor, Benjamin/Cummings, 1986, 233-246.
- [Colm84] Colmerauer, A., Equations and Inequations on Finite and Infinite Trees, *Proc. of the International Conference on Fifth Generation Computer Systems*, 1984.
- [Diet87] Dietrich, S. W., Extension Tables: Memo Relations in Logic Programming, *Proc. Symposium on Logic Programming*, 1987, 264-272.
- [Disc88] Discenzo, F. M., Motor Selection Expert System, Report No. 019, Corporate Research and Development Engineering, Reliance Electric Co., Cleveland, OH 44117, 1988.
- [HeEr89] He, X., Ernst, G. W. and Discenzo, F. M., An Expert Systems Which

- Intelligently Accesses an External Database, *Proc. of the North American Conference on Logic Programming*, 1989.
- [Ioan88] Ioannidis, Y. E. et al., BERMUDA - An Architectural Perspective on Interfacing Prolog to a Database Machine, *Proc. of the Second International Conference on Expert Database Systems*, L. Kerschberg editor, Tysons Corner, 1988, 91-105.
- [JaLa87] Jaffar, J. and Lassez, J-L., From Unification to Constraints, *Proc. of the 6th Conference on Logic Programming*, 1987, 1-18.
- [LaMa87] Lassez, J-L. and Marriott, K., Explicit Representation of Terms Defined by Counter Examples, *Journal of Automated Reasoning*, 3, 1987, 301-317.
- [LaMM87] Lassez, J-L., Maher, M. J. and Marriott, K., Unification Revisited, *Foundations of Deductive Database and Logic Programming*, J. Minker editor, Morgan Kaufman, 1987.
- [Lloy84] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1984.
- [Nils82] Nilsson, N. J., *Principles of Artificial Intelligence*, Springer-Verlag, 1982.
- [PeSh??] Pereira, F. C. N. and Shieber, S. M., *Prolog and Natural-Language Analysis*, University of Chicago Press, ????
- [Rama??] Ramamohanarao, K. et al., *The NU-Prolog Deductive Database System*, ????
- [ScWa86] Sciore, E. and Warren, D. S., Towards an Integrated Database-Prolog System, *Expert Database Systems*, L. Kerschberg editor, Benjamin/Cummings, 1986, 293-305.
- [Shor76] Shortliffe, E. H., *Computer-based Medical Consultations: MYCIN*, American Elsevier, New York, 1976.
- [StBe89] Sterling, L. and Beer, R. D., Meta-Interpreters for Expert System Construction, *J. of Logic Programming*, 1989.
- [Ster86] Sterling, L., Meta-Interpreters: The Flavors of Logic Programming?, *Proc. of Workshop on Foundations of Logic Programming and Deductive Databases*, Washington, 1986.

- [StLa86] Sterling, L. and Lalee, M., An Explanation Shell for Expert Systems, *Computational Intelligence*, 1986, 136-141.
- [StSh86] Sterling, L. and Shapiro, E., The Art of Prolog, MIT press, Cambridge, Massachusetts, 1986.
- [StYa89] Sterling, L. and Yalcinalp, L. U., Explaining Prolog-Based Expert Systems Using a Layered Meta-Interpreter, *Proc. of 11th International Joint Conference in Artificial Intelligence*, 1989.
- [TaSa86] Tamaki, H. and Sato, T., OLD Resolution with Tabulation, *Proc. Third International Conference on Logic Programming*, 1986, 85-98.
- [Ullm88] Ullman, J.D., Principles of Database and Knowledge-base Systems, Computer Science Press, 1988.
- [VaCJ83] Vassilou, Y., Clifford, J. and Jarke, M., How Does an Expert System Get Its Data?, *Proc. of 9th International Conference on VLDB*, 1983, 70-72.
- [VaCJ85] Vassilou, Y., Clifford, J. and Jarke, M., Access to Specific Declarative Knowledge by Expert Systems: The Impact of Logic Programming, *Decision Support Systems*, 1 (1985), 123-141.
- [Warh83] Warren, D. H. D., An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, 1983.
- [Warr89] Warren, D. S., THE XWAM: A Machine that Integrates Prolog and Deductive Database Query Evaluation, manuscript, 1989.
- [YaSt89] Yalcinalp, L. U. and Sterling, L., An Integrated Interpreter for Explaining Prolog's Successes and Failures, *Proc. of the Workshop on Meta-Programming in Logic Programming*, MIT Press, 1989.
- [Zani86] Zaniolo, C., Prolog: A Database Query Language for All Seasons, *Expert Database Systems*, L. Kerschberg editor, Benjamin/Cummings, 1986, 219-232.
- [ZhOz90] Zhou, N. and Ozsoyoglu, Z. M., Multiple Query Optimization, manuscript, 1990.

