NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

NL-339 (r. 88/04) c

Canada

Learning of Disjunctive Concepts with Explanation-based Learning

Maude Salembier-Pelletier

Thesis submitted to

the School of Graduate Studies and Research

in partial fulfillment of the requirements for the Master degree in Computer Science

Computer Science Department
University of Ottawa
November, 1990

ISBN 0-315-68076-8

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

A Yves

et

à ma fille Andréane

qui a fait ses premiers pas

dans le monde cette année

Acknowledgement

I would like to thank my supervisor Stan Matwin for his guidance in the course of this research. The contribution of Florence d'Alché was helpful and appreciated in the early phase of this thesis and of other publications derived from it. The environment provided by the Ottawa Machine Learning Group allowed me to meet other researchers and provided an excellent platform to exchange ideas.

I would like to thank my husband Yves for the support and understanding he gave me throughout this project. I must also thank my parents, whose support over these years and emphasis on the importance of study, made the completion of this work possible. *Merci!*

Learning of Disjunctive Concepts with Explanation-Based Learning

Table of Contents

List of figures

List of tables

The next two tables show the notation and abbreviations used throughout this thesis.

DIF	De-instantiated facts
EBG	Explanation-Based Generalization
FOL	First-order logic
LODIC	Learning Of DIsjunctive Concept
MCTD	Mixed Connective Tissue Disease
NH	Non-Horn clause
SLD	Selective Linear clause resolution
SLE	Systemic Lupus Erythematosis

Abbreviations

Ai	antecedent i
Bi	input clause i
Ci	clause i
Li	literal i
Qi	consequent i
S	set of all input clauses
w	weight
wc	weight of a clause
T	conjunction of all clause of domain theory
G	is the goal concept definition
E	conjunction of facts describing the training example
D	the statement that the example belongs to the concept
=>	logical implication
¬, not	logical not

Notation

Chapter 1

Introduction

1.1 The problem

The ability to learn is one of the fundamental attributes of intelligence [Michalski et al. 1983]. A natural task of learning is to handle uncertainties, which it is evident are a realistic part of our lives. Human decision-making and problem solving are often done in an environment where information concerning a problem is partial or approximate. A medical diagnosis, for example, or a financial investment involve uncertainties.

Uncertainties can be categorized in many ways [Kanal et al. 1986]: we are interested in those which represent a lack of knowledge or a lack of confidence in the source of information. Many times, humans reach uncertain conclusions from insufficient premises. For example, walking along a dark street, an animal suddenly runs in front of you. A glance tells you the animal has four legs and a tail. You may conclude that this animal is a cat or a small dog. The conclusion is uncertain because of the scarcity of information, but nevertheless shows a plausible reasoning.

In the field of Machine Learning, we want to be able to represent and manage such uncertainties. It is important for learning that the information present in the domain somehow finds its way into use. There are different approaches within Machine Learning to deal with uncertainties: they may be reduced, eliminated, or included in the process. In our research, the latter form is of interest. In particular, we attempt to retain uncertainties as a part of the process of learning.

The learning process transforms information provided by the environment into some new form which is stored for future use. The nature of this transformation determines the type and context of learning. The present research uses Explanation-Based Learning as the learning strategy, and attempts to formalize and develop a solution to the following problem:

> How can uncertainty be included in Explanation-Based Learning domain theory (the initial environment) and in the operational concept definition (a usable form of the initial information) ?

1.2 Basic definitions

Before proceeding, those terms whose use is widespread, and which are adopted throughout this thesis, will be defined.

Term: A term is either a *variable* or has the form FT, where F is a function of $n \geq 0$ arguments and T is a sequence of n terms. A *constant* is a function of zero arguments. *Variables* are denoted with the first letter in upper case type; *functions* and *constants* are denoted by lower case letters.

Atom: An atom has the form RT, where R is a predicate of $n \geq 0$ arguments and T is a sequence of n terms. *Predicates* are denoted by lower case letters.

Literal : A literal is either an atom or the negation of an atom, e.g., $\neg$ p(X). A positive literal is often called a *predicate* in Explanation-Based Learning terminology.

Clause: A clause is a set of literals, denoting a formula with a conjunction of antecedents implying a disjunction of consequents (a conclusion), e.g., *A1 and A2..and An => Q1 or Q2*

Horn clause: Clauses containing at most one consequent and only positive antecedents.

Non-Horn clause: Clauses that are not Horn clauses, i.e. involve a disjunction in the consequent or negation in the antecedents.

Empty clause (□): The empty clause is the set of no literals. It denotes the value false.

Input clause : An input clause is an element of the set of clauses representing a problem.

Resolvent: Given two clauses, A and B, that are of the form A = A′∪ {C} and B′ ∪ {¬C}, where A′ and B′ are (possibly empty) disjunctions, a resolvent of A and B is the clause A′ ∪ B′, and the literal resolved upon is C.

Substitution: A substitution is an assignment of terms to variables. Only one term is assigned to any given variable. It is convenient to represent a substitution as {t1 / v1, t2 / v2, ...} which means v1 replaces t1, v2 replace t2, ...

Satisfiability: A set of clauses S is unsatisfiable if and only if it is not satisfiable. It is satisfiable if and only if all its clauses are true. To demonstrate that a set of clauses logically implies the empty clause is sufficient to show that it is unsatisfiable.

1.3 Motivation

To date, explanation-based generalization (EBG) [Mitchell et al. 1986] produces only conjunctive concept definitions. In [Kedar-Cabelli and McCarty 1987], EBG is presented as an augmentation of resolution theorem proving for Horn clauses. But resolution is not restricted to Horn clauses and sometimes first-order Horn clause logic is too restrictive to explain a real-life application domain. The purpose of this research is to extend the EBG

framework to learn disjunctive theories. Disjunctions of the form *A1 and A2..and An => Q1 or Q2 ... or Qn* might appear in the domain theory or the training example. This form of expression represents uncertainty or lack of knowledge about the domain. For example, the rule `arthritis and rash and fever => mctd or sle` represents the uncertainty of the domain theory : symptoms like `arthritis, rash, and fever` can suggest either the mixed connective tissue disease (`mctd`) or the systemic lupus erythematosis (`sle`) .

Non-Horn clauses are the means by which disjunctions can be expressed. Although it can be shown that most problems can be expressed by means of Horn clauses [Kowalski 1979], some problems are expressed more naturally only through non-Horn clauses. Moreover, it is appropriate for the learned concept to represent the domain's uncertainty by including disjunctions in the concept definition. By re-expressing the problem in a Horn clause, natural correspondence between described objects and their representing predicates is often lost. Maintaining this information, as well as allowing a more natural way of expressing some application domains, motivates this research.

A simple example illustrates this point. Consider the following medical problem. The symptoms of arthritis, rash, and fever suggest either systemic lupus erythematosis (SLE) or mixed connective tissue disease (MCTD). Laboratory tests can generally distinguish these two conditions, even though both diseases are treated the same (often with a steroid drug such as prednisome). The concept to learn is the treatment. The concept definition can be learned at different levels of abstraction, for instance at the level of symptoms or at the level of diseases.

Re-expressing the problem by means of Horn clauses does not adequately represent the relation between the symptoms (`rash, fever, arthritis`) and the diseases (`mctd, sle`). There are two methods to re-express this problem in Horn clause logic.

The first creates new predicates to replace the negation of predicates. For example, the first rule of the disease example

```
arthritis and rash and fever => mctd or sle
```

can be written as:

```
arthritis and rash and fever and non_mctd => sle
arthritis and rash and fever and non_sle => mctd
```

but if non_mctd or non_sle is not known, then the problem is unsolvable. This may often be the case: we can diagnose one of the two diseases without being able to rule out or positively select either disease. Also, the notion of uncertainty introduced by sle or mctd is lost and cannot be included in the concept definition. Non_mctd or non_sle is an unnatural way to express predicates.

The second method merges rules and writes them as Horn clauses. For example, the first three rules of the disease example

```
arthritis and fever and rash => sle or mctd
sle => tests
mctd => tests
```

can be merged into:

```
arthritis and fever and rash => tests
```

This works, but changes the interpretation of the problem. Once again, the notion of uncertainty, introduced in the domain theory by the use of "sle or mctd" in the consequent of a rule, is lost. This formula "sle or mctd" might be the operational concept, and for this reason we want to keep it in the proof.

1.4 Contributions

This section briefly enumerates the main contributions of the thesis, which are developed in detail later.

• The extension of the Explanation-Based Generalization (EBG) framework to allow disjunctions to be used in the domain theory or in the training example.

• The utilization of a resolution algorithm possessing the fundamental characteristics of EBG, such as constraining input clauses to be selected from the domain theory only.

• Heuristics that allow disjunction or uncertainty to be expressed in the final concept description.

• Extension of the operationality criterion to be dynamic and granular.

• No necessity for the user of the learner to input explicitly the operationality criterion.

• Flexible ways of acquiring the final concept description.

1.5 Outline

Our system LODIC (Learning of DIsjunctive Concept) uses Explanation-Based Generalization (EBG) with a resolution technique to build the explanation. Concepts basic to both are reviewed in chapter 2. The next two chapters describe the EBG problem and the first step of the EBG method. Chapter 3 presents the problem and the method of its resolution. Chapter 4 extends this method to include non-Horn clauses by the means of two heuristics. This is followed, in chapter 5, by the second step of the EBG method: the generalization and acquisition of the operational concept. Flexible ways of defining operationality are proposed. In chapter 6, a general analysis of the LODIC system for completeness and soundness is conducted. This is followed by a discussion of related work in the two distinct areas of the research: theorem proving and operationality. A general algorithm to implement LODIC is presented in chapter 8. Finally, in chapter 9, conclusions are drawn and suggestions for future work are made.

Chapter 2

Basic concepts of EBG and resolution

2.1 Review of EBG

Explanation-Based Generalization (EBG) [Mitchell et al. 1986] consists of two parts: the EBG problem and the EBG method. The EBG problem is defined by four parameters: the goal concept, the training example, the domain theory and the operationality criterion. The EBG problem representation used in this research is described in section 3.1.

The EBG method is a two step approach: explanation and generalization.

1. EBG proves that a *goal concept* holds for a single positive *training example*, using a *domain theory* of rules and facts about the domain. In particular, the system must explain to itself why the training example is a member of the concept under study. Therefore, EBG is assumed to be given a definition of the goal concept, as well as a domain knowledge for constructing the required explanation.

2. EBG then generalizes the proof of the instance, defining a larger class of instances that are justified by the same generalized explanation within the domain theory. It extracts a description of that larger class of instances from an explanation structure, which constitutes a generalization of the training instance. Expressions are generalized if they satisfy an operationality criterion. The operationality criterion requires that the learned concept definitions be correct examples of the goal concept, and also be in a usable (operational) form.

The first step of our method is described in chapter 3 and 4. The second step is explained in chapter 5.

Explanation-based learning can be viewed as a search through the space of possible explanations of a concept definition. The EBG method as presented by [Kedar-Cabelli et al. 1987] is viewed as an augmentation of SLD-resolution theorem proving for Horn clause logic. Theorem proving is a good idea because it drives the search for the explanation so that the explanation is always obtained whenever it exists, and the EBG generalization is just a slight modification of resolution and can be obtained as a by-product of theorem proving. Theorem proving also facilitates the complexity analysis of EBG. As EBG problems scale-up, EBG needs to be augmented and modified. In this research, EBG is extended to allow the acquisition of disjunctive concepts and to extend the process of generalization.

We now turn to how does this extension of learning affects the choice of proof technique.

2.2 Statement of the problem

This research extends the EBG framework to allow the acquisition of disjunctive concept definitions. To do so, the inference engine of EBG must allow the use of theories with non-Horn clauses. Such clauses are the means by which disjunctions are expressed. The inference engine of EBG must also extend EBG generalization to allow the specification of any set of predicates such as a disjunction to be operational, and provide a mechanism to collect the concept definition under such constraints.

We also want to preserve the current characteristics of the implementation of EBG in Prolog [Kedar-Cabelli and McCarty 1987] such as guaranteeing that only clauses belonging to the domain theory participate in the proof. Other resolution strategies, such as

binary resolution, tend to rely on the lemmas obtained during the proof. Adding intermediate lemmas to the theory is inconsistent with the functioning of EBG: EBG constructs an explanation in terms of the domain theory that proves how the example satisfies the goal concept definition.

2.3 Review of existing proof techniques and strategies

The two proof techniques of input resolution and lock resolution provide algorithms that meet our requirements. The first technique described is input resolution which is a refinement of linear resolution. The second technique is lock resolution.

2.3.1 Linear resolution

Linear resolution starts with a clause, resolves it against another clause to obtain a resolvent, and resolves this, in turn against some other clause until the empty clause □ is obtained. A formal definition of *Linear resolution is:* Given a set S of clauses and a clause C_0 in S, a linear deduction of C_n from S with top clause C_0 is a deduction where :

1. for $i = 0,1,\ldots,n-1$, C_{i+1} is a resolvent of C_i (called a *centre* clause, or a *resolvent*) and B_i (called a *resolving* clause)

2. each B_i is either in S, or is a C_j for some j, $j<i$.

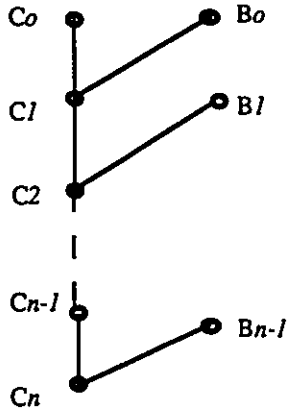

Fig. 2.1 Linear resolution structure.

Fig. 2.1 illustrates the principle of linear deduction. Figure 2.1 also introduces the term *resolving clause* to denote an argument in the resolution step.

Linear resolution introduces previous resolvent clauses *Cj* into the proof. This is undesirable within the EBG context because EBG relies on the proof to bring only relevant domain knowledge into the concept definition. For this reason, input resolution is used instead.

2.2.1.1 Input Resolution

Input resolution is a refinement of linear resolution in which every side clause, *Bi*, is in the set *S* and hence is an input clause. Input resolution is an incomplete but efficient refinement of linear resolution, much easier to implement than linear resolution, and although not complete, more computationally efficient than linear resolution. The sacrifice of completeness for efficiency is compensated by the fact that input resolution is sufficient to prove a large class of theorems.

2.3.2 Lock resolution

The second technique explored is lock resolution, it is a refinement of resolution which uses a concept similar to that of ordered clauses. Lock resolution uses indices to order literals of clauses in *S*. Each occurrence of a literal of the set *S* is arbitrarily indexed with an integer. Literals of lowest index in each clause are then able to be resolved. The literals in resolvents inherit their indices from their parent clause; if a literal has more than one possible inherited index, the lowest index is assigned to this literal. A formal definition of *lock resolution* is: let *C1* and *C2* be two clauses with no common variables and with every constituent literal indexed. Let *L1* and *L2* be two literals with the "lowest" index of *C1* and C2. If *L1* and $\neg L2$ have a most general unifier σ, then *C* is the clause obtained from *C1* σ and C2 σ by removing *L1* σ and *L2* σ.

For example, let *S* be the following set of clauses:
(1) $_1P \vee {}_2Q$

(2) $_3P \vee _4\neg Q$

(3) $_6\neg P \vee _5Q$

(4) $_8\neg P \vee _7\neg Q$

From clauses (1) to (4), there is only one lock resolvent:

(5) $_6\neg P$ from clauses (3) and (4)

From clauses (1) to (5), there is only two lock resolvents:

(6) $_2Q$ from clauses (1) and (5)

(7) $_4\neg Q$ from clauses (2) and (5).

Finally, resolving clause (6) and (7), we obtain the empty clause □ .

2.3.3 Proof Strategy

Strategy is a guidance to a theorem prover on how to approach the proof. Some strategies are *ordering strategies*, directing the choice of the next clause or literal to be resolved. Other strategies, called *restriction strategies*, avoid considering combinations of clauses irrelevant to the proof.

Set of support strategy

Set of support : Given a set of unsatisfiable clauses, $S = A \cup B$, if A is a satisfiable set of clauses, then B is a set of support for S . A subset B of a set S is called a *set of support* S if S-T is satisfiable. Typically, the set of support is assumed to be the clause(s) that refutes the theorem to be proved. To prove the theorem, we are essentially proving that A1 $\wedge$ A2 $\wedge$ ... $\wedge$ An $\wedge \neg$ B is unsatisfiable. Since A1 $\wedge$ A2 $\wedge$ ... $\wedge$ An is usually satisfiable, it might be wise to avoid resolving clauses in A1 $\wedge$ A2 $\wedge$ ... $\wedge$ An. This is what the set of support strategy tries to accomplish.

Weighting strategy

Weighting strategy assigns priority to terms, clauses and concepts. It reflects knowledge or intuition about how the proof should proceed. Weighting is a restriction and ordering strategy.

2.4 The proof technique chosen

As we have seen, linear resolution does not meet the requirements of EBG, but input resolution, an efficient refinement of linear resolution, does. Input resolution can be strengthened by interpolating two strategies. The first is weighting of the clauses, which greatly increases its efficiency, and the second is the set of support with the denial of the goal concept.

The weighting strategy was inspired by lock resolution, even though lock resolution itself was not chosen. Lock resolution is incompatible with the set of support strategy, and any combination with other strategies destroys the completeness of the method. Despite this, lock resolution is very efficient, and minor problems can be overcome. The major obstacle to the adoption of this method is that it does not give any support for the operationality problem.

The chosen proof technique is based on input resolution with a set of support strategy and a weighting strategy. Input resolution has the advantages of being easy to implement, having a simple structure to maintain, and guaranteeing that clauses belonging to the domain theory participate in the proof. In addition, heuristic methods such as the set of support strategy can be conveniently applied. The weighting strategy was inspired from lock resolution by ascribing weight to the literals, and by guiding the resolution to the choice of the lightest one. The weighting strategy can also help to create classes or

partitions of the explanation into levels according to the weight of literals involved in the proof. The weighting strategy is discussed in more detail in chapter 3.

Chapter 3

Resolution

This chapter delineates the problem representation used to extend EBG. The weighting strategy is developed for two general cases. First, the propositional logic case is introduced, and then first-order predicate logic. Then, a resolution step is explained using the weighting strategy as an ordering strategy.

3.1 Problem representation

EBG may be viewed variously [Ellman 1989]. The view adopted here is of EBG making a non-operational goal concept definition operational. The EBG problem begins with the non-operational goal concept and a domain theory. A training example is given to EBG by the expert.

Inputs to the EBG problem are in Skolemized clausal form. The notation used in our discussion is: D is the statement that the example belongs to the concept. T represents the conjunction of all the clauses contained in the domain theory. G is the goal concept definition. In first-order logic, G is instantiated from the example. E represents the conjunction of facts describing the training example.

The theorem to be proved can be stated as :

$$T \wedge G \wedge E \Rightarrow D$$

The following sub-sections describe how examples are represented throughout this report. The first two examples, `disease` and `mushroom` are expressed in propositional

logic. The last examples, `alpine club` and `block`, are expressed in first-order predicate logic. Each of these examples has a peculiarity that helps illustrate the contents of this research.

3.1.1 Disease example

The `disease` example is a problem taken from the medical domain [Musen 1990]. The object is to learn when to apply a particular treatment. The interest of this example lies in the coincidence of symptoms of systematic lupus erythematosis and mixed connective tissue disease. Consequently, this rule must be used in the proof. It is also possible to stop operationality at the level of information provided by this rule. To do so would produce a disjunctive concept definition.

```
D :   treatment (concept to be learned)
G :   diagnostic_of_sle        => treatment
      diagnostic_of_mctd       => treatment
T :   arthritis & rash & fever => sle or mctd
      sle              => tests
      mctd             => tests
      tests            => diagnostic_of_sle
      tests            => diagnostic_of_mctd
E:    arthritis
      fever
      rash
```

3.1.2 Mushroom example

This simple example expresses some typical beliefs concerning mushrooms and toadstools [Kowalski 1979]. The goal concept is to learn which mushrooms are poisonous. This problem is expressed more naturally with non-Horn clauses than in Horn

logic. The predicates need not be renamed by a negative literal when non-Horn clauses are used. This problem could be re-expressed using a Horn-clause, but doing so would introduce literals that represent the negation of existing literals. For example, `not mushroom` can be represented as the `nonmushroom` literal, thus eliminating the negation of the non-Horn expression, and allowing Horn clauses to be used. But what is the relation between the clauses containing the literals `mushroom` and `nonmushroom`? Does this formulation preserve the meaning of the domain theory? If the problem is expressed using non-Horn clauses, there is no ambiguity and the problem is expressed naturally.

```
D :   poisonous
G :   toadstool          => poisonous
T :   fungus             => toadstool or mushroom
      boletus            => fungus
      boletus            => not mushroom
E:    boletus
```

3.1.3 Alpine club example

This is an example of multiple non-Horn clauses. The problem consists of disjunctions in the consequent and negated literals in the antecedents. The first non-Horn clause `clubmember(X) and climber(X) and not skier(X) => is_climber(X)` has a negative literal among the antecedents. When this clause is transformed into clausal form, there is more than one positive literal; note that this clause is logically equivalent to `clubmember(X) and climber(X) => skier(X) or is_climber(X)`. The second non-Horn clause included in the domain theory is

clubmember(X) => skier(X) or climber(X), and is composed of two disjuncts in the conclusion.

```
D :   is_climber(X)
G :   clubmember(X) and climber(X) and not skier(X)
                          => is_climber(X)
T :   clubmember(X)     => skier(X) or climber(X)
      not likes(X,snow) => not skier(X)
      not likes(X,snow)=> not likes(X,rain)
      likes(tony,X)     => not likes(mike,X)
      not likes(tony,X)=> likes(mike,X)
E :   clubmember(tony)
      clubmember(mike)
      clubmember(john)
      likes(tony,rain)
      likes(tony,snow)
```

3.1.4 The block example

The block example belongs to the class of problems described using non-Horn clauses, which is the plan formation task [Kowalski 1979]. Plan formation tasks may require the construction of conditional plans from a disjunctive solution. Here, the concept to be learned is how to put the maximum of two numbers a and b in location loc. In this formulation, both states and statements are regarded individually, and are represented by terms. A statement "X holds true in a state Y" is represented by a binary relationship holds(X,Y).

```
D :   put_max(loc,a,b)
```

```
G :   holds(val(loc,X),W)    and    max(a,b,X)    =>
  put_max(loc,a,b)
T :   numb(U) and numb(V) =>smaller(U,V) or smaller(V,U)
      smaller(V,U)     => max(U,V,U)
      smaller(U,V)     => max(U,V,V)
      location(U)      => holds(val(U,V),assign(U,V,W))
      holds(X1,W) and diff(X2,val(X2,Y)) and location(U)
                       => holds(X1,assign(U,V,W))
E :   numb(a)
      numb(b)
      location(loc)
```

3.2 Weighting strategy

The purpose of the weighting strategy is twofold: to guide the proof process, and to prune the search process to acquire a new concept definition. The second purpose is discussed in chapter 5.

The weighting strategy is used to order and restrict the search path. The ordering strategy is a breadth-first search, ordering the search to consider more promising paths first. The restriction strategy restricts the choice to relevant clauses.

At every resolvent clause, a heuristic evaluates the next best literal to resolve, and works forward from that point. To achieve this goal, weights are attributed to all the literals appearing in the problem representation. These weights distinguish the literals belonging to *E, T, G* and *D*. The heuristic chooses to resolve first the lightest literal from among those in the current clause. If the lightest literal cannot be resolved by any clause from domain

theory, then this branch fails. The proof graph begins with the most general clauses, and goes on to the most specific ones, which are the relevant facts of the training example (see figure 3.1).