Appendix

A. The Rules of MOSES2

The entire rule set for MOSES2 is given in the following. The predicates in the rules are either system predicates, such as *nonvar* which checks whether its argument is a variable or not, "=" which checks if the two argument of it are equal or not, etc.; or user defined predicate, such as *main_goal*, *calculate*, etc. Arithmetic calculations and comparisons are also system predicates. They includes "+", "-", "*", "/", "=", "<=", "<", ">", ">=", "is", etc, where "is" is assignment. Among the user defined predicates, there are several special ones which are not defined by rules (or facts), they are database relation names instead. These special predicates are *main*, *field*, *frames*, and *remarks*.

Some user defined predicates are not defined here. They are mainly utilities and therefore are not included in this rule set. These predicates can be divided into several groups. The first group includes the predicates which are used to interact with users. *getinput* is responsible for collecting the user's input on certain basic application parameters. *report_query* shows the user the database query MOSES2 intends to send to DBMS and asks if he wants the query to be executed. *report_and_ask* report the motors selected and ask the user if he wants to select more. *end_statement* simply tells the user the search for motors is over.

The second group are those predicates which access the fine-table. *finds(line-speed, Value)* will find the value of line speed from the find-table and instantiate *Value* with that value; *putin(line-speed, 5000)* will make the value of

line speed 5000 in the find-table. the value for line speed as 5000. *modify* is currently same to *putin*, it is reserved for modifying the find-table in a way which is different than *putin*. *add-low* and *add-high* are similar to *putin*, but the new value replaces the old value in the find-table only when it is lower(higher) than the old one.

The third group includes the rest of utility predicates which do variety of miscellaneous jobs. e.g., *init* does the initiation; and *endw* does the final cleaning.

```
main_goal :- init, getinput, calculate, retrieve, endw.
```

```
calculate :- rpm_min, rpm_max, hp_tension, hp_decel, hp_min, fw_rpm_low,
            hp_low, hp_high, motor_type.
```

```
retrieve :- finds(substitution, Sub), Sub \== yes, finds(motor_type, Mt),
            report_query(Mt, Reply), user_choose(Mt, Reply).
```

```
retrieve :- finds(substitution, Sub), Sub == yes,
            report_query(substitution, Reply), user_choose(substitution, Reply).
```

```
user_choose(Mt, go_on) :- create(Mt), retract(n_of_motors(N)),
                          report_and_ask(N, Mt, More), maybe_more(Mt, More).
```

```
user_choose(Mt, skip) :- next_search(Mt).
```

```
user_choose(Mt, quit) .
```

```
maybe_more(Mt, more) :- next_search(Mt).
```

```
maybe_more(Mt, quit) .
```

```
next_search('RPMIII') :- modify(motor_type, 'SUPER_RPM'), retrieve.
```

```
next_search('SUPER_RPM') :- modify(motor_type, 'OBSOLETE'), retrieve.
```

```
next_search('OBSOLETE') :- substitute, modify(substitution, yes), retrieve.
```

```
next_search('OBSOLETE') :- \+ substitute, end_statement.
```

```
next_search(substitution) .
```

```
create('RPMIII') :- create_motors(rpmIII_motors).
```

```
create('SUPER_RPM') :- create_motors(super_rpm_motors).
```

```
create('OBSOLETE') :- create_motors(obsolete_motors).
```

```
create(substitution) :- create_motors(substitution_motors).
```

```

create_motors(IDB_name) :- asserta(n_of_motors(0)), motors(Design),
                           T=..[IDB_name, Design], assertz(T),
                           retract(n_of_motors(N)), N1 is N+1,
                           asserta(n_of_motors(N1)), fail.
create_motors(IDB_name) .

motor_type :- finds(motor_type, Mt), nonvar(Mt).
motor_type :- finds(motor_type, Mt), var(Mt), deduce_motor_type.

deduce_motor_type :- finds(hp_low, Hl), Hl <= 150, putin(motor_type, 'RPMIII').
deduce_motor_type :- finds(hp_low, Hl), Hl > 150,
                       putin(motor_type, 'SUPER_RPM').

rpm_min :- finds(line_speed, Ls), finds(diameter_max, Dx), Ri is Ls/(Dx*3.1416),
           putin(rpm_min, Ri).

rpm_max :- finds(line_speed, Ls), finds(diameter_min, Di), Rx is Ls/(Di*3.1416),
           putin(rpm_max, Rx).

fw_rpm_low :- finds(rpm_max, Rx), Frl is Rx*0.6, add_low(fw_rpm_low, Frl).

hp_tension :- finds(tension, Ts), finds(line_speed, Ls), Ht is Ts*Ls/33000,
              putin(hp_tension, Ht).

hp_decel :- finds(paper_inert, Ip), finds(steel_inert, Is), finds(line_speed, Ls),
            finds(stop_time, T), finds(diameter_max, Dx), finds(rpm_min, Ri),
            Temp1 is 2*(Ip+Is)/(T * Dx * 1930), Temp2 is Temp1 * Ls/5250,
            Hd is Temp2 * Ri, putin(hp_decel, Hd).

hp_min :- finds(hp_tension, Ht), finds(hp_decel, Hd), H is (Ht+Hd)/2,
          max(Ht, H, Hd), putin(hp_min, H).

hp_low :- finds(hp_min, Hi), add_low(hp_low, Hi).

hp_high :- finds(hp_min, Hi), Hh is 2 * Hi, add_high(hp_high, Hh).

motors(Design) :-
    finds(hp_tension, Ht), finds(hp_decel, Hd), finds(rpm_max, Rx),
    motors1(Design, Hp, Base_rpm, Fw_rpm, Winding, Motor_type, Cost,
            Mult_field, Tsel, Tdec),
    Y is (Fw_rpm - 3*Base_rpm)*(2*Hp - Ht - Hd)/Hp + Fw_rpm,
    Rpm_lim is Y/Rx, Rpm_lim > 0.7,
    low_inert_cost(Tsel, Tdec, F1),
    winding_cost(Winding, F2),
    motor_type_cost(Motor_type, F3),
    field_cost(Mult_field, F4),
    Effective_cost is Cost*(1+F1+F2+F3+F4+1-Rpm_lim),