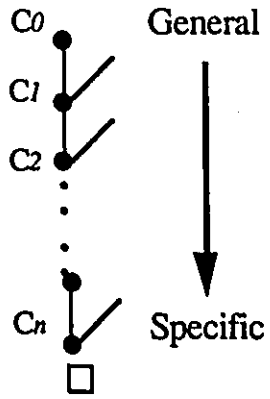

Fig. 3.1 Proof graph going from most general clauses to most specific ones.

3.2.1 Attributing weights to the literals

The first step of the weighting strategy is to allocate a weight to all literals. A weight pair that includes both the literal and its weight, is created to encapsulate this information. The collection of these pairs forms the weight database.

In the propositional case, weights of literals are attributed from a range of 0 to 4, as follows:

- the goal concept weighs 0 (lightest)
- the goal concept definition literal(s) weigh 1
- a literal of the domain theory weighs 2, with the exception of literals belonging to the goal concept, the goal concept definition and the operational literals.
- each operational literal weighs 4[1] (heaviest).

---

[1]Here, weight of 4 is used for consistency with the first-order logic case. Weight of 3 is not needed in propositional logic.

The predicate `operational` [Hirsh 1988; Kedar-Cabelli and McCarty 1987; Mitchell et al. 1986], stating which literals may appear in the operational concept definition is replaced in our research by assigning a weight of 4 to the operational literal, even though generally, a weight of 4 is associated with facts in the training example. Hereafter, we refer to literals of weight 4 as facts taken from the training example. The extension to consider other literals operational is left to chapter 5.

An atom has only a single weight assigned to it throughout the proof (i.e. p and $\neg$ p have the same weight). The weights of literals are assigned in the following order:

1. the goal concept is assigned its weight (w=0)
2. weights are assigned to goal concept definition literals (w=1)
3. facts are weighed (w=4).
4. All remaining literals with weight unassigned are given a weight of 2.

In the case of the `mushroom` example, the result of the attribution of weights is shown in table 3.1.

Literal:	Weight	Reason:
`poisonous`	0	goal concept
`toadstool`	1	goal concept definition
`boletus`	4	training example (fact)
`fungus`	2	domain theory
`mushroom`	2	domain theory

Table 3.1: Weight of literals of the `mushroom` example.

In first-order logic (FOL), the weighting strategy is refined so as to handle different instantiations of literals. The same literal might appear in several different clauses with a

different instantiation, because the weight of a literal is assigned according to the category of the clause in which the literal appears. For example, literal `max(a,b,X)` that appears in the goal concept definition of the `block` example weighs 1. A different instantiation of the same literal, `max(U,V,U)`, that appears in the domain theory, is assigned a weight of 2.

The weights are categorized as follows:

- the instance of the goal concept weighs 0 (lightest).
- the goal concept definition literal(s) instantiated by the goal concept weigh 1.
- a literal of the domain theory weighs 2, if and only if this literal has a different instantiation from the same literal that might appear in the goal concept, goal concept definition, literals of weight 4 and/or the de-instantiated fact database (for the definition of de-instantiated facts see below).
- de-instantiated facts weigh 3.
- each literal (fact) of the training example weighs 4 (heaviest).

As for the propositional case, the literals that participate in the goal concept definition are assigned their weight first. In addition, each literal of the goal concept is instantiated with constants that represent the individual example for which EBG is done. For example, in the `alpine club` example the goal concept definition is:

```
clubmember(X) and climber(X) and not skier(X) =>
    is_climber(X)
```

The goal concept instantiated with constants that represent the individual example is:

```
clubmember(mike) and climber(mike) and not skier(mike) =>
    is_climber(mike)
```

which describes "is `mike` a clubmember and a climber but not a skier?"

The literals of the training example (facts) are then weighed. The facts are usually fully instantiated. The next step is to derive an extra level of abstraction from the facts. This level is referred to as the de-instantiated fact database.

Each literal of weight 4 is transformed in the de-instantiation process. Each constant of the fact-literal not equal to constants in the instance of the goal concept is turned into a variable. If at least one variable is introduced in the literal then this new literal becomes part of the de-instantiated fact database, and is referred to as the de-instantiated fact (DIF). DIFs are assigned a weight of 3. A DIF can be partially or fully de-instantiated. The purpose of the DIF database is to create a level of abstraction different from facts. In the `alpine club` example, the only constant in the goal concept is `mike`. The fact `likes(tony,snow)` creates the de-instantiated fact `likes(X,Y)` because `tony` and `snow` are different from `mike`. The fact `clubmember(mike)`, though, does not create a new DIF.

Once all DIFs are created, all remaining literals of the domain theory are weighed. If the literal can be subsumed by a DIF then no weight is assigned to it, otherwise a weight of 2 is assigned. A literal is subsumed if and only if there exists a more general literal than the subsumed literal. For example, `likes(X,Y)` subsumes `likes(tony,Z)`.Therefore, `likes(tony,Z)` has no weight associated with it, and no weight pair is created for this particular instance of the literal `likes`. Keeping only the subsumed version of a literal saves memory space in the database and restricts the search space when matching literals.

The result of assigning weights to the `alpine club` example are shown in table 3.2. Weights are assigned in the order shown in the table.

Literals	Weight	Reason:
is_club_climber(mike)	0	goal concept instantiated
clubmember(mike)	1	instantiated literal of the goal concept definition
climber(mike)	1	instantiated literal of the goal concept definition
skier(mike)	1	instantiated literal of the goal concept definition
clubmember(mike)	4	literal of the training example (fact)
clubmember(tony)	4	literal of the training example (fact)
clubmember(john)	4	literal of the training example (fact)
likes(tony,snow)	4	literal of the training example (fact)
likes(tony,rain)	4	literal of the training example (fact)
clubmember(X)	3	de-instantiated fact
likes(X,Y)	3	de-instantiated fact
skier(X)	2	domain theory
climber(X)	2	domain theory

Table 3.2 : attribution of weights for the `alpine club` example

3.2.2 Weights of clauses.

We define the weight of a clause as the weight of the lightest literal contained by that clause. Because of the manner in which weight is introduced, the weight of a clause is the weight of its most general literal.The weights of clauses are used for two purposes.

Firstly, the weight of input clauses from domain theory and the training example are employed when searching for a match with the literal to resolve upon. This is the restriction strategy of the weighting strategy.

Secondly, when performing the proof, the weight of a resolvent clause is calculated. Each time a new resolvent is introduced in the proof, the weight of the resolvent clause is calculated. In the propositional case, a literal has only one weight. In the FOL case,

occurrences of the same literal in different resolvents can have different weights because arguments are instantiated by unification.

In chapter 5, we introduce the idea of an operationality level. The weights of resolvent clauses partition the proof graph into ***operationality levels***. Each level in the proof graph contains all clauses of the same weight.

- level zero contains the negation of the training instance of the goal concept.
- level i for i= 1,2,3 contains all clauses with at least one literal of weight i.
- level four contains clauses with all literals of weight 4 - this is the most specific level.

3.2.3 Weighting strategy: A control strategy and a restriction strategy

The control strategy evaluates a resolvent clause *Ci* to find the lightest literal from among all literals of clause *Ci* . Each literal weighted is then compared to the lightest one yet encountered. In the propositional case, the weighting strategy consists of straight matching with the weight database. First-order logic (FOL) case is more complex, since unification must be taken into account.

The rules for matching literals in FOL in the resolution process are:

1. If there is a single literal P′ with weight *w* that subsumes literal P, then assign weight *w* to literal P.

1a. If there is more than one literal in the weight database that subsumes P, then choose the most general one (i.e., the one with the lowest weight).

2. If there is no literal that subsumes F, then find the most general literal P′ with the same predicate name as P, and assign to P the weight of P′.

For example, in the propositional case, weighing the literal `toadstool` from the `mushroom` example would give a weight of 1 (see table 3.1). In the FOL case (see table 3.2), the literal `likes(tony,mike)` is subsumed by the literal `likes(X,Y)` and is

assigned a weight of 3 (the weight of literal `likes(X,Y)`). It does not match the literal `likes(tony,rain)` of weight = 4 or the literal `likes(tony,snow)` of weight = 4, because neither subsumes the literal `likes(tony,mike)`.

The weighting strategy is used here as a control strategy, but can also be considered as a restriction strategy. The resolution is made more efficient by pruning the search space of invalid clauses for the matching process. Only clauses with a weight equal to or greater than the weight of the literal selected for resolution are considered.This is more effective than undertaking a search of all clauses of the domain theory. This strategy consequently avoids introducing a lighter literal into the proof. In the propositional case, the weight of the clause is simply equal to that of the selected literal, because each literal is only a single proposition. Thus, if the selected literal is part of the clause, then the weight of the clause must be equal to its weight, because clauses with smaller weights are not considered.

The weighting strategy is different for FOL, as different instantiations of a literal may differ in weight. In this case, the weight of the potential clause is relevant if the weight of the clause is equal to or greater than the weight of the literal. More general clauses (i.e., those with the smallest weight) are always tried first. This results in the graph being developed from more general to more specific clauses. If two equally weighted clauses are found, LODIC can select one arbitrarily or apply a more sophisticated conflict resolution heuristic.

To demonstrate the FOL case, let us introduce the problem of putting the maximum of two numbers *a* and *b* in location *loc.*: this is called the `block example`. At one point in the proof of the `block example`, the literal ¬`holds(val(loc,X),W)` of weight = 1, is selected to be resolved. The possible input clauses matching the selected literal and their weights (*wc*) are:

```
        (1)                              (1)                    (0)
1. ¬holds(val(loc,X),W)  v ¬ max(a,b,X)  v put_max(loc,a,b),
   wc= 0

        (4)                    (2)
2. ¬location(U)  v holds(val(U,V),assign(U,V,W),  wc=2

         (2)                (2)                  (4)
3. ¬holds(X1,W)  v ¬diff(X2,val(X2,Y)  v ¬location(U)

                   (2)
    v holds(X1,assign(U,V,W),  wc=2
```

Of these three clauses, only the second and third input clauses are valid for the literal ¬holds(val(loc,X),W), whose weight is 1. Clause 1 does not qualify as its clause weight is 0, and 0 is smaller than the weight of the literal ¬holds(val(loc,X),W). In this particular case, clause 1 is the lemma that introduced ¬holds(val(loc,X),W).

3.3 Resolution algorithm

The resolution algorithm is an input resolution refinement with a set of support strategy and a weighting strategy. This method is efficient, and relies on a simple proof structure (see figure 3.2). Moreover, the resolution algorithm can be made more efficient by reducing the search space.

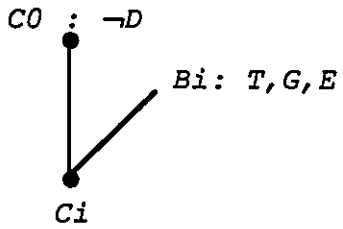

Fig. 3.2 Simple linear structure

3.3.1 Refinement of the resolution algorithm

The search space is reduced by efficiently detecting failure branches and infinite branches.

The first technique involves pruning search paths which could not lead to a solution. Forcing the process to resolve upon only the lightest literal, results in the shortest possible proof. In the resolution process, if the selected literal (the lightest) cannot be resolved upon, then this branch has no solution, as all input clauses that can be applied are known. The path cannot be extended into a complete sequence of choices that is satisfiable. Therefore, LODIC stops exploring that branch and backtracks. Pruning failure branches reduces the search space.

Resolution has no mechanism to prevent an inference which has already being carried out from being attempted once more in exactly the same way. The second technique avoids redundancy by preventing the system from going into loops in the unification algorithm, stopping it from using "circular" binding. This process, referred to as *occur check* in the Logic programming literature [Llyod 1984], ensures that literals already solved are not reintroduced in the proof. Each time new literals are introduced in the solution, they are matched against a list of already-solved literals. If a literal is an alphabetic variant of any of the already-solved literals, then resolution fails. Two terms are alphabetic variants if they are identical except for the variable names, and variables are mapped one-to-one in corresponding positions. For example, `a(X,Y)` and `a(Z,Z)` are not alphabetic variants, but `a(X,Y)` and `a(W,Z)` are.

3.3.2 Resolution Algorithm

The theorem $(T \wedge G \wedge E \Rightarrow D)$ is proven by refutation : we show that $(\neg D \wedge T \wedge G \wedge E)$ is unsatisfiable. In our case, input clauses consist of the clausal representation of T, G and E.

Resolution algorithm:

Step 0 Weigh all literals of *D,G,T,E*.

Step 1 Choose ¬D to be the top clause *C0*. This is the set of support strategy ss, where ss = ¬D.

Step 2a Select the literal ¬P to be resolved in clause *Ci* using the weighting strategy described in section 3.2.3. That is, choose the lightest literal ¬P from among literals of clause *Ci*.

Step 2b Assign the weight of literal P to the weight of *Ci*.

Step 3a IF there exists at least one matching clause *Bi* in the set of input clauses *T,G,E* that resolves ¬P, then *not_solved* becomes true if no literals already solved are introduced again in the proof.

OTHERWISE if no input clauses match then fail and backtrack.

Step 3b IF *not_solved* then resolve with *Bi*

OTHERWISE {introduction of an already-solved literal} fail and backtrack.

Step 4 Clause *Ci + 1* (the resolvent) is the new clause formed by the logical consequence of *Ci and Bi*.

Step 5 If *Ci + 1* is the empty clause □, then terminate. Otherwise, set *Ci = Ci + 1* and return to step 2a.

3.3.3 Demonstration of a resolution step

Here we describe the process realizing a single resolution step. *C* is the current centre clause. Assuming that literal $\neg P$ has already been selected to be resolved in *C*, there are three possible cases:

1. **a Horn-resolving clause** $B : \neg A \vee P$ **is found :** classical resolution can be applied.

2. **no Horn-clause is adequate to resolve $\neg P$ but there is a non-Horn candidate** $NH : \neg A \vee P \vee Q$ **to resolve** $\neg P$: specific heuristics described in chapter 4 (the line heuristic and diamond heuristic) must be used to treat this case.
3. **no clause is found to resolve** $\neg P$: this means the branch leads to a failure. Backtracking is performed to find the first alternative branch to continue the proof.

Figure 3.2 demonstrates a resolution step in case 2: a non-Horn clause is used to resolve away the selected literal. The underlined literal is the one selected by the weighting strategy to be resolved. The literal `¬smaller(a,b)` is selected because it has the smaller weight (2) of the resolvent clause `¬smaller(a,b) v ¬location(loc)`. The clause chosen to resolve away the literal is the NH: `¬numb(U) v ¬numb(V) v smaller(U,V) v smaller(V,U)`. This clause has a weight of 2, and so is a relevant clause to employ. By substituting `a for U`, and `b for V`, we obtain the next resolvent. The new resolvent clause is weighted, the lightest literal of the resolvent clause becomes the new selected literal to resolve away (`smaller(b,a)`), and the next step of resolution is performed.

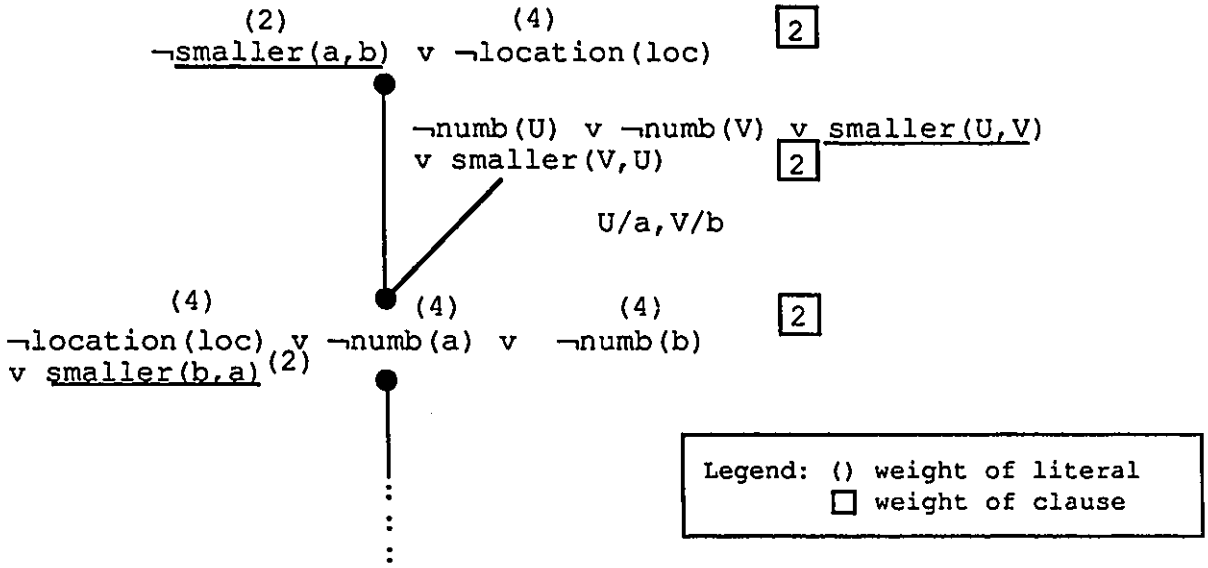

Fig. 3.3 A resolution step from the `block` example proof graph.

In the event that the literal $\neg P$ can be resolved by a Horn clause and a non-Horn NH clause, the Horn clause is preferred except when the non-Horn clause NH has been used before. This allows the proof structure to remain as simple as possible. This justification is explained in detail in chapter 4.

Chapter 4

The non-Horn heuristics

Input resolution alone is sufficient to demonstrate a given set of Horn clauses is inconsistent. Often it is adequate for some, but not all non-Horn clause problems. Combining input resolution and special heuristics extends the class of problems that can be resolved. We propose heuristics to deal with the non-Horn clauses and to apply the next step of resolution. Two heuristics are discussed: the line heuristic and the diamond heuristic. These are the means by which a disjunction is introduced in the proof.

4.1 The line heuristic

The line heuristic is an extension of the current input resolution. The non-Horn clause is used as an Horn clause. The literal $\neg P$ of the centre clause Ci is resolved away with the non-Horn candidate. The new centre clause $Ci + 1$ is the logical consequence of the two clauses less the literal P.

The line heuristic is always the first to be tried in dealing with non-Horn clauses, because it keeps the proof simpler than the diamond heuristic. If it fails, the diamond heuristic can be applied. The only condition on applying the line heuristic is that the non-Horn candidate has never been used before in the current proof. This restriction is applied if the non-Horn has been used before, and the branch where it was used led to a failure. In this situation there is a reasonable assumption that applying the non-Horn clause again as a resolving clause will not result in a success. If the failure was caused by one or more literals of the non-Horn clause or their descendants, then reintroducing the same clause will result in failure, for the same reason. It is only advisable to reapply the non-Horn candidate

if the cause of the previous failure is the literal being resolved at this point. However, for this to be true, LODIC must follow each literal introduced by the non-Horn clause and its descendants until there is nothing else that can be resolved.

Currently, in our system, this approach is not feasible because the weighting strategy stops the proof as soon as the lightest literal cannot be resolved away, or the occur check succeeds. Therefore, it is impossible to determine which literals remain unresolved at the point of failure.We believe the benefit of pruning the search space when developing the proof is greater than the cost of keeping track of the literals causing the failure and applying the line heuristic again.

To illustrate the line heuristic, assume that $NH : \neg A \vee P \vee Q$ is a candidate to resolve $\neg P$, and that NH has not yet been used in the proof. First, we try the line heuristic to resolve $\neg P$ with $(\neg A \vee Q) \vee P$. If resolution succeeds, the proof is finished (Fig 4.1a).We can observe in this case that a linear proof is performed, which explains the name of the heuristic. Otherwise, linear resolution does not succeed and we deal with the clause obtained so far (see Fig. 4.1b).

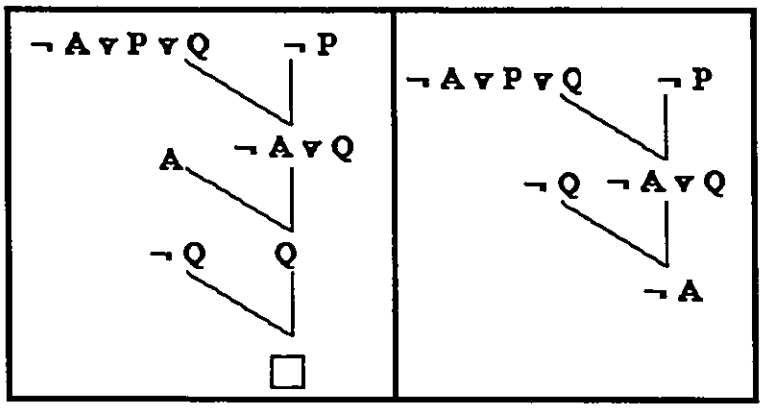

Fig. 4.1. a) Non-Horn clause with linear resolution succeeds.

b) A literal remains, consequently, NH cannot be used to resolve ¬P linearly.

If one or more literals remain unsolved (see Fig. 4.1b), this means the non-Horn clause cannot resolve $\neg P$ linearly. This kind of failure necessitates backtracking to find either an alternative resolving clause for $\neg P$ or an alternative to the branch that has introduced $\neg P$.

In the backtracking process, NH is saved as a non-Horn clause to be used with $\neg P$. We need to find other literals of the consequent of the non-Horn clause to continue the proof. In the previous example, literal $\neg Q$ is the remaining consequent of NH. Backtracking is performed to the point where $\neg P$ was introduced in the resolution graph, and a search is made for an alternative for $\neg P$. At this point, resolution is resumed. If the wanted literals reappear, then NH will be used for resolution using the diamond heuristic.

4.1.1 Justification of the line heuristic

The resolution principle underlies the line heuristic. The resolution principle states:

> "For any two clauses C1 and C2, if there is a literal L1 in C1 that is complementary to a literal L2 in C2, then delete L1 and L2 from C1 and C2 respectively and construct the disjunction of the remaining clauses." [Chang et al. 1973]

Hence, the line heuristic that applies a non-Horn clause to the resolvent is valid in terms of the resolution principle. The line heuristic is merely an extension of normal input resolution.

4.1.2 An example using the line heuristic

The `alpine club` is an example that uses the line heuristic. The goal concept is to learn what characterizes a club member who is a climber but not a skier. In this example, `mike` is a specific instance of the goal concept to be learned.

The first step of the EBG method is to produce an explanation, using the rules of the domain theory, of why the training instance satisfies the goal concept. The explanation takes the form of a proof graph, as seen in figure 4.2.

The salient interest of this example is the requirement of the line heuristic to prove that the denial of the goal concept is unsatisfiable. At resolvent R2, the only clause relevant to resolve away the literal `¬climber(mike)` is the non-Horn clause `¬clubmember(mike) or skier(mike) or climber(mike)`. The line heuristic is applied, which means that R2 and the non-Horn clause are resolved together, and R3 becomes the logical consequent of their resolution. After this, resolution is resumed normally. The success of the proof is shown by the empty clause, indicating the unsatisfiability of the refutation of the goal concept.

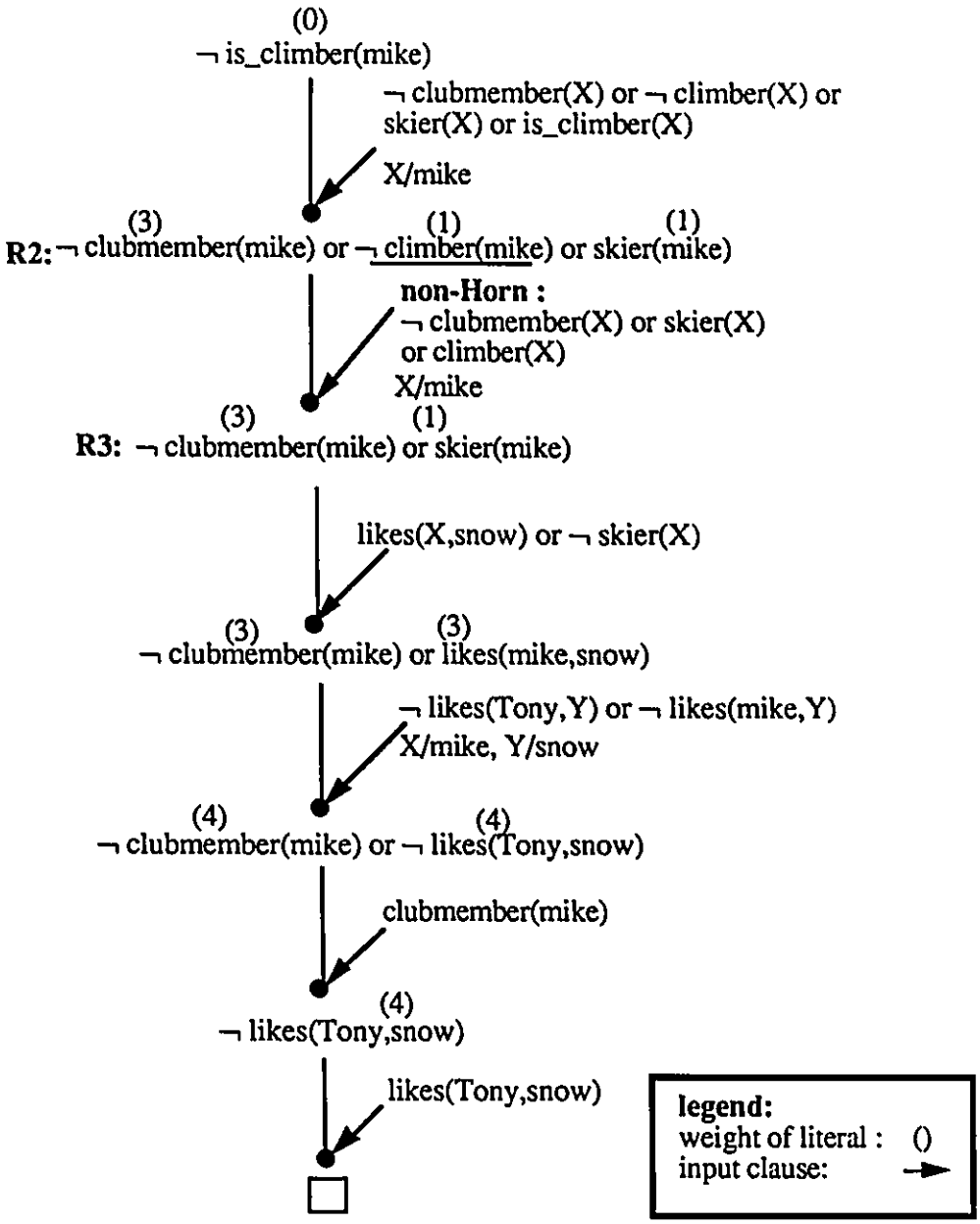

Fig. 4.2 Alpine club example using the line heuristic

4.2 The diamond heuristic

The diamond heuristic is a complement of the line heuristic -- the means by which a disjunction is introduced in the proof to resolve the non-Horn clause. An effect of applying the diamond heuristic is to introduce a disjunction into the concept definition, which is done to resolve simultaneously multiple resolvents with one expression, the non-Horn

clause, to produce a solution. A non-Horn candidate used as the input clause collects the appropriate resolvents (end nodes of branches). Each branch is developed separately until the non-Horn candidate is needed again for resolution. The end nodes of branches (needing the same non-Horn clause as an input clause) are joined each time there are at least two branches. Resolution is then performed on the resolvent of the conjunction of the branches and the non-Horn clause. If this succeeds, and the empty clause $\square$ is obtained, then the proof is accomplished, otherwise an alternative is sought by backtracking. When an alternative branch is developed, its resolution is resumed normally, unless the non-Horn candidate is encountered. It might therefore happen that one branch fails using a non-Horn clause *NH*, but the next branch to be developed does not need *NH* , and succeeds using Horn clauses or other non-Horn candidates.

Consider the simple case of a non-Horn clause with only two positive disjunctive elements in the consequent of the clause: $NH : \neg A \vee \underline{P \vee Q}$ is a candidate to resolve $\neg Q$ (*NH* has already been encountered once when trying to resolve $\neg P$).

Resolvents of the proof containing $\neg P$ and $\neg Q$ are merged together to continue the proof. The two clauses that contain $\neg P$ and $\neg Q$ are conjoined -- a transformation in disjunctive normal form. In this formula, $\neg(P \vee Q)$ can now be resolved with the non-Horn clause *NH*. The link created between the clause that contains $\neg P$ and the clause that contains $\neg Q$ creates a resolution graph with a diamond shape, hence the name of the heuristic .

Here we apply these two heuristic to the example of `disease`. When the line heuristic fails, then backtracking occurs and the diamond heuristic is used, where P =`sle`, Q =`mctd`, A =`arthritis & rash & fever` (see figure 4.3).

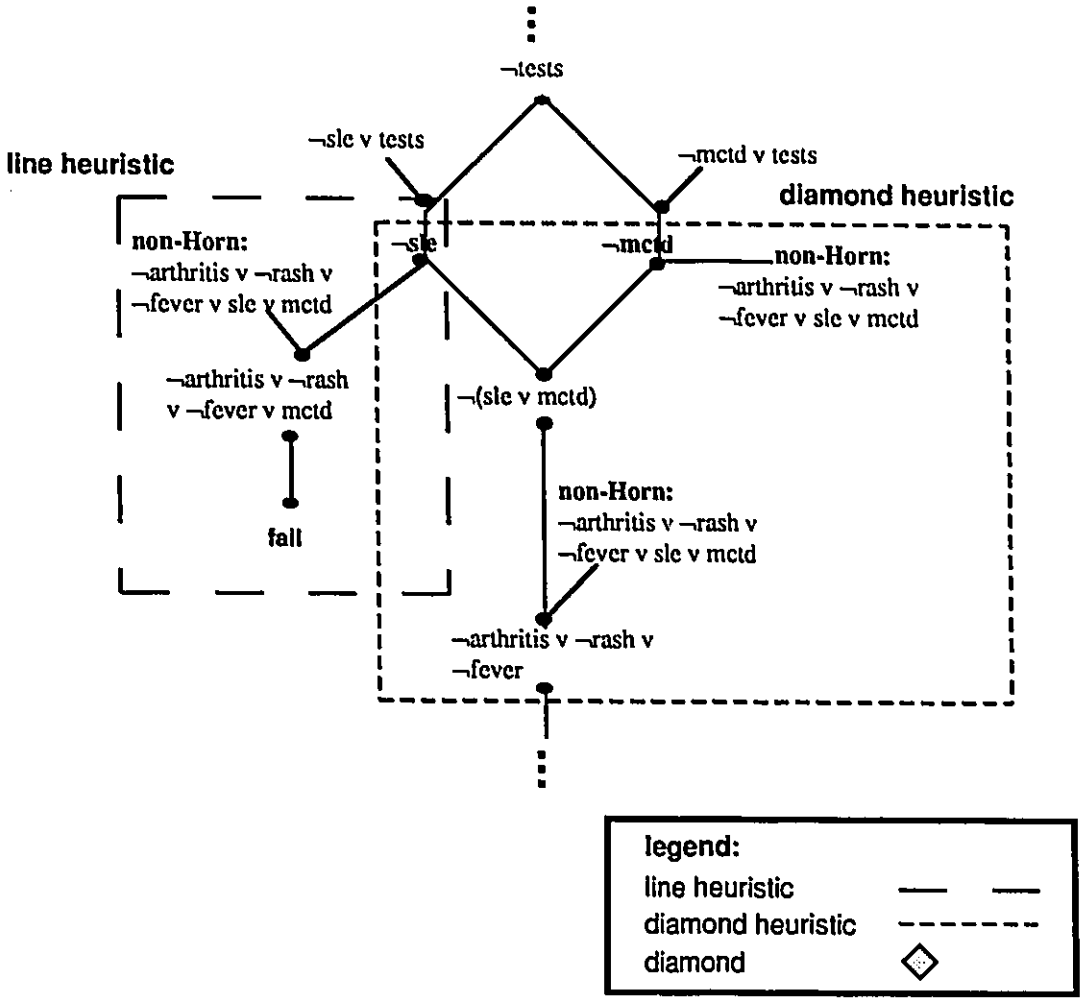

Fig. 4.3 Example of diamond resolution on the `disease` example

The general case involving non-Horn clauses with n+1 literals in the consequent (n>1) will build a diamond with multiple branches conjoined. Consider $NH : \neg A \vee P \vee Q1 \vee Q2 \ldots \vee Qn$. Multiple resolvents are needed for this problem, and the diamond can have as many branches as there are consequents $(n + 1)$.

An example of a diamond with multiple branches is shown in figure 4.4. At resolvent R1, the first alternative is explored, and NH1 is encountered at resolvent R2. Assume that the line heuristic is tried and fails (as indicated by the dashed line): backtracking occurs up to R1, and a second alternative is then explored. Part of the backtracking process of

LODIC is to save any non-Horn clauses encountered and also to save the branch from the non-Horn clause up to the next alternative tried.

In figure 4.4, NH1 is saved (as indicated by the asterisk *) and the branch denoted by α is saved. The next alternative is tried, at resolvent R3: NH1 is needed again, and so the diamond heuristic is applied to the resolvents R2 and R3. The diamond is formed, and resolvent R4 is deduced.

Assume that resolution fails again: backtracking occurs, NH1 is saved, and the branch (β) between R3 and R1 is saved. A new alternative is then explored from R1: at R5, NH1 is needed, and so the diamond heuristic is applied to all the previous resolvents saved using NH1, namely R2, and R3. The diamond is formed from R2, R3, and R5. Resolution resumes from R6.

Assume that resolution fails again, so the backtracking process is performed up to R0. The branch (χ) is then saved. Notice that alternatives do not have to be introduced from the same resolvent.

Finally, the last alternative of this example is explored. At R7, NH1 is needed, and the diamond is formed from R2, R3, R5 and R7. At R8, resolution resumes and succeeds. The only diamond remembered is the last one, as all previous ones were lost during backtracking.

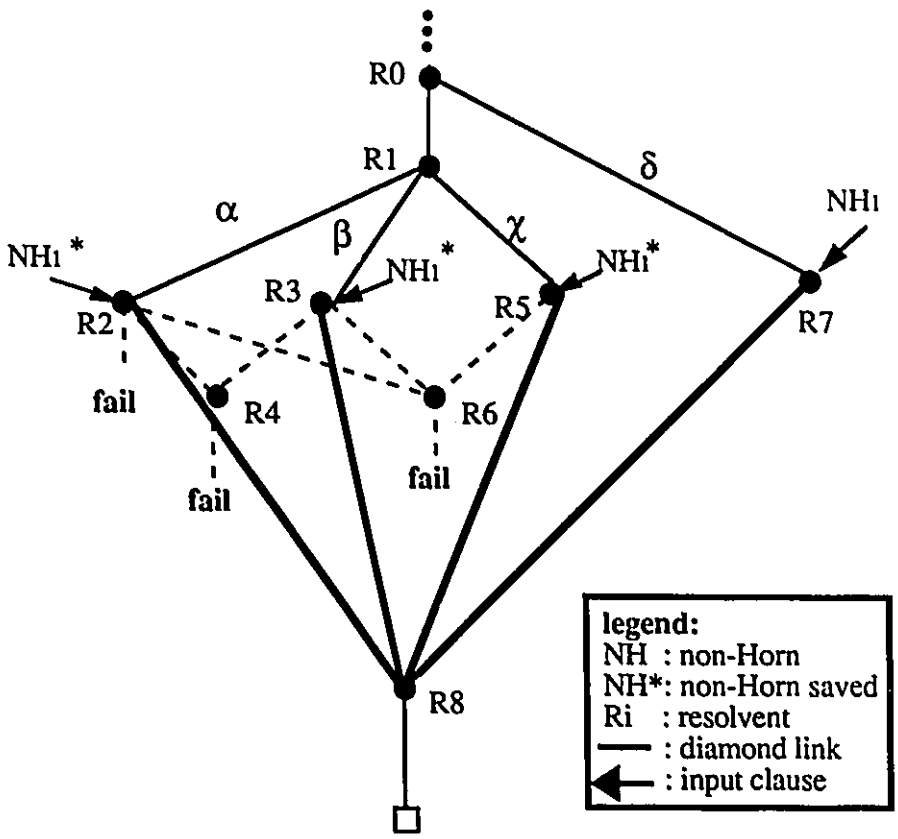

Fig 4.4 Diamond with multiple resolvents

The application of the diamond heuristic can be summarized as follows:

Diamond algorithm

1. Line heuristics failed

2. i) Backtracking occurs to find an alternative branch.

   ii) IF a non-Horn candidate *NH* was used, save it. And save the branch up to the resolvent *Ci* that needs *NH* to resolve away the selected literal.

3. IF a new alternative is found that needs *NH* to resolve the selected literal then:

   i) IF *NH* has occurred before, then form the diamond by making a conjunction of the current resolvent and all the other resolvents *Ri.* saved until now that needed *NH.*

   ii) Form the logical consequent of *NH* and *Ri.*

   OTHERWISE go to step 4.

4. i) Resume resolution.

   ii) IF resolution succeeds then terminate, OTHERWISE go to step 2.

4.2.1 Justification of the diamond heuristic

The following facts give grounds for the diamond heuristic:

1) The non-Horn clause cannot be used on a single resolvent, as is shown by the failure of the line heuristic. We can therefore conclude that the branch, by itself, is not sufficient to prove the refutation. Thus, branches using the same input clause must be joined.

2) The resolvents joined together are axioms that are always TRUE due to their derivation from the proof. Truth is preserved when the resolvents are conjoined because if each part of a conjunction is TRUE then the whole conjunction is TRUE.

3) Logically, the objective is to prove that a disjunction is true. We proceed by refutation and show that the negation of the disjunction is false. For example, disjunction $A \vee B$ is refuted and we are proving that $\neg (A \vee B) \equiv \neg A \wedge \neg B$ is false. This is done by deriving the empty clause from $\neg A \wedge \neg B$.

4.2.2 Diamond heuristic with multiple non-Horn clauses

If the domain theory includes multiple non-Horn clauses, then multiple subsequent and embedded diamonds are allowed in the proof. The following examples all represent possibilities of resolution. In each of these examples shown in figures 4.6 to 4.9, the dashed line represents a failed path, and an asterisk(*) represents a non-Horn clause saved during the backtracking process, as mentioned in step 2 of the diamond algorithm. Greek letters are used to denote branches of proof graphs.

Figure 4.5 is a valid example of the multiple subsequent diamonds. The first branch explored encounters *NH1**. Assume that the line heuristic is applied and fails. Backtracking occurs, and so the second branch developed needs *NH1* again. The diamond is formed and resolution is resumed. At R6, *NH2** is needed. If the line heuristic is applied and fails, then backtracking occurs up to R5, where an alternative is tried. This new branch encounters *NH2* again, and a second diamond is formed with resolvents R6 and R7. Resolution is resumed from the new resolvent, R8, which is the logical consequent of R6, R7 and *NH2*.

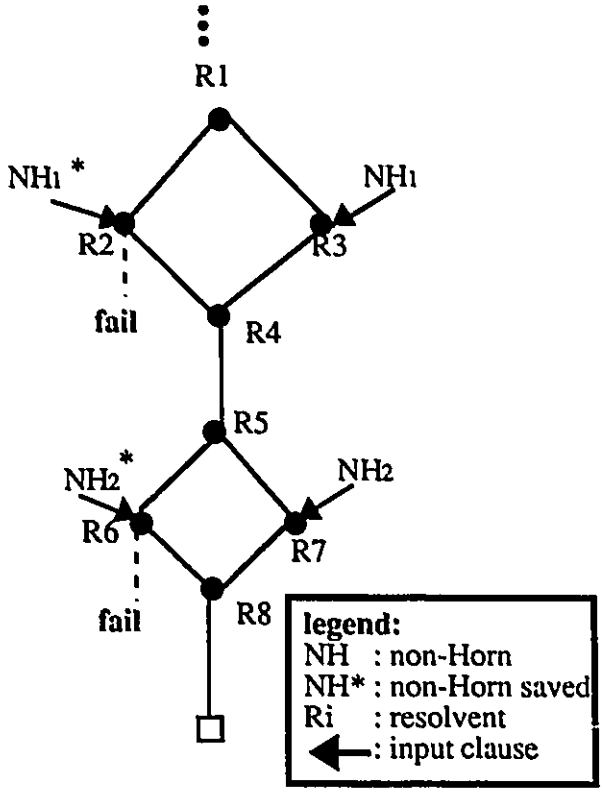

Fig. 4.5 Subsequent diamonds example

Figure 4.6, is also a valid example of multiple non-Horn clauses used in the same proof, but with only one diamond kept. The first branch explored encounters *NH1**, and the line heuristic is tried and fails. Backtracking then occurs up to R1. During backtracking, branch α between R2 and R1 is saved. From R1, a new alternative is explored. *NH2** is needed for the first time. Then the line heuristic is applied. On that same branch, *NH1* is encountered for the second time, but the line heuristic is not tried this time because *NH1* has been encountered before.Therefore, the diamond heuristic is applied: a diamond is formed from the resolvent of R2 and R4. Resolution is resumed, from the new resolvent R5. Assume that this resolution fails: backtracking occurs, and the second branch ($\chi + \beta$) of *NH1* * between R4 and R1 is saved as well as the first branch of *NH2** (χ)

between R3 and R1. The next alternative is once again obtained from R1. On that third branch (δ), *NH2* is encountered for the second time. A diamond is formed with resolvent R3 (this is where *NH2* was first encountered) and R6, producing the new resolvent R7. From this point onward, resolution is resumed and succeeds. In this example, only one diamond is kept in the proof. The first diamond formed by R2 and R4 was lost when backtracking was performed -- this is indicated in the figure by the dashed line. Only the diamond formed by the branch χ and branch δ and resolvent R7 is preserved (solid line).

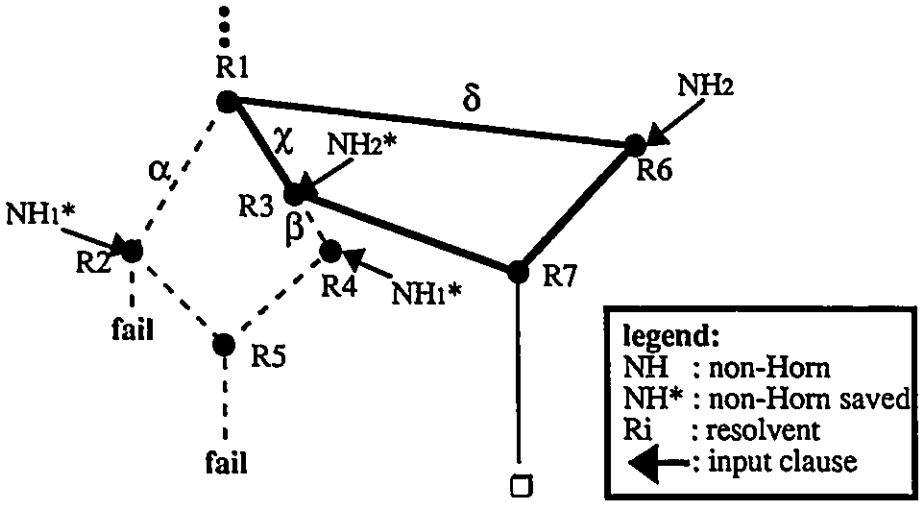

Fig. 4.6 Multiple non-Horn clauses in the same proof.

Now, let us consider an example of an embedded diamond. On branch α of figure 4.7, *NH1** is used to resolve away a literal.The line heuristic is tried. On that same branch, at R3, *NH2** is also needed. The line heuristic is applied for *NH2** and fails, so backtracking occurs, *NH2** is saved, as is branch (α + β). Backtracking continues up to R1, saving *NH1** and branch α. The next alternative to R1 is explored on branch χ. On that branch, *NH2* is needed for the second time. A diamond is formed from R3 and R4. Resolution is resumed, and encounters *NH1* for the second time. LODIC forms a second diamond (dashed diamond), because *NH1* was saved previously. This situation creates embedded diamonds which are not included in one in another.

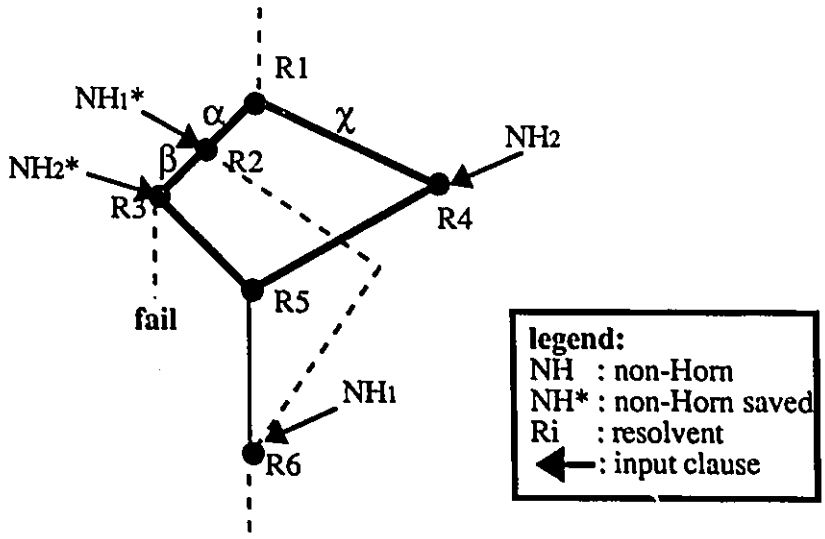

Fig. 4.7 Example of embedded diamonds, the first one formed is the diamond represented by the full line, the second one is the diamond represented by the dashed line.

Another type of embedded diamond allowed is that which is wholly contained in another, larger diamond (see figure 4.8). The shaded diamond represents the diamond formed with the non-Horn clause *NH2*, encompassed by the larger diamond formed with the non-Horn clause *NH1*.

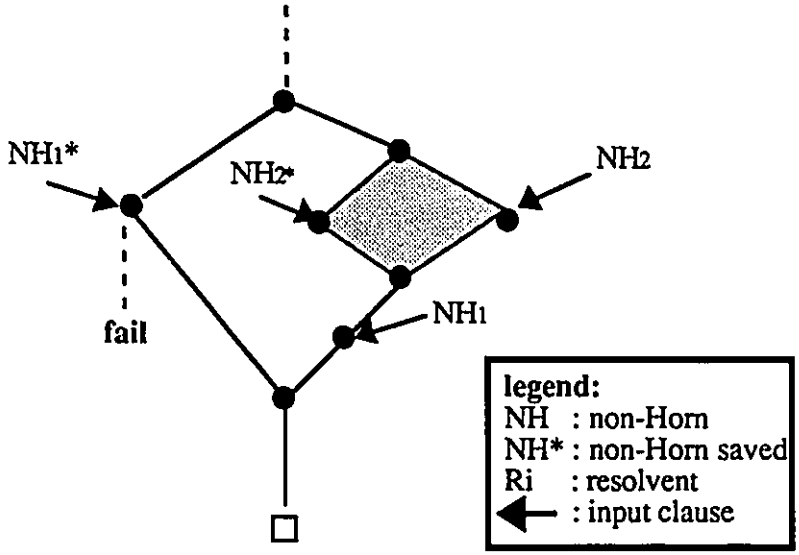

Fig. 4.8 A valid example of embedded diamonds.