```



```

assertz(all_motors(Design, Hp, Base_rpm, Winding, Motor_type, Cost,
Mult_field, Tsel, Tdec, Rpm_lim, Effective_cost)).

motors1(Design, Hp, Base_rpm, Fw_rpm, Winding, Motor_type, Cost, Mult_field,
Tsel, Tdec)
:- finds(frame_change, Fc), Fc == yes, finds(hp_low, Hl),
finds(hp_high, Hh), finds(base_rpm_low, Brl),
finds(fw_rpm_low, Frl), finds(fw_rpm_high, Frh),
finds(motor_type, Motor_type), finds(frame_series_low, Fsl),
finds(frame_series_high, Fsh), finds(frame_type, Frame_type),
finds(enclosure, Enclosure), finds(rpm_min, Ri),
finds(remark, Remark),
main(Design,_,Enclosure, Hp, Avolts, Base_rpm, Fw_rpm, Winding, Frame,
Duty, Ovld_pct, Ovld_time, Fvolts, Service_factor, Altitude, Cost,
Motor_type),
field(Design,_,_,_,_,_,_,_,_,_,Tsel,Tdec,_,_,_,_,_,_,_,_,_,Mult_field),
frames(Frame,_,_,_,_,_,_,_,_,_,Frame_type,Frame_series,_,_,_,_,_,_,_,_,_),
/* fred(-----), */
Hl=<Hp, Hp=<Hh, Brl=<Base_rpm, Frl=<Fw_rpm, Fw_rpm=<Frh,
Fsl=<Frame_series, Frame_series=<Fsh, Base_rpm =< Ri*Hp/Hl ,
impose_remarks(Design, Remark, _ ).

motors1(Design, Hp, Base_rpm, Fw_rpm, Winding, Motor_type, Cost, Mult_field,
Tsel, Tdec)
:- finds(frame_change, Fc), Fc \== yes, finds(hp_low, Hl),
finds(hp_high, Hh), finds(base_rpm_low, Brl),
finds(fw_rpm_low, Frl), finds(fw_rpm_high, Frh),
finds(motor_type, Motor_type), finds(enclosure, Enclosure),
finds(rpm_min, Ri), finds(remark, Remark),
main(Design,_,Enclosure, Hp, Avolts, Base_rpm, Fw_rpm, Winding, Frame,
Duty, Ovld_pct, Ovld_time, Fvolts, Service_factor, Altitude, Cost,
Motor_type),
field(Design,_,_,_,_,_,_,_,_,_,Tsel,Tdec,_,_,_,_,_,_,_,_,_,Mult_field),
/* fred(-----), */
Hl=<Hp, Hp=<Hh, Brl=<Base_rpm, Frl=<Fw_rpm, Fw_rpm=<Frh,
Base_rpm =< Ri*Hp/Hl ,
impose_remarks(Design, Remark, _ ).

impose_remarks(Design, Remark, Rmk_value) :-
nonvar(Remark), remarks(Design, Remark, Rmk_value).
impose_remarks(Design, Remark, Rmk_value) :-
nonvar(Rmk_value), remarks(Design, Remark, Rmk_value).
impose_remarks(Design, Remark, Rmk_value) :-
var(Remark), var(Rmk_value).

if_low_inertia :- finds(app, W), nonvar(W), low_inertia(W).

```

```

low_inert_cost(Tsel, Tdec, 0.2) :- if_low_inertia, Tsel>Tdec.
low_inert_cost(Tsel, Tdec, 0) :- if_low_inertia, Tsel <= Tdec.
low_inert_cost(_, _, 0) :- \+ if_low_inertia.

```

```

winding_cost(Winding, 0.1) :- Winding == straight_shunt.
winding_cost(Winding, 0) :- Winding \== straight_shunt.

```

```

motor_type_cost(Mt, 0.05) :- Mt == 'SUPER_RPM'.
motor_type_cost(Mt, 0.1) :- Mt == 'OBSOLETE'.
motor_type_cost(Mt, 0) :- Mt == 'RPMIII'.

```

```

field_cost(M, 0.4) :- M > 1 .
field_cost(M, 0) :- M <= 1 .

```

```

substitute :- sub1.
substitute :- sub2.
substitute :- sub3, modify(frame_change, yes).