The last two examples show embedded diamonds that can be constructed by LODIC. Observe that these examples can be extended to multi-branches diamonds and multi-

embedded diamonds (more than two). Embedded diamonds are important for allowing multiple non-Horn clauses to be used at the same time, and are required if the theorem prover is to be able to solve a large class of problems. As the ratio of non-Horn to Horn clauses increases, the occurrence of embedded diamonds also increases.

There is, though, one exception to the application of the diamond heuristic. It is the case in which a non-Horn clause is used twice on the same branch. Forming the diamond in this specific case is quite likely to introduce a loop in the proof. Using the same non-Horn clause more than once on a branch would probably reintroduce literals that have already being solved.

Deleting non-Horn clauses for recurrence on the same branch does not require special attention. The algorithm of the diamond heuristic handles this case by saving non-Horn clauses only when the line heuristic has failed and backtracking occurs.

For example, consider the partial proof graph of figure 4.9, where *NH* is required to resolve away $\neg a$. First, the line heuristic is applied. If *NH* is needed again, the diamond indicated by α will not be formed because LODIC does not know about the previous *NH*. Moreover, if the α diamond was formed, literal $\neg b$ would be reintroduced in the proof, as may resolvents that lead to $\neg c \vee \neg e \vee d$.

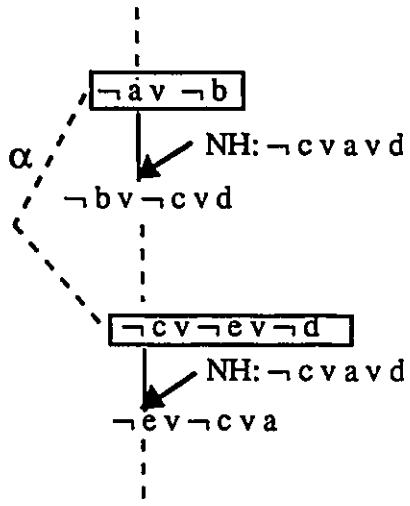

Fig. 4.9 Partial proof graph

The next step of LODIC is to generalize the obtained proof and to apply the operationality criterion to define the operational concept description. This is discussed in next chapter.

Chapter 5

Concept definition

The second step of the EBG method is to obtain from the proof graph a generalization of the training example that is a sufficient concept definition for the goal concept, and that satisfies the operationality criterion [Mitchell et al. 1986]. Operationality is the key property that distinguishes the final concept definition learned in an Explanation-Based system from the initially unusable concept description given as input to the system [Keller 1988].

In classical EBG [Kedar-Cabelli and McCarty 1987; Mitchell et al. 1986], relevant facts from the list of operational literals describe the training example in operational terms. Relevant features of the training example are the facts of the training example that are used in the proof. As the proof uses a top-down strategy, these relevant facts can be found in the leaves of the proof-tree, and consequently the leaves must satisfy the operationality condition. This kind of operationality criterion, relying on the position of the literals in the proof, imposes a strong restriction on the learner.

It is useful to be able to choose literals for their meaning rather than for their position in the proof. Moreover, as DeJong [DeJong et al. 1986] points out, the operationality can sometimes depend on the arguments of a literal as well as the literal itself. In this way, examples will be the means for the learner to point out features that can be of interest, and avoid specifying other information known to be useless.

Many explanation-based systems treat operationality as an independent, static property of descriptions. We argue that operationality is a dynamic, discrete-valued

property of descriptions, which depends on the learner task. LODIC extends the operationality definition to include a more flexible and dynamic criterion. To introduce those ideas, we propose creating levels of operationality in LODIC.

5.1 Operationality levels

The proof graph is partitioned into ***operationality levels***. Each level in the proof graph contains all resolvents of the same weight. The weight of a clause is that of the lightest literal contained in that clause. The proof graph has five levels, progressing from the most general level 0 to the most specific level 4.

- level 0 contains the denial of the goal concept.
- level 1 contains resolvents with at least one literal of weight 1, that is resolvents with literals from the initial goal concept definition.
- level $i = 2, 3$ contains resolvents with at least one literal of weight i.
- level 4 contains resolvents with all literals of weight 4.

Levels 0 and 1 are considered non-operational levels, because they define the goal concept description given as input to the system. Levels 2 and 3 are referred to as potentially operational levels. Level 3 contains clauses with de-instantiated facts (DIFs) and therefore is not present in propositional logic. Finally, level 4 is most specific level, containing all literals expressed in terms of the relevant features of the training example or in operational terms. Figure 5.1 is an example showing propositional logic; note that level 3 is not present in this figure.

The structure of the proof graph does **not** include input clauses: only resolvents are kept. In figure 5.1, input clauses are shown for the sake of clarity.

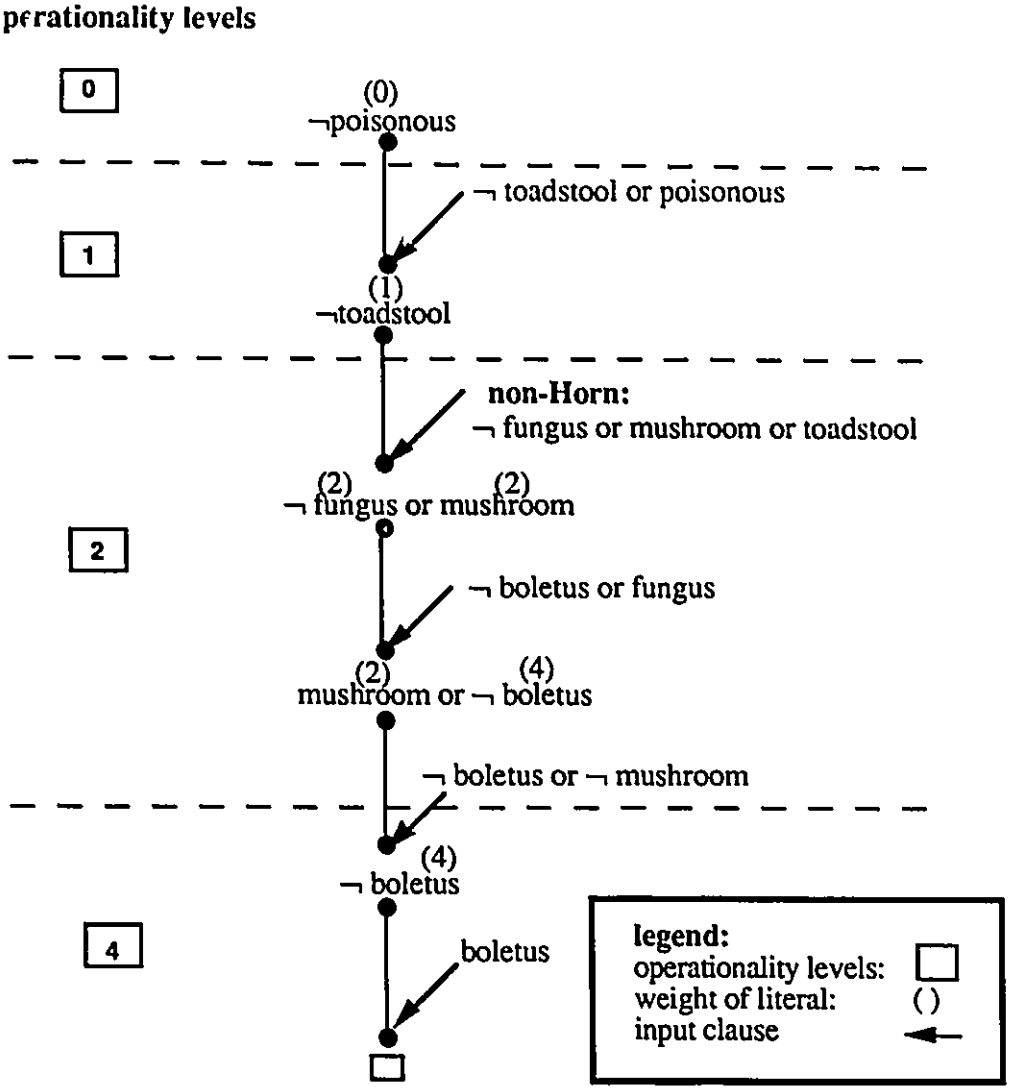

Fig. 5.1 Proof graph of mushroom example with operationality levels.

In FOL, all levels do not necessarily exist in a specific proof graph. For instance, levels 2 or 3 might not be present in an application. Some application may not have any DIF (de-instantiated fact) in their domain theory. This is so in the case of the `block example`. All the facts (`numb(a)`, `numb(b)` and `location(loc)`) are one argument literals, and all arguments instantiate the goal concept,therefore no DIFs are created. Other application might not have any literals or clauses of weight 2.

The operationality levels are used in two ways: first, they prune the search for the collection of the concept definition: second, they extend the operationality criterion to be dynamic and multi-valued.

5.2 Generalization

Before the concept definition is acquired, the explanation structure (proof graph) must be generalized. Naturally, the generalization step, as described below, does not apply to the propositional case.

The technique used for generalizing explanations is inspired by Dejong's work [DeJong and Mooney 1986]. The technique maintains a version of the proof graph along with uninstantiated versions of the general rules, and one version with the instantiation due to the specific example.

Both structures are maintained while the proof is constructed. When an input clause is invoked, a copy of the uninstantiated rule is added to the GENERAL proof graph, and the instantiated version is added to the SPECIFIC proof graph. This process ensures that both structures contain only and all those substitutions which maintain the valid proof.

The SPECIFIC structure determines the operationality level of both structures. When the weight of a clause is assigned to the SPECIFIC structure, the same weight is carried over to the clause of the GENERAL structure.

Figure 5.2 shows our generalization process applied to the `block` example. Only the proof graphs are shown; input clauses were intentionally omitted. The SPECIFIC proof graph contains all the instantiated resolvents of the proof. The GENERAL proof graph contains all the resolvents unified by the global variable A,B,C. Only the SPECIFIC proof graph has weighted literals. The GENERAL one has only operationality levels, because the

weight of literals and the weight of clauses are no longer necessary or meaningful at this stage.

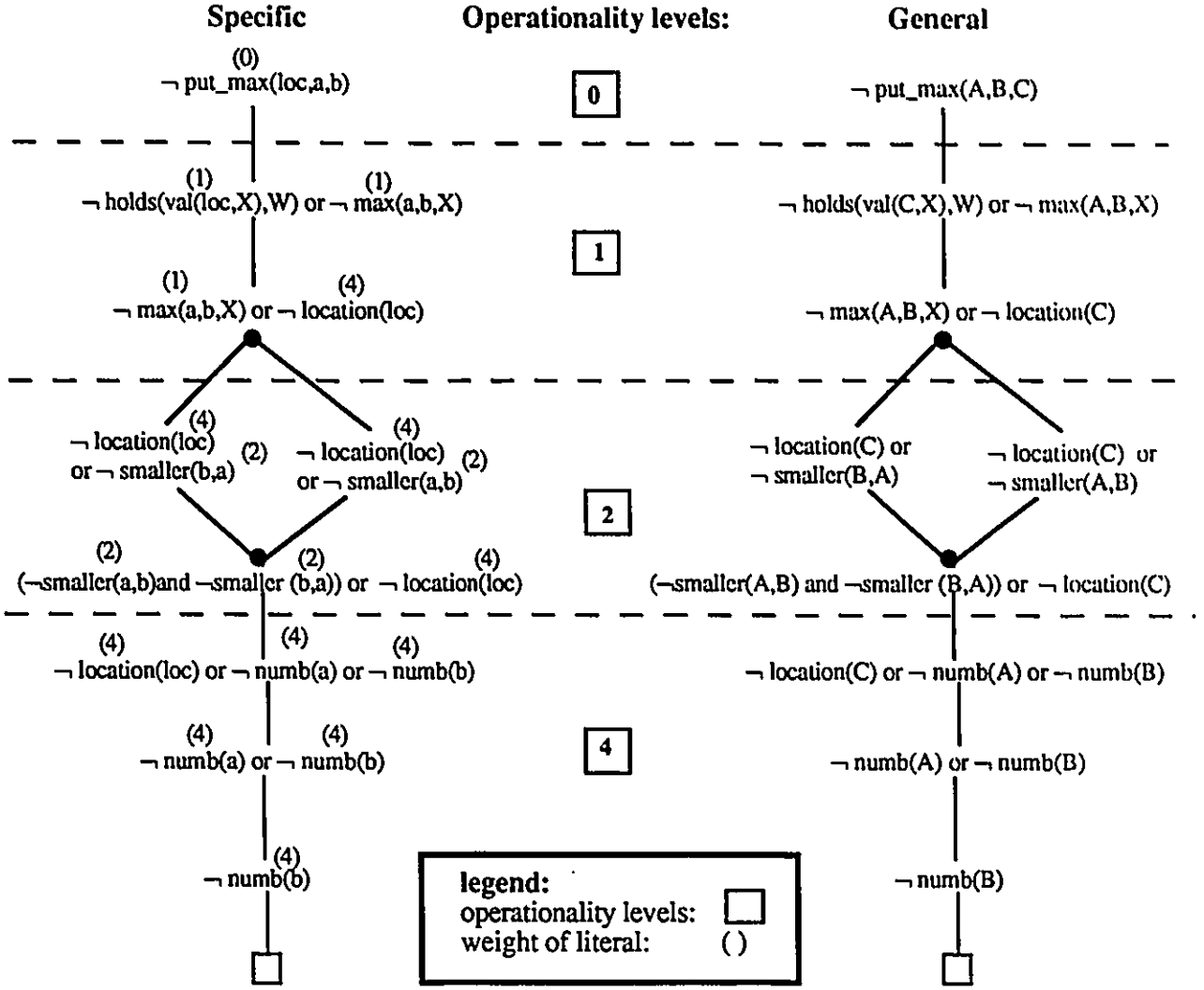

Fig. 5.2 Specific and general proof graphs of the `block` example.

This technique has some advantages over the goal regression used by Mitchell [Mitchell et al. 1986]. First, the generalization process is incorporated in the proof process instead of being separate. Second, the generalized antecedents produced by an explanation are the same for both structures. There is no chance of having an argument used in different places with different relations, generalized to the same variable.

With the general structure now attained it is possible to acquire an operational concept definition.

5.3 Acquisition of concept definition

In this section, we discuss how the concept definition is acquired. The operational concept definition is obtained from the general proof graph of the example. Note that in the proof graph, the concept definition is the negation of the selected clause. From symbolic logic, we know that $\neg B \Rightarrow \neg A$ is equivalent to $A \Rightarrow B$. If $\neg B$ is the negation of the goal concept of the proof graph, and $\neg A$ is any resolvent clause of the proof implied by $\neg B$, then $A \Rightarrow B$ is the negation of the selected resolvent clause that implies the goal concept. Therefore, a conjunction in the concept definition appears as a disjunction in the proof graph and vice versa. Hereafter, a disjunction or conjunction in the concept definition is referred to as cd-disjunction and cd-conjunction respectively.

Different options are available to the user. It is possible to define the operationality criterion as it was defined in previous systems :

> "The concept definition must be expressed in terms of the literals used to describe examples... or selected, easily evaluated predicates from the domain theory." [Mitchell et al. 1986]

The classical operationality criterion, as defined by Mitchell, is achieved naturally by our system by collecting the negation of the last clause of operationality level 3, or the negation of the first clause of level 4 if there is no level 3. The last clause is selected because the literal to be resolved away is deleted by a fact. Level 4 contains only literals that are proved to be true by facts, and that are easily evaluated literals of the domain theory. To make these easily evaluated literals operational, the user must provide them to the system, and allocate to them a weight of 4 by declaring them as facts.

Applied to the `mushroom example` proof (figure 5.1), the classical operationality criterion would produce a concept definition expressed by literals describing examples: `boletus => poisonous` -- all boleti are poisonous. This concept definition, although operational, is not very interesting, but it might be interesting to declare as operational the literal `fungus`.

The `mushroom example` can be transformed to reflect this new operationality criterion. `Fungus` would be declared as a fact, and be assigned a weight of 4. The resulting proof graph would also reflect this change by solving the literal `mushroom` first, rather than `fungus`. The new concept obtained from this transformation is `boletus and fungus => poisonous` -- all boleti and fungus are poisonous.

It is also possible to take a new approach provided by the operationality levels. This offers the possibility of defining the operationality criterion as a level-based notion, rather than a goal-based notion. The advantage of the level-based operationality is that the user is not required to enter explicit inputs: the concept definition can be defined to be operational at a specific level.

The level-based approach offers a variety of ways to define operationality, out of which we have explored two possible avenues. The first is to define operationality statically, as being always a specific clause of the specified level. The user only inputs the preferred level of abstraction; only levels 2 or 3 are operational. LODIC statically chooses the last clause of the specified level. If the level does not exist, the first clause of the following level is the selected clause. The active operational concept definition is simply the negation of that selected clause.

The advantage of this first method is that the user is not concerned with what should be operational, but rather can simply select the level of abstraction desired. Level 2 is more abstract than level 3. Level 3 is the construction of de-instantiated facts (DIF).

In the `block example` (figure 5.2), the user can select level 2 as the operational level. LODIC proposes the last clause of level 2, `smaller(A,B) or smaller(B,A) and location(C) => put_max(C,A,B)` as the operational concept definition. The concept definition is expressed by the means of a disjunction. This means that A is put in location C if B is smaller than A, or B is put in location C if A is smaller than B. In this example we were lucky -- the cd-disjunction appears as the last clause of the selected level. This is not always the case, for the cd-disjunction can be positioned anywhere in levels 2 or 3. Moreover, there might be more that one clause with cd-disjunctions as is the case of multiple embedded diamonds. So how can operationality be defined more flexibly to allow for the acquisition of the cd-disjunction?

The second approach that is provided by the operationality levels offers the possibility to search through operationality levels, allowing different degree of operationality. The idea is to offer a preference criterion approach.

The preference criterion approach can be seen as a two-tiered approach to operationality. The first tier is the definition of the potentially operational levels (levels 2 and 3). The second tier is the application of a different preference criterion within those potentially operational levels to select the best clause for the operational concept definition.

Levels 2 and 3 are chosen because they are the only ones considered potentially operational. Level 0 and 1 are non-operational, containing the definition of the initial goal concept. Level 4 is the elimination level, where all remaining literals are eliminated by an atom that is true, a feature of training examples and operational predicates.

The preference criterion can reflect any type of efficiency or criteria that may be a proper measure of performance for the operationality criterion. Criteria like syntactic simplicity, cost, elegance, and space efficiency are only a few of the potential criteria which may be relevant to performance.

For the purpose of this research, we are interested in describing the concept definition using cd-disjunctions, therefore our preference criterion favours the clause with a cd-disjunction (if there is one). The preference criterion can be implemented by allocating an extra weight to each clause of levels 2 and 3. This weight of the clause would represent the weights allocated to each of the syntactic features of the clause. A cd-disjunction is allocated more weight than a cd-conjunction, and LODIC simply selects the clause with the heaviest weight. If more than one clause has the heaviest weight, than the user chooses from among those clauses, starting from the most general to the most specific.

For example, a cd-disjunction is allocated a weight of 20 and a cd-conjunction is allocated a weight of 1. In the `block example`, the preceding weight allocation produces the operational concept description `smaller(A,B) or smaller(B,A) and location(C) => put_max(C,A,B)` because of its weight of 21 (see figure 5.3).

In the case of the `mushroom` example, there is no cd-disjunction in the proof graph. The user is given the choice of selecting either `fungus and not mushroom => poisonous` or to select `not mushroom and boletus => poisonous` because they each have the heaviest weight of 1.

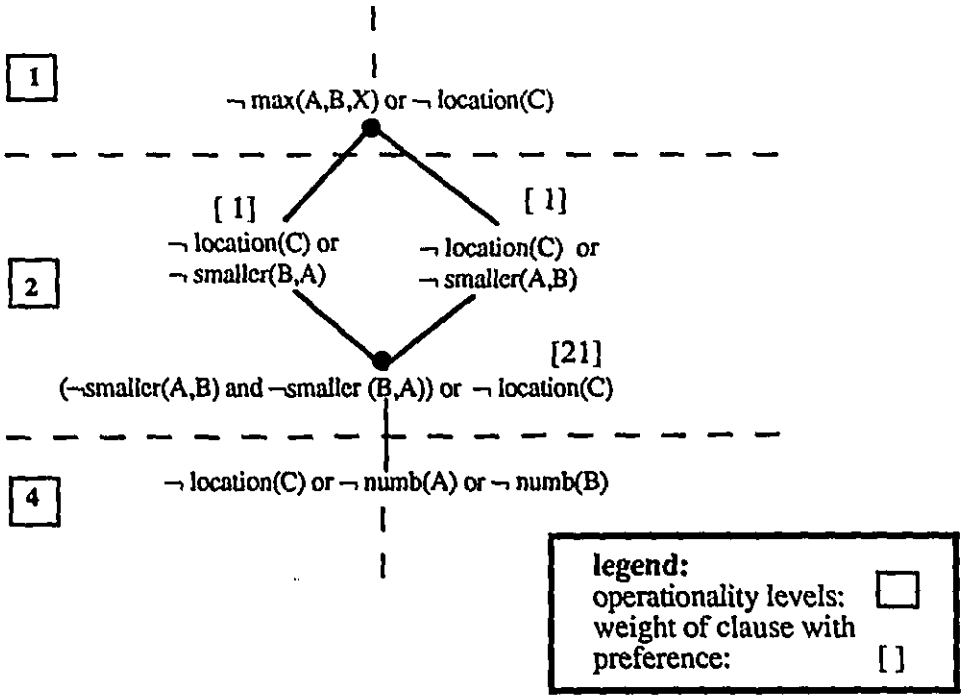

Fig. 5.3 Partial proof graph of the `block` example with weight of clause according to the disjunction preference criterion.

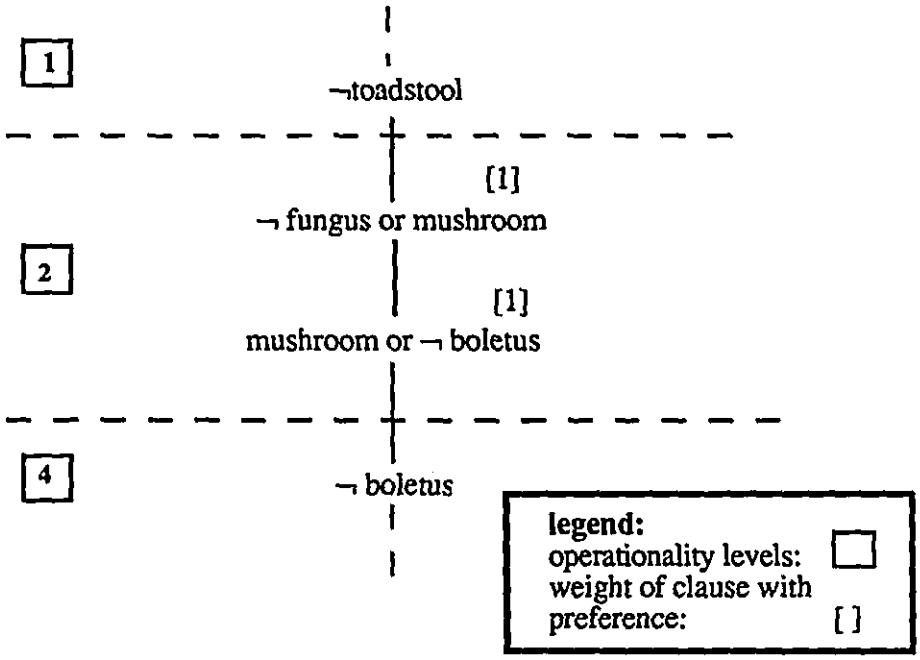

Fig. 5.4 Partial proof graph of the `mushroom` example with weight of clause according to the disjunction preference criterion.