```

```

sub1 :- finds(enclosure, En), En == teao, finds(base_rpm_high, X), X >= 1750,
      finds(base_rpm_low, Y), Y <= 1750,
      modify(enclosure, tefc), modify(motor_type, 'RPMIII'),
      add_low(base_rpm_low, 1750), add_high(base_rpm_high, 1750).
sub2 :- finds(enclosure, E), nonvar(E), member(E, [dpgfv,spgfv]),
      finds(base_rpm_high, X), X >= 1750, finds(base_rpm_low, Y), Y <= 1750,
      modify(enclosure, dpg), modify(motor_type, 'RPMIII'),
      add_low(base_rpm_low, 1750), add_high(base_rpm_high, 1750).
sub3 :- finds(enclosure, E), nonvar(E), member(E, [dpgfv,spgfv]),
      finds(base_rpm_high, X), X >= 2500, finds(base_rpm_low, Y), Y <= 2500,
      modify(frame_type, c), modify(frame_series_low, 21),
      modify(frame_series_high, 36), modify(enclosure, dpg),
      modify(motor_type, 'RPMIII'), add_low(base_rpm_low, 2500),
      add_high(base_rpm_high, 2500).

```

B. The Rules with Graph Search

This section shows the new rules which assume the use of the graph search interpreter (GSI). These rules are a rewritten form of the rules of MOSES2 given in Appendix A.

```

main_goal :- init, motor_type(Motor_type), retrieve(Motor_type), endw.

```

```

retrieve(Motor_type) :-
    report_query(Motor_type, Reply), user_choose(Motor_type, Reply).

user_choose(Motor_type, go_on) :-
    report_and_ask(Motor_type, More), maybe_more(Motor_type, More).
user_choose(Motor_type, skip) :- next_search(Motor_type).
user_choose(Motor_type, quit) .

maybe_more(Motor_type, more) :- next_search(Motor_type).
maybe_more(Motor_type, quit) .

next_search('RPMIII') :- retrieve('SUPER_RPM').
next_search('SUPER_RPM') :- retrieve('OBSOLETE').
next_search('OBSOLETE') :- substitute(yes), retrieve('SUBSTITUTE').
next_search('SUBSTITUTE') .

report_and_ask(Mt, More) :-
    motors(Mt, Design, Hp, Base_rpm, Motor_type, Rpm_lim, Effective_cost),
    report_motor(Mt, Design, Hp, Base_rpm, Motor_type, Rpm_lim, Effective_cost),
    fail.
report_and_ask(Mt, More) :- get_reply(More).

motors(Mt, Design, Hp, Base_rpm, Motor_type, Rpm_lim, Effective_cost) :-
    hp_tension(Ht), hp_decel(Hd), rpm_max(Rx),
    motors1(Mt, Design, Hp, Base_rpm, Fw_rpm, Winding, Motor_type, Cost,
        Mult_field, Tsel, Tdec),
    Y is (Fw_rpm - 3*Base_rpm)*(2*Hp - Ht - Hd)/Hp + Fw_rpm,
    Rpm_lim is Y/Rx, Rpm_lim > 0.7,
    low_inert_cost(Tsel, Tdec, F1),
    winding_cost(Winding, F2),
    motor_type_cost(Motor_type, F3),
    field_cost(Mult_field, F4),
    Effective_cost is Cost*(1+F1+F2+F3+F4+1-Rpm_lim).

motors1('SUBSTITUTE', Design, Hp, Base_rpm, Fw_rpm, Winding,
    Motor_type, Cost, Mult_field, Tsel, Tdec)
:- frame_change(yes), hp_low(Hl), hp_high(Hh), s_base_rpm_low(Brl),
    s_base_rpm_high(Brh), fw_rpm_low(Frl), fw_rpm_high(Frh),
    frame_series_low(Fsl), frame_series_high(Fsh), frame_type(Frame_type),
    s_enclosure(Enclosure), rpm_min(Ri), Motor_type='RPMIII',
    remark(Remark),
    main(Design,_,Enclosure,Hp,Avolts,Base_rpm,Fw_rpm,Winding,Frame,
        Duty,Ovld_pct,Ovld_time,Fvolts,Service_factor,Altitude,Cost,
        Motor_type),
    field(Design,_,_,_,_,_,Tsel,Tdec,_,_,_,Mult_field),
    frames(Frame,_,_,Frame_type,Frame_series,_,_,_,_),
    Hl=<Hp, Hp=<Hh, Brl=<Base_rpm, Base_rpm=<Brh, Frl=<Fw_rpm,

```

```

Fw_rpm=<Frh, Fsl=<Frame_series, Frame_series=<Fsh,
Base_rpm =< Ri*Hp/Hl, impose_remarks(Design, Remark, _ ).

```

```

motors1('SUBSTITUTE', Design, Hp, Base_rpm, Fw_rpm, Winding,
Motor_type, Cost, Mult_field, Tsel, Tdec)
:- \+ frame_change(yes), substitute(yes),
hp_low(Hl), hp_high(Hh), s_base_rpm_low(Brl),
s_base_rpm_high(Brh), fw_rpm_low(Frl), fw_rpm_high(Frh),
s_enclosure(Enclosure), rpm_min(Ri), Motor_type='RPMIII',
remark(Remark),
main(Design,_,Enclosure,Hp,Avolts,Base_rpm,Fw_rpm,Winding,Frame,
Duty,Ovld_pct,Ovld_time,Fvolts,Service_factor,Altitude,Cost,
Motor_type),
field(Design,_,_,_,_,_,Tsel,Tdec,_,_,_,_,Mult_field),
Hl=<Hp, Hp=<Hh, Brl=<Base_rpm, Base_rpm=<Brh, Frl=<Fw_rpm,
Fw_rpm=<Frh, Base_rpm =< Ri*Hp/Hl,
impose_remarks(Design, Remark, _ ).

```

```

motors1(Mt, Design, Hp, Base_rpm, Fw_rpm, Winding, Motor_type, Cost,
Mult_field, Tsel, Tdec)
:- Mt \== 'SUBSTITUTE', Mt==Motor_type, hp_low(Hl), hp_high(Hh),
base_rpm_low(Brl), base_rpm_high(Brh), fw_rpm_low(Frl),
fw_rpm_high(Frh), enclosure(Enclosure), rpm_min(Ri), remark(Remark),
main(Design,_,Enclosure,Hp,Avolts,Base_rpm,Fw_rpm,Winding,Frame,
Duty,Ovld_pct,Ovld_time,Fvolts,Service_factor,Altitude,Cost,
Motor_type),
field(Design,_,_,_,_,_,Tsel,Tdec,_,_,_,_,Mult_field),
Hl=<Hp, Hp=<Hh, Brl=<Base_rpm, Base_rpm=<Brh, Frl=<Fw_rpm,
Fw_rpm=<Frh, Base_rpm =< Ri*Hp/Hl ,
impose_remarks(Design, Remark, _ ).

```

```

impose_remarks(Design, Remark, Rmk_value) :-
nonvar(Remark), remarks(Design, Remark, Rmk_value).
impose_remarks(Design, Remark, Rmk_value) :-
nonvar(Rmk_value), remarks(Design, Remark, Rmk_value).
impose_remarks(Design, Remark, Rmk_value) :-
var(Remark), var(Rmk_value).

```

```

if_low_inertia :- app(W), nonvar(W), low_inertia(W).

```

```

low_inert_cost(Tsel, Tdec, 0.2) :- if_low_inertia, Tsel>Tdec.
low_inert_cost(Tsel, Tdec, 0) :- if_low_inertia, Tsel <= Tdec.
low_inert_cost(_, _, 0) :- \+ if_low_inertia.

```

```

winding_cost(Winding, 0.1) :- Winding == straight_shunt.
winding_cost(Winding, 0) :- Winding \== straight_shunt.