The advantage of the second approach is to provide the user with more flexibility, and the ability to tailor the performance system at hand to the customer and to the user's preferences.

In general, the concept acquisition in LODIC allows a flexible approach to operationality. It also allows for operationality to be defined dynamically over time. The dynamic value of operationality is obtained by changing the weights of literals in the domain theory, without changing the domain theory itself. It seems reasonable to assume that, over time, literals will assume a different meaning from their initial one. The variation of a literal's weight changes the purpose of that literal, and consequently influences the description of the final concept definition.

Chapter 6

Analysis of the system

The method presented here leads to the question of whether EBG-LODIC is preferable to the classical EBG? This thesis clearly favours the LODIC method. Naturally, we claim that this bias is supported by convincing, rational arguments. These arguments are expounded in section 6.1

A second question concerns the completeness and soundness of LODIC-resolution. The performance of the theorem prover is evaluated in the general discussion of sections 6.2, 6.3 and 6.4.

Finally, we examine the limitations of LODIC.

6.1 Comparison with EBG

A natural implementation of EBG [Kedar-Cabelli and McCarty 1987; Mitchell et al. 1986] uses SLD-resolution, in which case EBG is a meta-interpreter that uses SLD-resolution and unification to prove its goal from left to right.

6.1.1 EBG SLD-resolution characteristics

SLD-resolution stands for Selective Linear resolution. Standard Prolog systems employ the computation rule, which always selects the leftmost atom in a goal, and also uses a depth-first search rule. The search rule is the strategy for searching the SLD-trees to find success branches. The depth-first search is combined with an ordering strategy to choose the matching clause. Standard Prolog systems follow the order of clauses in a program as the fixed sequence in which they will be tried, and thus are very simple and

efficient to implement. However, this simplicity is obtained at the cost of completeness of the method.

6.1.2 Parallel comparison of EBG-SLD and EBG-LODIC

EBG SLD-resolution	EBG-LODIC
• no mechanisms are provided by SLD to access only domain theory lemmas.	• Input resolution only allows access to lemmas from the domain theory.
• SLD-resolution is complete. Prolog SLD-resolution is incomplete because it uses depth-first search strategy. Considering efficiency forces the adoption of an unfair search strategy [Llyod].	• Input resolution is not complete because only input lemmas are allowed in the proof, this is a fundamental characteristic of EBG.
• The ordering strategy for the next literal to be resolved upon is to choose the leftmost literal of the clause. No importance is given to the literal itself, only to the position of the literal in the clause.	• The ordering strategy is to choose the lightest literal in the clause. Different literals are given different weights according to the interpretation given to each specific literal.
• All clauses of domain theory are considered for the matching process.	• The choice of the deleting clause is made from the weights of the matching clauses. Only clauses with a weight superior or equal to the weight of the selected literal are considered for the matching process.
• Operationality is used as the proof termination criterion. If this cannot be satisfied, the proof fails.	• The proof develops progressively, following the degree to which literals are operational. It is developed from non-operational to fully provable literals.

• The operationality criterion can be expressed as relevant features of examples.	• The operationality criterion can be expressed in different ways: as relevant features of examples, in different instantiations of the same predicate,in a level-based approach, or as a two-tiered operationality criterion. • Operationality can be dynamic over time by changing weights of literals.
• Non-Horn clauses cannot be incorporated easily into the proof process, while preserving the tree structure, and keeping disjunctions within the structure.	• Non-horn clauses are added to the proof by the means of two heuristics and can be kept as part of the proof graph.
• A tree structure is developed through a depth-first search.	• A graph structure is developed through a breadth-first heuristic search.

6.1.3 Comparison of LODIC-graph structure to SLD-tree structure

[Mitchell et al. 1986] presents an example of EBG, the safe_to_stack example, that will be used here to demonstrate how LODIC-structure can be compared to EBG-SLD [Kedar-Cabelli and McCarty 1987]. The domain theory has rules about the safety of stacking one object on another.

Goal concept : `not (fragile(y)) -> safe_to_stack(x,y)`
`lighter(x,y) -> safe_to_stack(x,y)`

Domain theory:

```
weight(p1,w1) and weight(p2,w2) and <(w1,w2)
                                ->lighter(p1,p2)
compute_weight(p,w)                        -> weight(p,w)
isa(p,Endtable) and density(p,d) and mult(x,d,w)
                                -> compute_weight(p,w)
```

Training example:

```
on(Obj1,Obj2)
isa(Obj1,Box)
isa(Obj2,Endtable)
color(Obj1,Red)
color(Obj2,Blue)
volume(Obj1,1)
density(Obj1,0.1)
```

Finally, the operationality criterion for EBG-SLD states which predicates may appear in rules:

```
operational(volume(p,w))
operational(density(p,d))
operational(on(x,y))
operational(color(p,c))
operational(isa(x,o))
operational(mult(x,y,z))
operational(< (x,y))
```

The operationality criterion for EBG-LODIC can simply specify the collection of the last clause of **level 3** of the proof. There is no need to declare the predicate as `operational`.

Figures 6.1 and 6.2 show the two structures generated by each learner. Comparing these two structures shows that an analogue construction can easily be achieved by the LODIC method. The LODIC proof-graph represents the literals of the SLD-tree developed with a breadth-first search, as indicated by the arrows in figure 6.1.

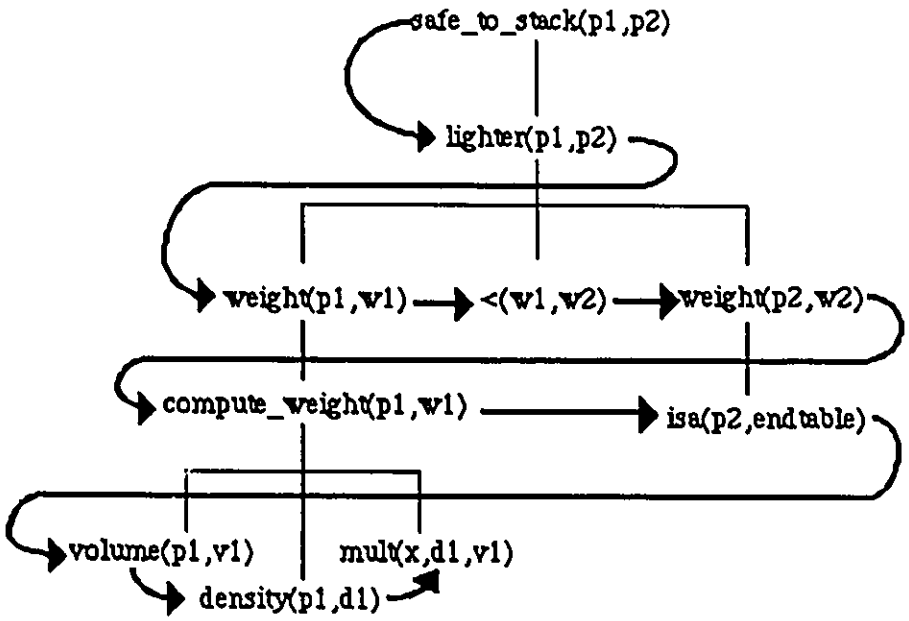

Fig.6.1 The SLD-tree structure of the safe-to-stack example. The arrows represent the traversal of the structure in a breadth-first search.

SLD-EBG differs from LODIC in this example by its representation, by the way it expresses the operationality criterion, and also by the manner in which the operational concept definition is acquired. The LODIC structure provides a clearer view of the final concept description than does the SLD-tree. The concept is defined as a clause in a normal disjunctive form (represented by the box of figure 6.2), rather than as leaves of branches needing to be conjoined.

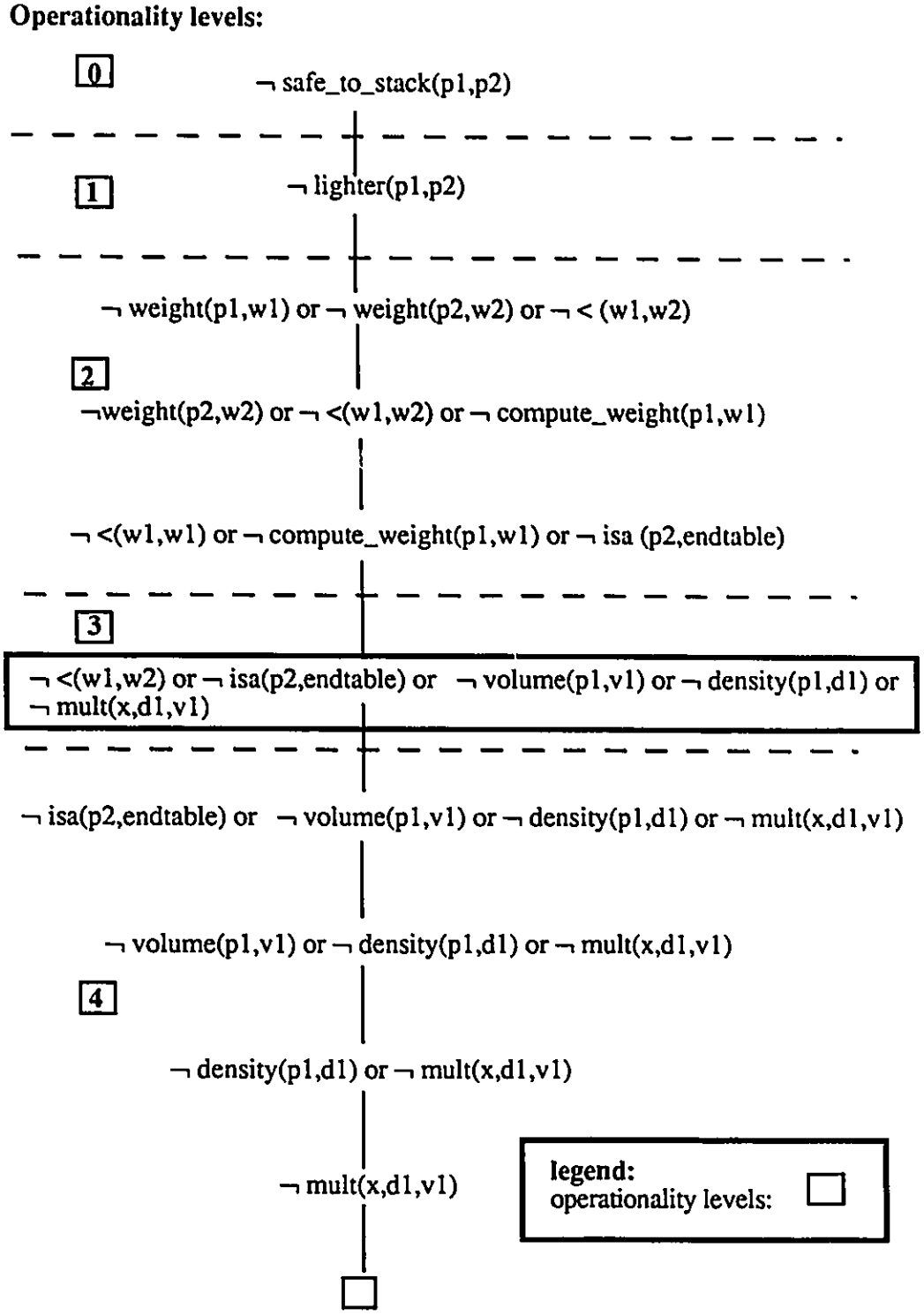

Fig. 6.2 The graph-proof of the safe-to-stack example. The boxed clause represents the negation of the operational concept definition at the level of features relevant to the training example.

6.2 Complexity of the method

The method's performance may be measured by experience with running systems or by mathematical analysis. In the former case, one would compare the relative performance of implementation of different methods on a number of samples. This approach is not feasible, as there is no other EBG system that allows non-Horn clauses.

Complexity analysis, the other possibility of measuring performance, is rather limited as well. In the literature, we find mainly discussions of the idealized *worst-case* analysis, but even those results are sometimes far from reality.

In the average-case and worst-case analysis, the inputs to the method are measured by some attribute: in our case the formulae are measured by length.

In the worst-case analysis, resolution is exponential [Bibel 1987]: 2^{cn} where c is a constant > 0 and n is the total length of all clauses. This holds for a purely deductive system with a complete, rule-like resolution. But our learner is not purely deductive, as it works with tools: a weighting strategy, a set-of-support strategy, a refinement of input resolution, and non-Horn heuristics. All these methods add speed to the search and reduce the size of the exponent associated with it but do not change its exponential character.

The average case treatment of the linear resolution procedure [Bibel 1987] is the number of steps on average needed to obtain a proof. The problem with this treatment is that it needs, as a prerequisite, a probability distribution of the inputs (for example, the length of clauses). A variety of such distributions has been assumed by Bibel for the propositional logic, and the expected time complexity for regular resolution has been derived. The average computational cost is $c * r * n$, where c is some constant, r is the number of distinct atoms, and n is the number of clauses. Unfortunately, little can be said

about the realism of these distributivity assumptions. Further, it is much more complicated for the FOL case.

It therefore seems that all these mathematical results do not say much about real conditions. However, the results are more encouraging than those obtained by the worst-case analysis.

6.2.1 Complexity of the diamond heuristic.

The worst-case computational complexity of the diamond heuristic is evaluated to $O(m^2)$ plus the cost of backtracking. The variable m is the number of consequents in the non-Horn clause that form the diamond. Backtracking is actually responsible for deriving the exponential cost of the resolution. It can, however, be improved for the diamond heuristic (see discussion in chapter 9).

To demonstrate the worst-case analysis of the diamond heuristic, consider the following non-Horn clause : *A1 and A2 ... and Am => C1 or C2 or ...Cm* . The first diamond is formed when two different, negated consequents of the non-Horn clause are encountered, say Ci and Cj where $i \neq j$. Assume that resolution fails; the next branch found with a negated *Ck* ,where $k \neq i \neq j$, forms a three-branch diamond with the two preceding branches. Assume resolution fails again; the worst-case will have a branch for each of the consequents of the non-Horn clause, m branches, and $2 + 3 + 4 \ldots m\text{-}1$ *branched* diamonds will be formed and deleted. This is equivalent to $m(m+1)/2 \rightarrow O(m^2)$. Observe that this result is possible because the order in which the consequents are found is irrelevant.

6.2.2 Comparison with Linear resolution of Horn clause problem

To compare the approach described in our work with linear resolution, we shall investigate how both methods search a specific solution space. The next example illustrates

that LODIC can reduce the search space by as great a ratio as 1/2 over normal linear resolution, even though linear resolution is an efficient method.

Linear resolution was chosen because it generates the same linear structure as LODIC. The algorithm of linear resolution chosen is the one described by [Chang and Lee 1973]. Linear resolution resolves the leftmost atom of the clause, and only stops when no literal of the clause can be resolved upon. This algorithm was augmented by the occur check to prevent resolution from going into an infinite loop. Both methods build new resolvents the same way, appending the remaining literals of the input clause to the end of the remaining literals of the parent resolvent. If a resolvent clause equally weighted literals, LODIC chooses to resolve the leftmost atom with the lightest weight.

The example chosen to make the comparison is a simplification of the `carnivore` domain (see table 6.1).

D :	carnivore (X)	
G : 1)	eat(X,meat) & is(X,animal)	=> carnivore(X)
T : 2)	is(Y,animal) & eat(X,Y)	=> eat(X,meat)
3)	has(X,Z) & is(Z,teeth)	=> eat(X,meat)
4)	mammal(X)	=> animal(X)
5)	fish(X)	=> animal(X)
6)	bird(X)	=> animal(X)
E: 7)	mammal(whale)	
8)	eat(whale,fish_1)	
9)	fish(fish_1)	

Table 6.1 Theory of the `carnivore` example

Figures 6.3 and 6.4 show the complete search path obtained by both methods. The input clause number corresponds to an input clause of domain theory shown in table 6.1. The number of nodes generated by LODIC is nine and by linear resolution it is 22.

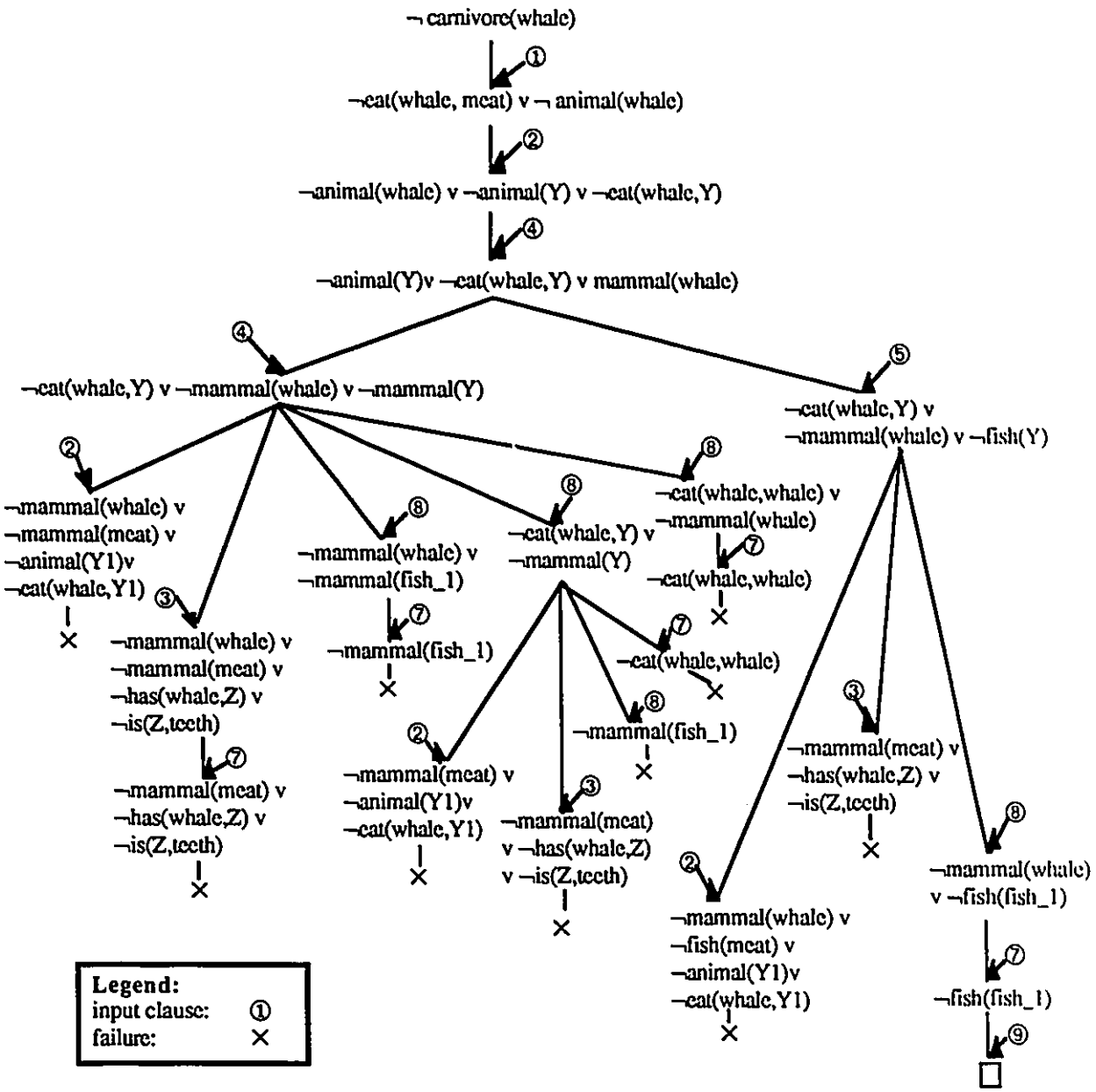

Fig. 6.3 Complete search path of the `carnivore` example with linear resolution. The search traverses 22 nodes before reaching the empty clause.

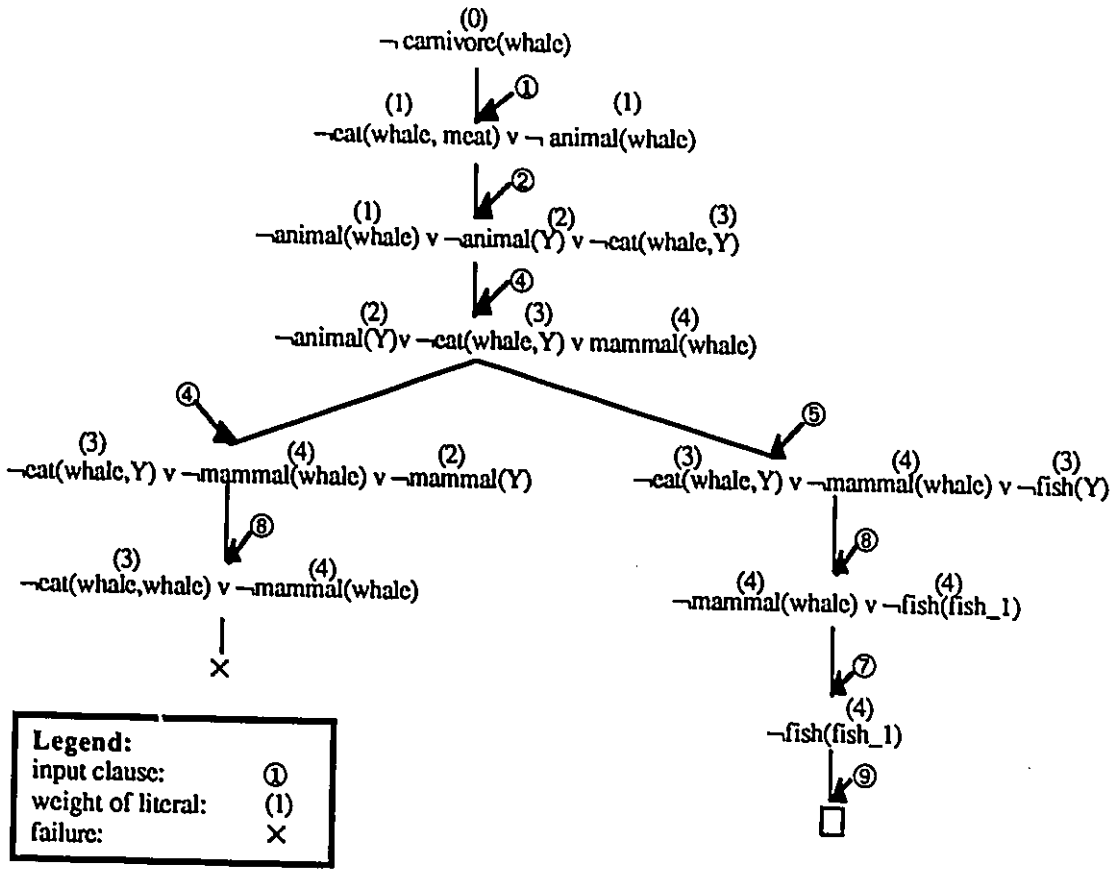

Fig. 6.4 Complete search paths of the `carnivore` example with LODIC. LODIC only explores nine nodes before reaching the empty clause.

What causes the pruning of the search in LODIC? Two types of pruning occur in LODIC. The first type stops the proof for a branch where the lightest literal of its resolvents cannot be resolved upon. For example, in the resolvent `¬eat(whale,whale) v ¬mammal(whale)` on the left branch of the search, the lightest literal `eat(whale,whale)` cannot be resolved upon, and so the proof is stopped. The process backtracks to the previous resolvent, where it finds that the lightest literal `mammal(Y)`, also cannot be resolved upon. The proof fails and the process backtracks again to the previous resolvent. This pruning strategy results in the exploration of only two nodes rather than 13 nodes, as was the case with linear resolution.

The requirement that the weight of the matching input clauses be superior or equal to the weight of the literal to be resolved upon is the second pruning method. This is illustrated by the right branch of the search path. In the resolvent ¬eat(whale,Y) v ¬mammal(whale) v ¬fish(Y), the lightest literal is ¬eat(whale,Y) with a weight of 3. Linear resolution chooses the same literal because it is leftmost in the resolvent. LODIC considers only one input clause (8) as a valid matching clause. The two other input clauses (2 and 3) are invalid because their weight clause is 2. In our example, the gain is two fewer nodes to explore.

LODIC improves on linear resolution by the order in which it selects literals. Linear resolution selects the leftmost literal of the resolvent, whereas LODIC gives a special interpretation to each literal (reflected by the weight of a literal) and resolves from the most general to the most specific literal.

6.3 Completeness of the method

Resolution completeness is understood as the ability to use resolution to refute any inconsistent set of clauses. Why have we chosen input resolution augmented by a graph resolution ? Is this method complete?

SLD resolution allows only Horn clauses. Linear resolution, which is a refinement of general resolution, was another candidate method. Both are known to be complete [Chang and Lee 1973]. Moreover, linear resolution's linear structure makes the generalization step simple. However, using a previous resolvent as a new resolving clause has a side effect on the proof. When disjunctive rules (non-Horn clauses) occur in the proof, disjuncts are often separated, while applying the rule in practice requires that they be kept together. For instance, we do not want to separate the disjuncts in sle v mctd, because in practice we

may not know which of the two diseases we are dealing with , even though we can be certain that it is one or the other.

As we want to show clearly when the proof is using a non-deterministic rule, we have chosen a refinement of linear resolution: input resolution. Although this method is not complete, it is known to be sufficiently efficient and powerful to prove a large class of theorems. The class of problems is also expanded by the diamond heuristic, which adds to the completeness of input resolution by conjoining unsolved branches if they required the same non-Horn clause for their resolution. In addition, the use of non-Horn clauses in the input clauses extends input resolution to that class of problems in FOL that cannot be expressed by Horn clauses.

Another element of LODIC increasing the completeness of our method is its breadth-first search. Resolution based on a breadth-first search approach has been proven to be complete [Gallier 1986], unlike the depth-first search approach (PROLOG). In addition to proving completeness, [Mooney 1989] conducted an experiment with systems that use learned rules to improve their learning performance. The breadth-first search systems improved their performance as the search space expands. A comparison of systems shows that the breadth-first search can solve more problems in less search time than depth-first search systems and non-learning systems. However, without learning, the run-time of a breadth-first search system is slower than a depth-first search system. LODIC improves breadth-first search run-time by ordering clauses and pruning branches of the proof.

In [Sickel 1976], Sickel characterizes the kind of problems that are 'lost' because of the incompleteness of input resolution. The method cannot handle sets of rules that must be factored, for example. Such cases occur with Horn resolving clauses as well as non-Horn resolving clauses. Factorization generates instances of clauses so that duplicate atoms may be deleted. Factorization introduces new resolvents or new input clauses in the proof

by transforming existing ones. However, as mentioned earlier, an undesired side effect for EBG is the introduction of new input clauses. Hence, we do not want to extend our method to factorization and we have not identified realistic examples of EBL violating this restriction.

6.4 Soundness of the method

An inference rule is sound if every set of clauses, which has a refutation constructed in accordance with this inference rule, is inconsistent. Input resolution [Chang and Lee 1973] has been proven to be a sound rule of resolution, so our method uses it along with two heuristics to handle non-Horn clauses. The line heuristic is included in the general resolution principle.

The diamond heuristic can be proven to be sound. Recalling the argument of section 4.2.1 in the fourth chapter, we can prove that the diamond heuristic is correct. The general example shown in figure 6.5 establishes a proof by contradiction. Suppose there is an unsound inference step. Let us call the first such step obtained during the proof *S*. Since resolution is sound, then the unsound inference must be an application of the diamond step. The antecedents of this step are logical consequences of the axioms, so for the diamond heuristic to yield an unsound result the consequent of the diamond step (*R1 and R2 and R3 ... and R4*) must not be a logical consequent which is a contradiction with 4.2.1.3.

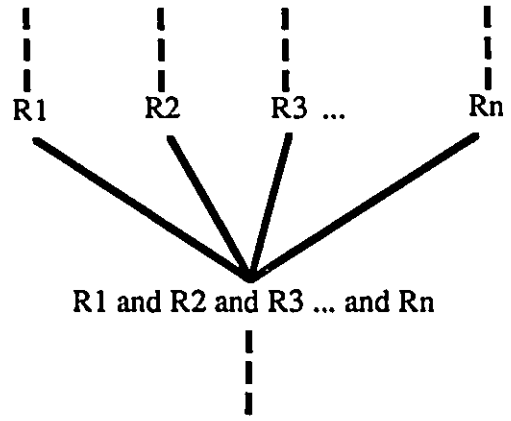

Fig. 6.5 Partial proof graph illustrating the diamond heuristic

6.5 Limitations

Sections 6.2, 6.3 and 6.4 show that LODIC is sound and sufficiently complete to prove a large class of theorems. This is satisfactory from a theoretical point of view, but in practice there are some limitations to LODIC.

• Multiple embedded diamonds could create looping conditions, but loops are detected by LODIC, however this will slow down the learning process .

• LODIC performs better if the ratio of Horn to non-Horn clauses is high.

Chapter 7

Related work

The research presented in this thesis explores two distinct areas of Computer Science and combines them in EBG. The first area of interest is theorem proving, and the second is operationality, which is related specifically to Explanation-Based Learning.

It is difficult to find work that concerns both these areas. The next section explores theorem proving methods that allow non-Horn clauses. The second section refers to work that examines the operationality problem. And finally, the third section discusses a paper on the extension of the EBG framework and operationality, although this enhancement does not allow the use of non-Horn clauses.

7.1 Related work in the theorem proving area

Formal logic provides a uniform framework within which all aspects of EBG may be defined and carried out. The EBG method can be seen as a theorem proving mechanism. In our research, we want the inference engine of EBG to allow the use of non-Horn clauses as well as Horn clauses. No one method adequately solves the problem of non-Horn clauses, but several alternative methods exhaust all cases.

In this section, we will compare our work with two non-resolution methods that build connection graphs using first-order logic.

7.1.1 Connection Graph Proof Procedure

[Kowalski 1979] tries to avoid the redundancy inherent in applying the resolution rule. Redundancy can be avoided only at the expense of flexibility, by restricting

resolution. He proposes the connection graph proof procedure to avoid redundancy without loosing flexibility.

To demonstrate the inconsistency of a set of clauses by the connection graph proof procedure, one generates and solves the initial connection graph. The initial connection graph consists of all clauses within a set of clauses. Each pair of matching atoms on opposite sides of arrows in different clauses are linked, and pseudo-links connect atoms on opposite sides of the arrow in the same clause, if the atoms match in different copies of the clause.

A connection graph is solved if it contains the empty clause □.

To solve a connection graph which does not contain the empty clause,

1) delete a link whose resolvent is a tautology, and solve the resulting connection graph or

2) delete a clause containing an unlinked atom, together with its associated links, and solve the resulting connection graph

or

3) select a link which is not a pseudo-link, delete it, add the resolvent together with its new links to the graph, and solve the resulting connection graph.

The alternative of solving the connection graph should be tried one at a time in the order in which they are written. There is no backtracking involved in this procedure.

Figure 7.1 illustrates the initial graph of the `block` example. Link "a" is a pseudo-link, connecting `holds(X,W)` on one side of the arrow to `holds(X,assign(U,V,W))` on the other side of the arrow of the same clause. The literal in bold, `diff(X,val(U,Y))`, is an unlinked atom. There is no tautology in this

graph, therefore the first step is to delete the clause with the unlinked atom along with all of its associated links. The result of this procedure is shown in figure 7.2.

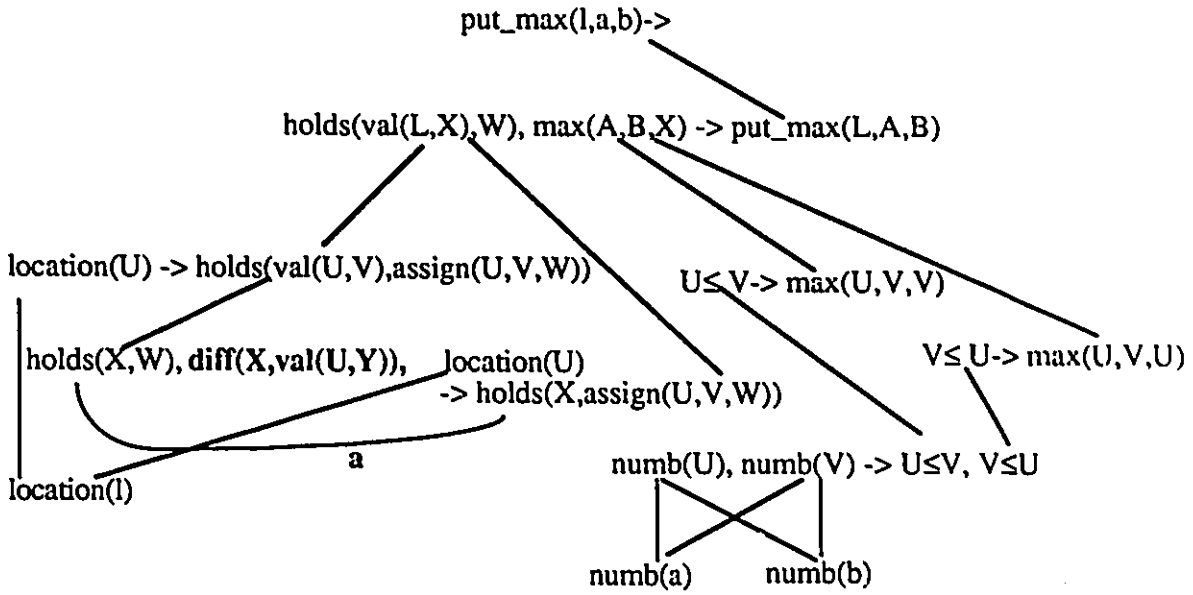

Fig. 7.1 Initial graph of the `block` example. Literal `diff(X,val(U,Y))` is an unlinked atom and therefore, the corresponding clause is deleted from the connection graph.

The resulting connection graph (figure 7.2) does not satisfy options 1 and 2, and so a link is selected. Any link can be selected from the graph. Selecting different links generates different search spaces, some of which may be easier to search than others. We select the links of the literals `put_max` and `location`, which are indicated by bold lines.

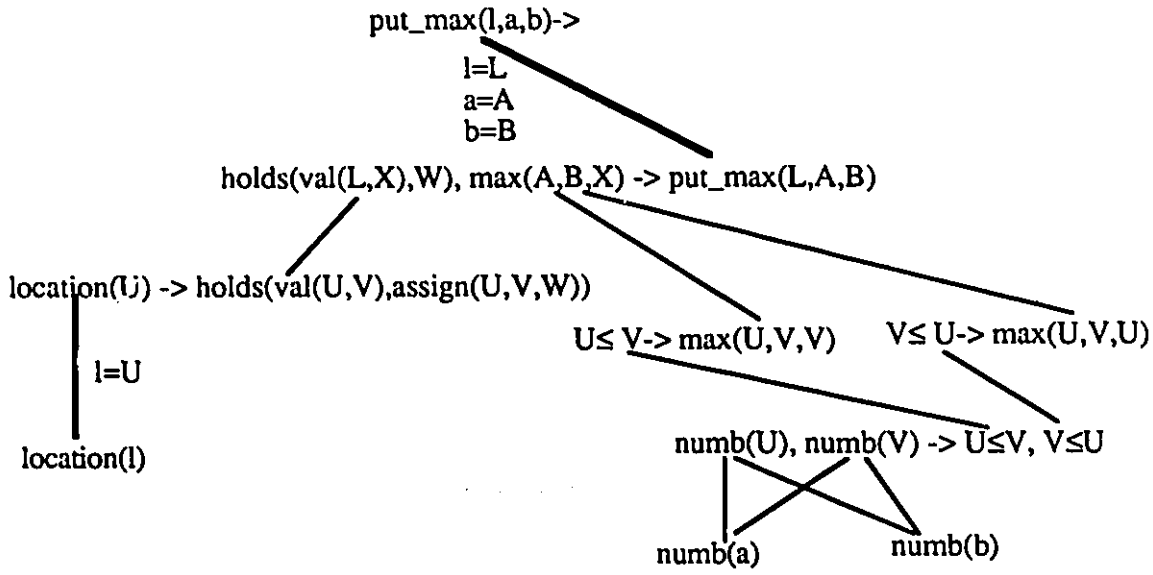

Fig. 7.2 Second connection graph of the `block` example with two selected links to delete: literal `location` and `put_max`.

Figure 7.3 shows the resulting connection graph after the selected links are deleted. Once again, the only option available is to select a link.

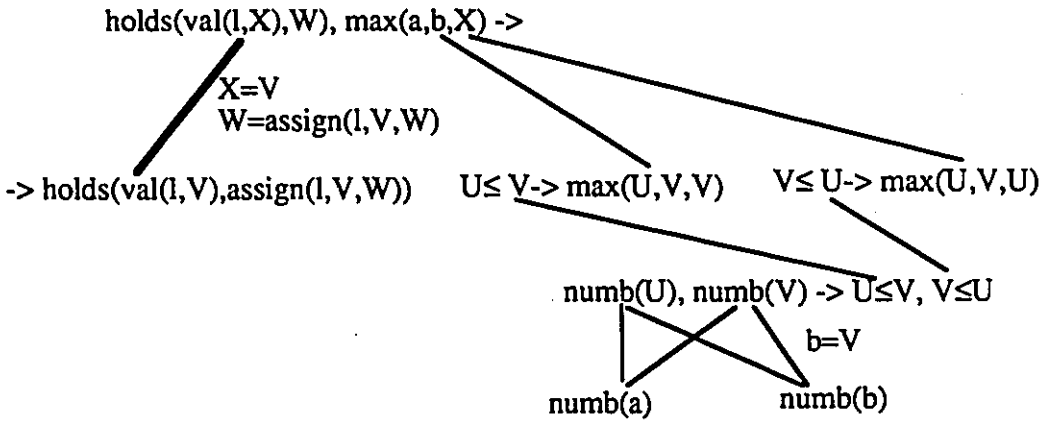

Fig.7.3 Third connection graph with the selection of the link of literal `holds`.

The next figures (7.4, 7.5) show the resulting connection graphs after links are selected. The first connection graph of figure 7.4 provides us with one answer to X: X=V=b. Next, we continue to apply the proof procedure until the empty clause is found.

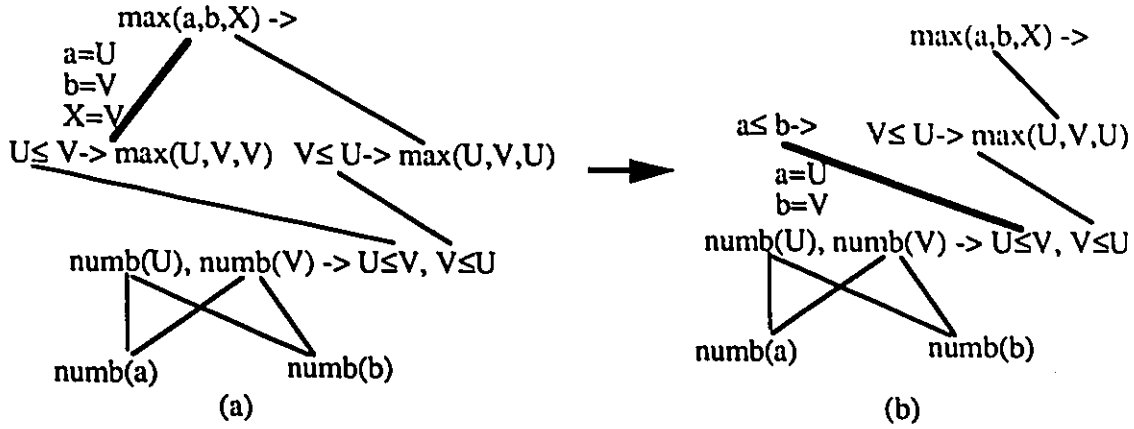

Fig. 7.4 **(a)** Resulting connection graph after the deletion of literal `holds` and the selection of the link of literal `max`.
(b) Resulting connection graph after deletion of one link of literal `max` and the selection of literal ≤.

The last connection graph, shown in figure 7.5, shows that both resolvents associated with the remaining link are the empty clause, and that both parents are deleted. Notice that the proof gives a disjunctive answer to X, X = a or X = b: Max(a,b,X) can either be `a or b`.

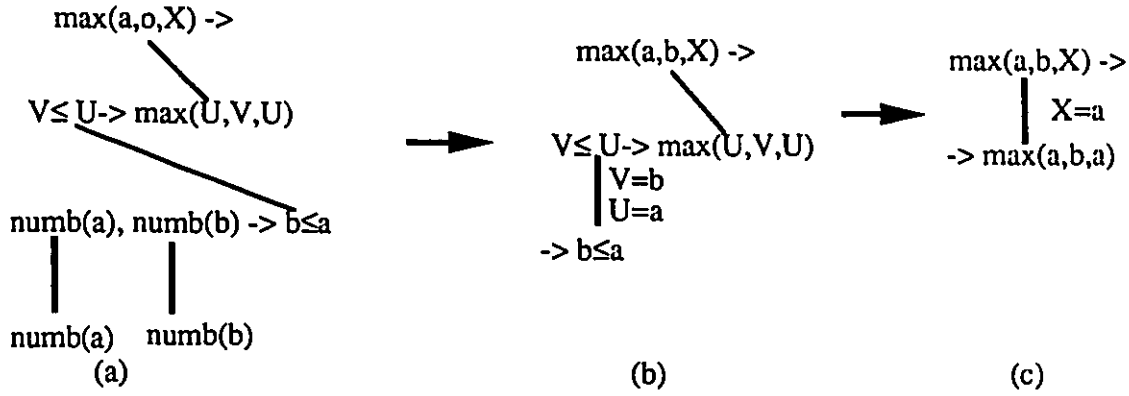

Fig. 7.5 **(a)** Selection of literal `numb(a)` and `numb(b)`.
(b) Resulting connection graph after the deletion of literals `numb` and selection of literal ≤.
(c) Deletion of the last literal, the result is the empty clause.

The sequence of successive connection graphs generated by the proof procedure constitutes a proof on grounds of inconsistency (as indicated by the empty clause) and also a search for the proof.

7.1.1.1 Discussion

Our approach is similar to that taken by Kowalski . We both allow full first-order logic, that is, Horn and non-Horn clauses. Kowalski's graph proof procedure, similarly, can produce disjunctive results when there exist disjunctions in the set of clauses. It can also be restricted to input clauses, although his method provides mechanisms to allow a more complete theorem proving method. And finally, Kowalski's method also has a technique for pruning the search space.

Two disjunctive results are obtained by connection graphs. The first is obtained as new resolvents are constituted. When a link is deleted, the remaining literals of the two subject clauses are merged to form a new resolvent. It is possible for the new resolvent to have more than one consequent on the left side of the arrow, forming therefore a disjunction.

The second form of disjunction is a variable that can be instantiated variously, as is shown in the `block` example. A solution exists, but it is expressed in a disjunctive format. This introduces non-determinism or uncertainty, that correlates to our approach, although only non-determinism is expressed in our learner.

Kowalski's technique for pruning the search space deletes clauses that contain unlinked atoms, and can be compared to the pruning accomplished by the weighting strategy. An unlinked atom is not liable to resolution, and so the clause containing it cannot contribute to the proof: this is the justification for deleting the clause. In our resolution

method, the branches of the proof are pruned as soon as the lightest literal cannot be resolved upon, no matter what literals remain.

The differing goals in Kowalski's work and our own explain the inadequacy of the graph proof procedure to solving our problem. The purpose of Kowalski's connection graph proof procedure is to provide a general proof procedure for automated theorem proving,whereas we are interested in providing a theorem-prover accepting non-Horn clauses conducive to EBG. Moreover, to build new concept definitions we must obtain a usable trace of the proof. The proof is suitable if we can isolate easily the sufficient conditions of the goal concept.

The lack of a definite, simple proof structure, *per se*, is the major obstacle to using Kowalski's connection graph proof procedure to serve the purpose of our work. The connection graph does not keep track of resolvents created during the proof, as we required of EBG.

The connection graph proof procedure is a flexible proof strategy. We can remove links anywhere in the graph connection, as long as we respect the order in which operations (such as deleting a tautology, deleting an unlinked atom, etc.) are carried out. This flexibility is not needed for our purposes, as we require to start the proof from the goal concept and to have some way of collecting the final concept definition from there.

7.1.2 Clause Interconnectivity Graphs

[Sickel 1976] gives a new, non-resolution technique to prove theorems automatically. From the set of Skolemized conjunctive normal form (CNF) input clauses, she constructs a graph that represents the clauses and their interconnections based on unifiable complements of literals (a literal and its negation).

A clause interconnectivity graph (CIG) is represented by a quadruple (*Nodes, Edges, Subst, Clause)* where:

- *Nodes* is a set of graph nodes, one for each literal of a clause.
- *Edges* is a symmetric(bidirectional) relation between nodes if and only if the nodes are unifiable literal of opposite signs. *Edges* are labeled by *Subst*.
- *Subst* is the substitutions such that Subst is the most general unifier of the literals connected by the edge.
- *Clause* allows determination of the clause membership of literals.

Figure 7.6 illustrates an annoted CIG of the `alpine club` example. Note that a CIG does not represent directly literals and terms. Rather, the fundamental aspect is captured in the edges and substitutions that label edges. Here, literals were kept for the sake of clarity.

The method of resolution can be put in parallel with resolution-based, automatic theorem proving. In the context of a CIG, the analogue of resolving two literals is obtained by walking across the path linking the two. The literal that they walk toward is the deleting literal; it deletes the literal they walk from. The other literals in the clause containing the deleting literal are residual (remaining to be resolved).