```

```

motor_type_cost(Mt, 0.05) :- Mt == 'SUPER_RPM'.
motor_type_cost(Mt, 0.1) :- Mt == 'OBSOLETE'.
motor_type_cost(Mt, 0) :- Mt == 'RPMIII'.

field_cost(M, 0.4) :- M > 1 .
field_cost(M, 0) :- M <= 1 .

substitute(yes) :- sub1(yes).
substitute(yes) :- sub2(yes).
substitute(yes) :- sub3(yes).

frame_change(yes) :- \+ sub1(yes), \+ sub2(yes), sub3(yes).

sub1(yes) :- enclosure(teao), base_rpm_high(X), X >= 1750, base_rpm_low(Y),
            Y <= 1750.
sub2(yes) :- enclosure(E), nonvar(E), member(E, [dpgfv,spgfv]),
            base_rpm_high(X), X >= 1750, base_rpm_low(Y), Y <= 1750.
sub3(yes) :- \+ sub2(yes), enclosure(E), nonvar(E), member(E, [dpgfv,spgfv]),
            base_rpm_high(X), X >= 2500, base_rpm_low(Y), Y <= 2500.

rpm_min(Ri) :- line_speed(Ls), diameter_max(Dx), Ri is Ls/(Dx*3.1416).
rpm_max(Rx) :- line_speed(Ls), diameter_min(Di), Rx is Ls/(Di*3.1416).

fw_rpm_low(Frl) :- rpm_max(Rx), Frl is Rx*0.6.

hp_tension(Ht) :- tension(Ts), line_speed(Ls), Ht is Ts*Ls/33000.

hp_decel(Hd) :- paper_inert(Ip), steel_inert(Is), line_speed(Ls), stop_time(T),
            diameter_max(Dx), rpm_min(Ri), Temp1 is 2*(Ip+Is)/(T * Dx * 1930),
            Temp2 is Temp1 * Ls/5250, Hd is Temp2 * Ri.

hp_min(Hi) :- hp_tension(Ht), hp_decel(Hd), H is (Ht+Hd)/2, max(Ht, H, Hi).

hp_low(Hl) :- hp_min(Hl).

hp_high(Hh) :- hp_min(Hi), Hh is 2 * Hi.

motor_type('RPMIII') :- hp_low(Hl), Hl <= 150.
motor_type('SUPER_RPM') :- hp_low(Hl), Hl > 150.

diameter_max(Dx) :- ask('maximum paper roll diameter', Dx).
diameter_min(Di) :- ask('minimum paper roll diameter', Di).
tension(Ts) :- ask('paper tension', Ts).
paper_inert(Ip) :- ask('paper roll inertia', Ip).
steel_inert(Is) :- ask('steel core inertia', Is).
stop_time(T) :- ask('stop time', T).

```

```

line_speed(Ls) :- ask('line speed', Ls).
remark(Rk) :- ask('remark', Rk).
base_rpm_low(Brl) :- ask('base rpm lower bound', Brl).
base_rpm_high(Brh) :- ask('base rpm upper bound', Brh).
fw_rpm_high(Frh) :- ask('field weakened rpm upper bound', Frh).
enclosure(Encl) :- ask('enclosure', Encl).
s_base_rpm_low(1750) :- sub1(yes).
s_base_rpm_low(1750) :- sub2(yes).
s_base_rpm_low(2500) :- sub3(yes).
s_base_rpm_high(1750) :- sub1(yes).
s_base_rpm_high(1750) :- sub2(yes).
s_base_rpm_high(2500) :- sub3(yes).
s_enclosure(tefc) :- sub1(yes).
s_enclosure(dpg) :- sub2(yes).
s_enclosure(dpg) :- sub3(yes).
frame_series_low(21) :- sub3(yes).
frame_series_low(Fsl) :- ask('frame series lower bound', Fsl).
frame_series_high(36) :- sub3(yes).
frame_series_high(Fsh) :- ask('frame series upper bound', Fsh).
frame_type(c) :- sub3(yes).
frame_type(Ft) :- ask('frame type', Ft).

```