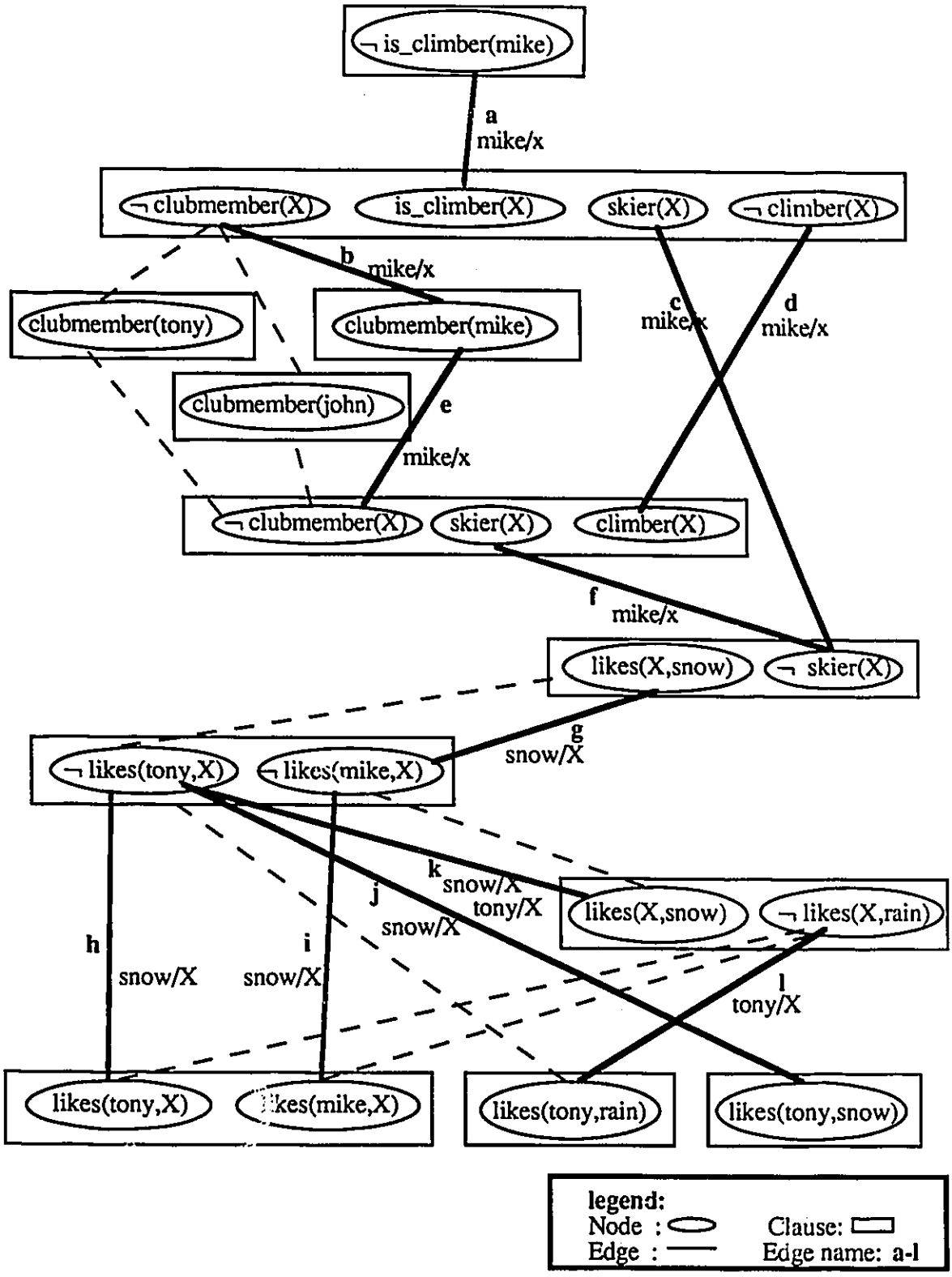

Fig.7.6 CIG plus annotations for the alpine club example.

Theorem proving begins at a specified clause, referred to as the *start clause*. A marker is placed on each of the literals of the start clause. Each of these markers may be moved along any unifiable edge connected to its present position; then, the marker is removed, and new markers are placed on each of the residual literals. Each of these markers may then be moved, and so on. This procedure can be represented as a tree. Figure 7.7 represents the tree structure of the `alpine club` example. Each label edge corresponds to a walk taken along that edge. Each endpoint represents a fact. The loops are unrolled as many times as desired, depending on the wanted level of complexity. The post-order traversal of this tree can be translated into a resolution sequence.

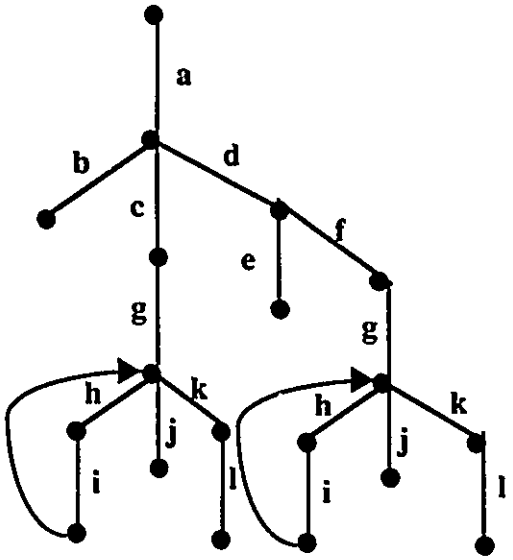

Fig. 7.7 Total solution tree for literal `is_climber(mike)`. The letters correspond to edges of the CIG of figure 7.6.

The tree structure defines a partial solution for a literal, and can be seen as a plan to eliminate that literal. The total solution for a set of clauses refutes that set. A total solution is a set of partial solutions, one for each literal in the start clause.

The CIG is assisted by strategies to ensure that substitutions are consistent, and also by special techniques applied to loops in the graph.

7.1.2.1 Discussion

Sickel's CIG is analogous to our resolution algorithm. CIG coheres to the input resolution strategy, because at each step CIG resolves against a node in a partition of the static CIG. However, CIG is made complete by the introduction of a merge loop, a tautology loop and a factorization technique

CIG allows Horn and non-Horn clauses without distinction. A clause is simply expressed as a collection of literals, whether positive or negative. In LODIC, an important distinction is made between Horn and non-Horn clauses: Non-Horn clauses are processed with special heuristics that preserve disjunctions in the proof.

CIG's solution is expressed as a tree of edges. Traversing the solution tree corresponds to a breadth-first search approach, which is the one taken by our resolution algorithm. This solution tree can be translated into a structure with resolvents, which can be compared to our graph without diamonds. However, the complexity of the resulting structure makes it harder to acquire elegantly the concept definition. This two-step approach to obtain resolvents is also a weakness of CIG for the application of EBG.

CIG is reputed to be more efficient than resolution, but it must be restricted to be applied to EBG. For example, there must be no merge loop, no tautology loop, and the start clause must be restricted to be part of the set of support. These restrictions reduce the completeness and efficiency of CIG. Factorization is another resolution heuristic allowed by CIG. To reiterate the argument of chapter 6, there is no need for such a heuristic in our system.

7.2 Related work in Operationality

The final concept description is distinguished from the initial concept by the key property of EBL systems: operationality. Several researchers [DeJong and Mooney 1986; Hirsh 1988; Keller 1988; Mitchell et al. 1986; Segre 1988] have offered new methods to define the term "operationality" for the purpose of EBL. We will examine two of these methods: Keller's and Dejong's. Keller's method is a general approach to operationality that supports our idea of a level-based approach. Dejong, on the other hand, extends the concept of operationality to produce varying concept definitions depending on the instantiation of predicates. Both systems employ some measure of utility of the learned concept.

7.2.1 Defining Operationality for Explanation-Based Learning

[Keller 1988] revises the operationality definition. Most existing systems attempt to define operationality in terms of "efficient instance recognition", which leads to simplified assumptions, that may eventually deteriorate the effectiveness of the learning system. Keller argues that efficiency alone is insufficient to define operationality. He proposes a revised definition of operationality that permits instance recognition, but also other tasks, such as generating instances.

Keller's definition (see table 7.1) has two requirements: the usability and utility of concept learned. The usability requirement ensures that the description can be used by the system, that is expressed in terms of capabilities possessed by the system. The utility requires that the description must not only be usable but also worth using. The system must improve in accordance with the specified objectives.

Given - a concept description - a performance system using the description to improve preferences. - performance objectives specifying the type and extent of system improvement desired.

Table 7.1 Revised operationality definition of Keller

Keller also discusses how operationality can be assessed in explanation-based systems. Each explanation-based system evaluates a concept by an operationality assessment procedure. To assess operationality, Keller describes three dimensions; variability, granularity, and certainty.

Variability characterizes an operationality assessment's variance over time; its values are static or dynamic.

Granularity characterizes the assessment measure produced; its values are binary or continuous. Binary systems produce operational and non-operational results. Continuous values give degrees of operationality.

Certainty characterizes confidence in the operationality of the concept definition produced; its values are guaranteed and unguaranteed.

Keller has created a system, MetaLEX, that exhibits these three dimensions with the values dynamic, continuous and guaranteed.

7.2.1.1 Discussion

Keller's work, among that of others, gives a great insight into the concept of operationality. It supports some of the ideas that we express intuitively in this research.

Keller recognizes that operationality can be more than just efficient execution time: performance can be expressed by other criteria. This approach supports our idea of offering the possibility for user's preference to be expressed in terms other than those used to express examples.

Keller's notion of granularity can be compared with our partitioning of operationality levels. Both offer different degrees of operationality. His method is more empirical than ours, by measuring the granularity of a learned concept in terms of efficiency (cpu seconds) and by the degree of effectiveness (number of problems solved), whereas our method is more abstract, proposing different measures of abstraction to the user, moving from general concepts to more specific concepts.

Keller's work differs from ours in the context of learning. His system has a performance element, and hence, the final concept definition is evaluated empirically to judge the utility of the rule, before being added to the domain theory. LODIC does not have a performance element, so the learned concept is not added to the domain theory.

7.2.2 Explanation-Based-Learning: An Alternative view

[DeJong and Mooney 1986] presents a detailed critique of EBG, in which Dejong claims EBG suffers from problems of under-generalization. He points out that EBG cannot generalize the predicates appearing in the domain theory, and it cannot generalize the explanation structure itself. He also discusses, among other things, the operationality criterion, arguing that it depends on the arguments of predicates.

> "A training example might be selected in which the propositions are easily evaluated but the very same predicates may be difficult to evaluate when applied to later examples. Nor can we limit the operationality criterion to predicates that are known to be operational for all arguments." [DeJong and Mooney 1986]

Dejong proposes his own formalism as an alternative to EBG. According to Dejong, many of the above mentioned problems can be solved by organizing the system's knowledge base in terms of schemata. This method makes it possible to learn by observation, allows for the generalization of the explanation structure, and it allows for altering the structure of the explanation.

Dejong clearly distinguishes the two components of the explanation-based learning system: the acquisition element and the performance element. The acquisition element builds an explanation of the training example and then generalizes it to acquire a new concept. The performance element makes use of the new, generalized concept. The generalization phase of the acquisition element is affected by the abilities of the performance element, and the operationality criterion is judged on the abilities of a particular performance element.

Thus, the operationality criterion is exactly the set of goals for which the system possesses schemata. It does not need to be input independently in the system. The operationality criterion is also dynamic. As the system learns new schemata, additional goals become operational. New explanations can be constructed with these new schemata.

In the generalization phase, the schema knowledge itself can be used to generalize the example. This form of generalization is possible because the schemata are themselves generalized chunks of knowledge. The explanation is derived by trying to use already operational concepts, which are more general than those used in the particular example, and then determining how these schemata can fit together. The already existing generalities can be exploited in the new concept to prune the too specific parts of the explanation. As a result, the operational description learned is not stated in terms of low level features present in the instances, but rather in terms of the generalized schemata possessed by the system.

7.2.2.1 Discussion

We can only compare Dejong's acquisition element to our learning system, as our's lacks a performance element. Three ideas in Dejong's work are similar to those in LODIC, and all are concerned with operationality.

The first idea is that the operationality criterion does not have to be input independently in the system. Dejong's system considers a goal operational if the system possesses a schemata for it. The operationality of a clause is known by the weight of the clause (weight 2 or 3 in the existing system). LODIC offers this possibility to the user, so there is no need for inputs.

The second idea is to make operationality dynamic, which is done differently in each system, but accomplishes the same end. Dejong's system makes operationality dynamic by learning new schemata that are added to the system and hence adds a new operational goal. LODIC serves the same purpose by allowing the weight of literals to change over time, and consequently the construction of the proof and the interpretation of the concept definition may change.

Finally, Dejong says that operationality depends on the arguments of predicates. This same idea was developed further by Hirsh [Hirsh 1988] by declaring operational a specific instantiation of predicates. LODIC adheres to this idea by assigning to specific instantiations of literals (as chosen by the user) a weight of 4, and thereby making them operational.

As previously mentioned, Dejong's work differs from LODIC in that the learning system has both an acquisition element and a performance element. LODIC has only an acquisition element. Dejong's system is therefore able to judge the utility of the learned rule, and also has extra inputs given to the system. The system is given already known

general schemata and an observed sequence of low-level operators performed by an expert (the latter is optional).

In general, if only the acquisition element is considered, LODIC offers the same various operational possibilities that Dejong's work offers.

7.3 Higher order and Modal Logic

[Dietzen et al. 1989] explores two ways of enriching the representation domain of Horn clause logic, while staying within the logic programming framework: integrated support for higher-order objects, including variables that range over them, and support for modal concepts.

A domain is higher-order if it allows for higher-order objects such as functions to be bound to variables or passed as parameters, and returned from function calls. Higher-order EBG is, then, an explanation-based generalization in which the candidates for variable replacement include higher-order objects. When higher-order values are expressed in first-order representation domains, we often need "new variables", need to check conditions such as "where ... does not occur in ...", or must implement substitution in a way that "renames bound variables if necessary". Instead, higher-order representation domains naturally express and manipulate of higher-order objects. For example, the function f(x) =g(x,2) can be represented as f= λx.g(x,2).

Modal logic provides propositions with multiple levels or modes of truth, such as "may be" and "must be". The $\Box$ (box) operator precedes *necessarily* true sentences or, equivalently, those sentences which are true in all *possible states* or at *all times*. Non-prefixed sentences are only *contingently* true, in the current state or at the current time.

λProlog-EBG [Dietzen and Pfenning 1989] relies on the separation of the domain theory and training instance, as only rules of the former are incorporated into the generalized proof. EBG's distinction between domain theory and the training instance corresponds to distinction made in modal logic between necessary and contingent truth. The ❑ clauses (domain theory) represent knowledge which is already general in that it is necessarily true. The unboxed clauses (training instance) represent particular knowledge.

λProlog re-defines the notion of operationality to be expressed as a predicate over clauses, rather than as a predicate over goals. Because only domain theory clauses are allowed in the generalization, they have made the resulting generalization more specific by converting training instance clauses to domain theory. They can also make it more general by converting domain theory clauses to training instances.

7.3.1 Discussion

Dietzen and Pfenning's work pursues a goal parallel to our work: λProlog seeks to extend the EBG logic framework to higher-order objects. We want to extend the EBG logic framework further to allow for disjunctions in the domain representation of the problem. We both wish to naturally express the problem at hand, without the need to re-posit the problem. Our approaches to operationality also follow the same line of thought, pursuing a general clause approach rather than a predicate over goals approach.

The final concept definition can include a logical implication within the pre-conditions of a derived rule. The generalization obtained by our work allows for disjunctions of the domain theory to be preserved in the operational concept definition.

Differences are found in the method used to obtain results. λProlog uses a meta-interpreter of Prolog to build its higher-order modal logic framework, whereas we want to

use an inference engine different from SLD-resolution (Prolog) to extend the EBG framework.

The implementation of λProlog does not allow the full exploitation of Dietzen and Pfenning's research. Problems with unification that cannot be resolved at the level of the meta-interpreter prevent λProlog from completely working. In addition, the concept definition obtained by λProlog is not what the user would expect, despite their claim that it is still valid.

Chapter 8

High-level description of the program

This chapter provides a high-level description of the implementation of LODIC. LODIC is implemented in PROLOG. Most of the ideas put forth in this research are included in the implementation of LODIC, which has been tested with all the examples presented herein.

The implementation of LODIC was accomplished in two steps: first, with a propositional implementation, and second, in a first-order version. Both versions are given as appendices.

The next section presents a functional view of the program. The second section provides a general algorithm of each of the modules.

8.1 Functional view of the system

The LODIC implementation is shown in the functional graph of figure 8.1. Each module of LODIC has a specific function; both implementations follow the functional diagram.

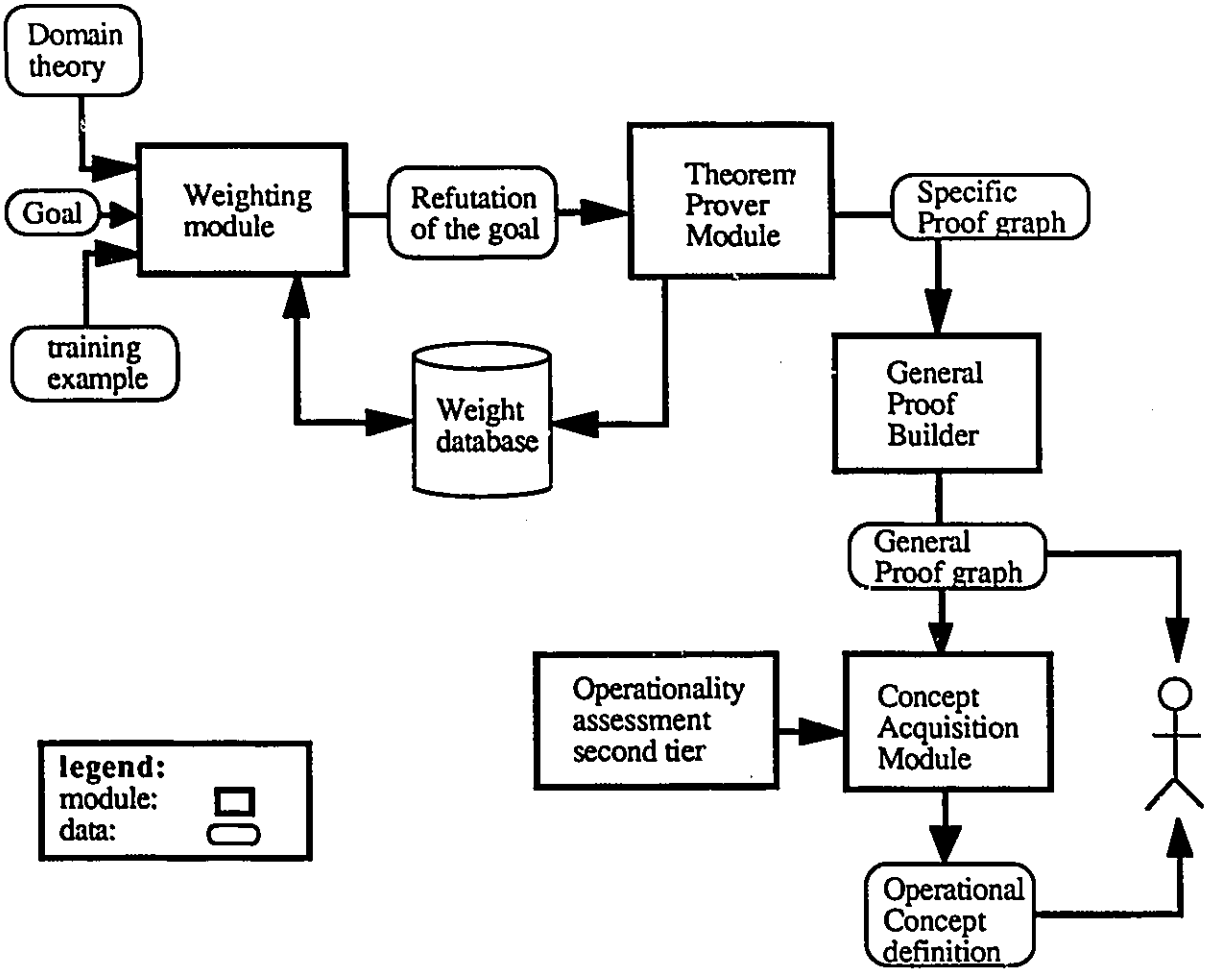

Fig. 8.1 Functional view of the LODIC implementation

Weighting module : attribution of weights to all literals and clauses of the domain theory and the training example.

Theorem prover module : From the refutation of the goal, it derives a usable trace of the proof by using input resolution with an ordering strategy, a set-of-support strategy and non-Horn heuristics.

General Proof Builder: Once the proof based on the training example is established, the GPB builds the GENERAL proof structure, and gives the general structure as output to the user.

Concept acquisition module : This determines the concept definition to be learned from inputs provided by the operationality assessment module. Finally, it gives the operation concept definition as output to the user.

Operationality assessment module -second tier : Its function is to provide the concept acquisition module to receive any special treatment of operationality, according to the preference criterion of the user. It is added as an independent piece of code to allow the user to modify it easily.

8.2 High-level algorithms

8.2.1.Weighting module algorithm

WHILE there are still literals and clauses to be weighed DO
IF the clause is the goal concept THEN
• assign a weight of 0 to the goal
• assign a weight of 1 to each literal of the goal concept
• assign a weight of 0 to the goal concept clause
ELSE
IF the clause is a fact (normally a single literal) from the training instance THEN
• assign a weight of 4 to the literal
• build a de-instantiated fact (DIF), if possible
• assign a weight of 3 to the DIF
ELSE it is a clause from the domain theory
REPEAT for each literal of the clause
IF the literal cannot be subsumed by any previous literal THEN
• assign to the literal a weight of 2
ENDIF
UNTIL all literals are treated.
• find the lightest literal of the clause and assign its weight to the clause
ENDIF
ENDIF
ENDWHILE of the weighting

8.2.2 Theorem Prover algorithm

WHILE the goal is not the empty clause DO
IF one step of resolution can be proven THEN
- find the weight of the lightest literal of the goal clause.
- resolve away the literal by giving the input clause that contains the complementary literal (the negation of the literal to be resolved upon) a clause weight greater than or equal to the weight of the goal to be resolved upon.
- build the substitution list for the GENERAL case.
- return the remaining literals of the input clause.
- return the type of input clause used (whether Horn or non-Horn clauses).
- merge the remaining literals of the goal and the remaining literals of the input clause to form a new goal (resolvent).

IF there is no remaining literal THEN
- the new goal to be resolved is the empty clause.

ELSE
- Assign a weight to the new goal
- Add the new goal to the SPECIFIC proof graph.

IF the state of the input clause is a non-Horn clause THEN
IF it is not the first time this non-Horn clause is used THEN
- retract all previous branch resolvent that used that non-Horn clause
- conjoin them
- add the conjunction to the proof graph
- resolve the first conjunct with the non-Horn clause clause to form the new resolvent

REPEAT
resolve this new resolvent with the next element of the conjunction to form the next resolvent.
UNTIL there is no conjunct left.
ENDIF
ENDIF

ENDIF

ELSE backtrack[2]

IF the step to backtrack uses a non-Horn clause THEN

• save the branch resolvent indexed by the non-Horn clause.

ENDIF

ENDIF

ENDWHILE of the theorem prover

8.2.3 Acquisition Concept algorithm

The last algorithm delineates the two last modules of the program : the General Proof Builder and the Concept Acquisition module.

Build the GENERAL proof with the substitution list .

Assess the operationality of the potentially operational clause, depending on the option chosen by the user.

IF this option is the classical EBG criterion THEN

IF there is a level 3 THEN

• negate the last clause of level 3

ELSE negate the first clause of level 4.

ENDIF

ELSE

IF the option is the last clause of a specific level THEN

IF level-option is 2 THEN

• negate the last clause of level 2

ELSE negate the last clause of level 3

ENDIF.

ELSE the selected option is the disjunctive preference criterion

REPEAT for each clause in levels 2 and 3

---

[2]The backtracking algorithm is not presented here. We rely on the Prolog interpreter for the backtracking mechanism.

• calculate the disjunctive weight of a clause by attributing a weight of
20 to disjunctions and a weight of 1 to conjunctions.
• Keep the heaviest clause(s).
UNTIL no clause remains
• negate the resulting clause(s).
ENDIF
ENDIF
Output to the user the new concept definition.

8.3 Program specifics.

8.3.1 Unification problem

The most basic operation of Prolog is the unification of a variable against a term. Unification in Prolog gives a common instantiation of both terms. However, this effect of unification is not always desirable, as there are cases where it is not required, as when matching literals with the weight database. The literals of the weight database can be non-instantiated, partially instantiated or fully instantiated. We do not want to modify any literal in the weight database, but simply to match the literal, even though literals from the weight database can modify the literal to match. We therefore need a special, one-way unification.

LODIC solves this problem by subsuming unification. Subsumption succeeds if the literal to be matched is more specific than the literal in the weight database. Literals in the database are not modified, but the literal to match is unified with the matching literal in the database.

8.3.2 Redundancy problem

The multiple appearances of the same literal in a clause, with different variables may sometimes be redundant. When a literal is fully instantiated, it is clear what is redundant and what is not, but in the case of uninstantiated literals it is not so obvious. In LODIC,

when the general proof-graph is built, the merging of remaining literals from the previous resolvent and the input clause can sometimes introduced duplication of the same literal with different variables. The problem of multiple occurrences of the same literal in a clause is hard to decide when it is appropriate to delete them.

Consider the `block` example, at one point in the general proof the resolvent obtained is : `(not location(A) or not numb(B) or not numb(C) or not location(D))`. From the theory, human know that `location(A)` and `location(D)` are redundant but `numb(B)` and `numb(C)` are not. LODIC although, does not know how to distinguish the two cases. This remains an open issue in the current implementation.

Chapter 9

Conclusion

The problem of including uncertainties in concept learning by means of disjunctions in the domain theory and in the concept description has been the subject of this research. This thesis has presented an extension of the EBG framework that deals with disjunctive concepts. In this chapter, we conclude our work with a review of the principal results and a discussion of possible extensions of the research.

9.1 Summary

This thesis has shown how Mitchell's Explanation-Based Generalization can be extended to include explicit disjunctions in the domain theory and in the operational concept definition. It obviates the need for the strict assumption that the domain theory must be expressed in Horn clauses, and it uses a more general mode of expression by allowing non-Horn clauses. This approach makes explanation-based learning more flexible by extending the representation of the problem.

The insertion of non-Horn clauses was permitted by modifying the inference engine of EBG. The new inference engine uses non-Horn heuristics in addition to input resolution. The novelty of this approach to resolution is to order and restrict the selection of the literal to be resolved by a weighting strategy, and to develop special heuristics to handle non-Horn clauses. The aggregate of these strategies makes the inference engine a powerful explanation tool.

The other contribution of this thesis is to offer a new approach to operationality. The operationality criterion can be defined at the clause-level rather than in a goal-based manner. The advantages of this approach are the flexibility provided to the user, the independence of the criterion from the training example, and that operationality is made dynamic.

Our analysis of the method demonstrates that it is sound and the degree of incompleteness does not seem to create problems in practice. The system is well-founded by its implementation in the language PROLOG, and successful in its use by acquiring concept definitions for all the examples given in this thesis.

9.2 Future work

There are a number of promising directions in which this work can be extended. First, the context of learning can be expanded. Second, an investigation of selection methods where there is more than one alternative can be undertaken. Third, different inference methods can be explored. Fourth, the backtracking of LODIC can be enhanced to be more efficient and less redundant. And finally, the user interface of LODIC can be improved.

9.2.1 Extension of the context of learning

The learning approach can be enhanced by including LODIC to a performance element, which would use uncertainty in the domain theory in the form of disjunctive rules. LODIC would acquire new concepts that represent this uncertainty and extend the domain theory.

9.2.2 Selection methods

Currently, LODIC has a very simple conflict resolution method to select the lightest literal, and also to match clauses of the same weight: it chooses the first one. There may be better approaches, such as matching clauses by selecting the clause with the smallest length, and hence introducing fewer new literals into the proof. Another approach would be to perform a look-ahead, to discover the depth of the search needed for that clause.

The lightest literal could be selected from the number of instantiated arguments. Preference would be given to the most general literal to preserve the generality property of the proof.

Naturally, an empirical analysis would be needed to know the benefit of better selection methods, and whether they are worth the computational effort.

9.2.3 Inference methods

A number of issues relate to the use of input-resolution with non-Horn heuristics, as mentioned in chapter 6. It will be useful to look at a non-resolution method to solve or alleviate these problems. Tableau methods [Oppacher et al. 1988] are a good candidate for this experiment.

9.2.4 Backtracking in LODIC

Currently, LODIC uses a simple backtracking strategy that is invoked whenever an unsolvable sub-problem is encountered. If a failure occurs while solving a sub-problem, our Prolog system will discard all deductions between the point of failure and the last such sub-problem for which there is (possibly) an alternative solution. As there is no mechanism that leads the system directly to the source of failure of the proof, the name *blind backtracking* seems to suit this method well.

In chapter 6, we showed that the exponential cost of the non-Horn heuristics and of resolution in general is caused by backtracking. It would be more efficient to detect the exact source of backtracking and some research has been done in this area, called intelligent backtracking (see [Matwin et al. 1985]).

LODIC backtracking could be enhanced easily where non-Horn heuristics are used. Once a non-Horn heuristic has been applied, it is easy to trace the literals of the non-Horn clause, and their descendants introduced in the proof. If resolution fails and some remaining literals at the point of failure are marked literals that were introduced by the non-Horn, then the only possible alternative is to seek a resolvent that does not contain the literal that needed the non-Horn clause. Rather than backtracking all possible alternatives between the point of failure and the point of introducing the literal that needed the non-Horn clause (which are bounded to fail), all deductions will be discarded up to the point of introduction of the literal that needed the non-Horn clause, and a search will be started for an alternative at this point only.

9.2.5 User interface enhancement

The user interface in our current implementation is very limited. The user issues commands at the Prolog interpreter prompt. A basic help facility ensures that the user is reminded of the commands at any time. A desirable enhancement of our implementation of LODIC would be to provide windows. The user would then have a window on the domain theory, a window on the proof graph obtained, a window for the concept definition acquired, and a command window. This could be done using the PRO-WINDOW package on QUINTUS Prolog.

The interface could also be improved by drawing the proof graph as a graph rather than as indented text. This improvement would require a graphic program to handle branching and the formation of diamonds.

References

Bibel, W. (1987). *Automated Theorem Proving*, Artificial Intelligence. Vieweg, Braunschweig.

Chang, C. and Lee, R.C. (1973). *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York.

d'Alché, F., Salembier, M. and Matwin, S. (1990) "Learning Disjunctive Concepts", Research Report, University of Ottawa, TR-90-05, 1990

DeJong, G.F. and Mooney, R. (1986). "Explanation-Based Learning: An Alternative View", *Machine Learning*, vol. 1, pp. 145-176,

Dietzen, S. and Pfenning, F. (1989) "Higher-Order and Modal Logic as a Framework for Explanation-Based Generalization", Research Report, Carnegie Mellon University, Pittsburgh, CMU-CS-89-160, 1989

Ellman, T. (1989). "Explanation-Based Learning: A survey of Programs and Perspective.", *ACM Computing Surveys*, vol. 21, no. 2,

Gallier, J., H. (1986). *Logic for Computer Science - Foundations of Automatic Theorem Proving*, Harper & Row, New York.

Hirsh, H. (1987) "Explanation-based generalization in a logic-programming environment, " Procs. of *Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, pp. 221-227.

Hirsh, H. (1988) "Reasoning about Operationality for Explanation-Based Learning, " Procs. of *5th International Conference on Machine Learning*, Los Altos, Calif., pp. 214-220.

Kanal, N. and Lemmer, J.F. ed. (1986). *Uncertainty in Artificial Intelligence*, 4, North Holland, Amsterdam.

Kedar-Cabelli, S.T. and McCarty, L.T. (1987) "Explanation-Based Generalization as Resolution Theorem Proving, " Procs. of *ML87*, pp. 383-389.

Keller, R.M. (1988). "Defining Operationality for Explanation-Based Learning", *Artificial Intelligence*, vol. 35, pp. 227-241.

Kowalski, R. (1979). *Logic for Problem Solving*, Artificial Intelligence. Nilsson (ed.), North Holland, New York.

Lloyd, J.W. (1984). *Foundation of Logic Programming*, Symbolic Computation - Artificial Intelligence. Springer-Verlag, Berlin.

Marie , X. and Habib, M. (1989) "A graph algorithm for learning disjunction, " Procs. of *4th Euporean Workshop Sessions on Learning*.

Matwin, S. and Pietrzykowski, T. (1985). "Intelligent Backtracking in Plan-Based Deduction", *IEEE trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no.6, no. November 1985, pp. 682-692.

Michalski, R.S., Carbonell, J.G. and Mitchell, T.M. ed. (1983). *Machine Learning an Artificial Intelligence Approach*, 1, Morgan Kauffman, Palo Alto, Calif.

Mitchell, T., Keller, R.M. and Kedar-Cabelli, S.T. (1986). "Explanation-Based Generalization: A Unifying View", *Machine Learning*, vol. 1, pp. 47-80.

Mooney, R. (1989) "The Effect of Rule Use on the Utility of Explanation-Based Learning, " Procs. of *IJCAI 1989*, pp. 725-730.

Musen, M. (1990), Private communication, University of Stanford.

Oppacher, F., Suen E (1988). "HARP: A Tableau-Based Theorem Prover", *Journal of Automated Reasoning*, vol. 4, pp. 69-100.

Robinson, J.A. (1965). "A Machine Oriented Logic Based on the Resolution Principle", vol. 12, no. January 1965, pp. 23-41.

Salembier,M., Matwin, S., d'Alché, F., Learning of Disjunctive Concept with Explanation-Based Learning, Procs. of *ISMIS 90,* Knoxville, Tennessee.

Segre, A.M. (1988) "Operationality and real world plans, " Procs. of *AAAI Spring Symposium on Explanation-Based Learning*, Menlo Park, Calif., pp. 158-163.

Sickel, S. (1976). "A Search technique for Clause Interconnectivity Graphs", *IEEE transactions on Computers*, vol. C-25, no. 8 - August 1976, pp. 823-835.

Winston, H., P. (1984). *Artificial Intelligence*, Addison-Wesley, Massachusetts.

Wirth, R. (1989) "Completing Logic Programs by Inverse Resolution, " Procs. of *4th European Working Session on Learning*, Montpellier, pp. 239-250.

Wos, L., Overbeek, R., Lusk, E. and Boyle, J. (1984). *Automated Reasoning - Introduction and Applications*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Appendix A

Listing of LODIC

The code of LODIC is located in six files. The content of the files are:

interface.pl	contains the interface to the user.
weight.pl	This files contains the weighting procedure that creates the database.
prover.pl	It contains procedures for the theorem proved that constructs the explanation structure required by EBG. The theorem prover is the inference engine of EBG.
resolution.pl	contains the procedures to resolve one step of the proof.
tools.pl	contains a collection of programming tools. record tools : record and retrieve information in the database. logical tools: performs logical operations on terms.
generalizer.pl	contains procedures to derived the final concept descriptions

This appendix presents the six files in the order mentioned above, in addition to the propositional version of some files.

```
/* INTERFACE.PL */
/*************************************************************************
 LODIC (Learning of DIsjunctive Concepts)

 LODIC is a learning system design to allow the use of non-Horn clauses in
 the domain theory and in the acquisition of the final concept. Non-Horn clauses
 are the means by which uncertainty is introduced in domain application.

 For a complete explanation of the system, please consult:
       Maude Salembier-Pelletier, Learning of disjunctive concepts with
       Explanation-Based Learning, Master Thesis, University of Ottawa, 1990.

 Required files:
 ---------------
 weight.pl :    This file contains the weighting procedure that creates the
                weight database
 prover.pl :    This file contains the theorem prover that constructs the
                explanation required by EBG. The theorem prover is the
                inference engine of EBG.
 resolution.pl: This file contains the procedures to resolve a step of
                the proof.
 tools.pl   :   This file contains a collection of programming tools.
                -The record tools to record and retrieve entry in the database.
                -The logical tools that perform logical processing of terms.
 generalizer.pl : contains procedures to derived the final concept description.

 block.pl:      contains a demo of the block example.


 Inputs from the user:
 ---------------------
       LODIC prompts you for the name of the domain theory and also for the
       operationality criterion.
       To use the demo, enter block. To use your own domain theory, make your
       own file in the required format and enter the name. The domain theory
       of the block example is a good example.

 STARTING up:
 ------------
       To start up the program, enter the following command:
       ebg(+General_goal, +Specific_goal, -Specific_proof, -General_proof,
       -Final_concept).

 Notation convention:
 --------------------
       In a comment:
       +X : means an input variable,
       -X : means an output variable,
       ?X : means a temporary variable.

 *************************************************************************/

%%%%%%%% using external built-in predicates %%%%%%%%%%%%%%%%%%%%%%%%%
:- use_module(library(subsumes), [subsumes/2,variant/2]).
:- use_module(library(same_functor), [same_functor/2]).
:- use_module(library(basics),[append/3,member/2,memberchk/2]).
:- use_module(library(lists), [remove_dups/2,rev/2,nth0/3]).
:- use_module(library(change_arg),[change_arg/4]).
:- use_module(library(sets),[seteq/2]).

%%%%%%%% operators needed by the program %%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- op(1130, xfy, or).
:- op(1120, xfy, and).
:- op(1100, fy, no).
:- op(1150, xfy, =>).
```

```
%%%%%%%% files that construct the program %%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- ensure_loaded(weight).
:- ensure_loaded(prover).
:- ensure_loaded(resolution).
:- ensure_loaded(tools).
:- ensure_loaded(generalizer).
:- ensure_loaded(tests).

/*************************************************************************/


/* ------------------------------------------------------------------------
   ebg(+General_goal, +Specific_goal, -Specific_proof, -General_proof,
        -Final_concept)
   explanation-based generalization program finds a specific explanation,
   generalizes it, and gives as an output the final concept derived from
   the initial concept in an operational form.
   ------------------------------------------------------------------------*/

ebg(GGoal,IGoal,SProof,GProof,Concept):-
        ask_file(File),
        consult(File),
        treatment_of_literal,
        write('All the literals and clauses have being weighted'), nl,
        write('Starting the proof now ...'), nl,
        proof((no IGoal),(no GGoal),SProof,GProof),
        write('The proof is done.'),nl,
        write_the_output(SProof,GProof),
        ask_operationality(Option_list),
        concept_acquisition(GProof,Option_list,Concept),
        write('*** The final concept definition is :'),
        numbervars(Concept,0,_),
        write(Concept),
        clean_up.


/* ***************** Interface to the user ******************** */


ask_file(File) :-
        write('Please, give the name of the file that contains the example '),
        nl,
        write('in the form of clauses like "a and b => c or d" '), nl,
        write('File name: '),
        read(File),nl.

/* ------------------------------------------------------------------------
   ask_operationality(-Option_list) finds the operationality criterion
   to consider for the acquisition of the final concept.
   ------------------------------------------------------------------------ */

ask_operationality(Option_list):-
        write('Enter the operationality criterion option:'), nl,
        write('1. Classical EBG criterion'), nl,
        write('2. Level-based operationality'), nl,
        write('3. Two-tiered operationality'),nl,
        read(Option),
        treat_option(Option,Option_list).

/* ------------------------------------------------------------------------
   treat_option(+Option, -Option_list) builds a list with the information
   needed for the operationality criterion based on the option selected
   ------------------------------------------------------------------------ */
```

```
treat_option(1,[1]).
treat_option(2,[2,Level]):-
        write('Enter the specific level, 2 or 3: '),nl,
        read(Level),
        Level > 1,
        Level < 4,!.
treat_option(2,[2,_]):-
        write('Error, please enter the level again'),
        treat_option(2,[2,Level]).
treat_option(3,[3,Filename]):-
        write('Enter the filename of the operationality assessment procedure'),
        nl,
        read(Filename).

/* ------------------------------------------------------------------------
    Procedures for writing the output to the user.
   ------------------------------------------------------------------------ */

write_the_output(SProof,GProof):-
        write('*** The specific proof ***'), nl,nl,
        rev(SProof, RevSProof),
        write_proof(RevSProof),
        write('            NIL'),nl,nl,
        write('*** The general proof  ***'), nl,nl,
        rev(GProof,RevGProof),
        write_proof(GProof),
        write('            NIL'),nl.


write_proof([]):-!.
write_proof([X,Y|T]):-
        write('     '),
        write(X),
        write(' -- '),
        write(Y), nl,
        write('             | '),nl,
        write_proof(T).
```

```
/* WEIGHT.PL */
/* ------------------------------------------------------------
   treatment_of_literal performs the loop to treat all the facts,
    rules, and the goal.
   ------------------------------------------------------------ */
treatment_of_literal :-
        attribute_weight, treatment_of_literal.
treatmert_of_literal .

/* ------------------------------------------------------------
   attribute_weight(+Term) attributes a weight to all the literals
   of the input file.
   First case is the goal
   ------------------------------------------------------------ */
attribute_weight:-
        goal(X),
        record_z(w,weight(X,0)),
        X =.. [_|Arg_list],
        record_z(g,goal_list(Arg_list)),
        retract(goal(X)),
        neg(X,Goal),                    % negate the goal for the
        record_z(g,goal(Goal)).         % refutation of the proof

/* ------------------------------------------------------------
   2nd case : it is a fact.
   Facts have a weight of 4.
   Facts are needed to build the fact base .
   ------------------------------------------------------------ */
attribute_weight:-
        fact(X), ! ,
        build_fact_base(X),
        record_z(w,weight((X),4)),
        record_z(4,(X)),
        retract(fact(X)).

/* ------------------------------------------------------------
   3rd case : it is the goal concept.
   Literals of the goal concept have a weight of 1 and assert the
   rule with the lightest literal weight.
   ------------------------------------------------------------ */
attribute_weight:-
        goal_concept(X), !,
        retract(goal_concept(X)),
        is_clause(X,_,Rest),
        treat_goal_concept(Rest),
        goal_concept(X1),
        retract(goal_concept(X1)),
        transform_into_clausal(X1,Clause),
        record_a(0,Clause).


/* ------------------------------------------------------------
   4th case : it is a rule of the domain theory.
   Literals have a weight of 2 (potentially-operational) or
   3 (partially instantiated from the fact-base).
   ------------------------------------------------------------ */
attribute_weight:-
        rule(X), !,
        transform_into_clausal(X,Clause),
        treat_rule(Clause),
        weigh_and_record_rule(Clause),
        retract(rule(X)).

/* ------------------------------------------------------------
   build_fact_base(+Fact) builds a base of partially instantiated
   facts. If the fact contains constant(s)  that are part of the
   goal's arguments list, then keep the constant(s)and replaced
   with unique variables the ones that are not in the goal's argument
   list.
   ------------------------------------------------------------ */
build_fact_base(X) :-
        functor(X,_,Fact_arity),
        replace_constant(X,Fact_arity,New_fact),
        \+ numbervars(New_fact,0,0),
        \+ recorded_(w,weight((New_fact),3)),
        record_a(w,weight((New_fact),3)).
build_fact_base(_).

/* ------------------------------------------------------------
   replace_constant(+Predicate,+Predicate_arity,-New_predicate)
   Replace the constant by a variable if not in the list of goal's
   constants.
   ------------------------------------------------------------ */
replace_constant(Old_fact,0,New_fact) :-
        New_fact = Old_fact,!.
replace_constant(Old_fact,Arity,New_fact) :-
        arg(Arity,Old_fact,Fact_argument),
        recorded_(g,goal_list(Goal_arg_list)),
        \+ (member(Fact_argument,Goal_arg_list)),
        change_arg(Arity,Old_fact,New_f,_),
        F is Arity-1,
        replace_constant(New_f,F,New_fact).
replace_constant(Old_fact,Arity,New_fact) :-
        F is Arity-1,
        replace_constant(Old_fact,F,New_fact).
/* ------------------------------------------------------------
   is_clause(+Rule,-Head,-Body) returns the head and body of a rule.
   ------------------------------------------------------------ */
is_clause((Body => Head),Head,Body) :- !.
is_clause(Head,Head,true).

/* ------------------------------------------------------------
   Treat_goal_concept(+Body_of_rule) attributes a weight of 1 to
   each of the literals that compose the goal concept.
   ------------------------------------------------------------ */
treat_goal_concept((Arg1 and Rest)) :-
        !,
        treat_goal_concept(Arg1),
        treat_goal_concept(Rest).
treat_goal_concept((no Arg)):-
        \+ recorded_(w,weight((Arg),_)),
         record_z(w,weight((Arg),1)).
treat_goal_concept(Arg) :-
        \+ recorded_(w,weight((Arg),_)),
        record_z(w,weight((Arg),1)).
treat_goal_concept(_):-!.

/* ------------------------------------------------------------
   Treat_rule(+Clause) gives a weight to the head of the rule and then
   to the rest of the literals. A weight of two is associated if
   the literal is not part of the fact base, otherwise it is 3.
   ------------------------------------------------------------ */
treat_rule((no Literal)):-
        pliteral(Literal),!,
        check_and_put_in(Literal).
treat_rule(Literal):-
        literal(Literal),!,
        check_and_put_in(Literal).
treat_rule((Exp1 or Exp2)):-
        treat_rule(Exp1),!,
        treat_rule(Exp2).
```

```
/* ------------------------------------------------------------
   transform_into_clausal(+Logic_exp, -Clausal) transforms a logic
   expression of the form (a and b => c) into a disjunctive clausal
   form (not a or not b or c).
   ------------------------------------------------------------ */
transform_into_clausal((Cond => Head),(Rcond or Head)):-
        transform_clausal(Cond,Rcond),!.
transform_into_clausal(X,_):-
        write('This formula is not well formed: '),
        write(X), nl.

/* ------------------------------------------------------------
   transform_clausal(+Logic_exp, -Clausal) transforms a logic
   expression of the form (a and b => c) into a disjunctive clausal
   form (not a or not b or c).
   ------------------------------------------------------------ */
transform_clausal((no Literal), Literal):-
        pliteral(Literal), !.
transform_clausal(Literal,(no Literal)):-
        pliteral(Literal),!.
transform_clausal((X and Y), (RX or RY)):-
        !,neg(X,RX),
        neg(Y,RY).
transform_clausal((X or Y), (RX or RY)):-
        !,neg(X,RX),
        neg(Y,RY).

/* ------------------------------------------------------------
   weigh_and_record_rule(+Clause) weigh a clause and record the weight
   and the rule.
   ------------------------------------------------------------ */
weigh_and_record_rule(Clause):-
        copy_term(Clause,Ctemp),
        lightest_weight(Ctemp,_,W),!,
        record_a(W,Clause).

/* ------------------------------------------------------------
   check_and_put_in(+Partial_Fact) checks if the partial fact is already
   part of  the fact database. If not it records it.
   ------------------------------------------------------------ */
check_and_put_in(Fact):-
        recorded_w(w,(weight(Fact,_))),!,
        put_in(Fact).
check_and_put_in(Fact):-
        record_a(w,weight(Fact,2)).

put_in(Fact):-
        \+ recorded_(w,weight((Fact),_)),!,
        record_a(w,weight((Fact),3)).
put_in(_):-!.
```

```
/* PROVER.PL */
/******************* These project ********************/
/***  2nd module : theorem prover in FOL  ***/
/*****************************************************/

/* ------------------------------------------------------
   proof(+Goal,+GGoal,-SProof,GProof) returns the list Proof containing
   the center clauses of the graph resolution of Goal. The
   elements are alternatively a clause and its weight.
   ------------------------------------------------------ */
proof(Goal,GGoal,SProof,GProof) :-
        lightest_weight(Goal,_,W),
        prove([Goal|GGoal],[],[Goal,W],[GGoal,W],SProof,GProof).

/* ------------------------------------------------------
   prove(+Goal, +Memory, ?Acc, -Proof,-Gsub) :
   Goal is the goal to be resolved.
   Memory contains the list of literals that have been resolved
   so far.
   Acc is an accumulator for this current proof.
   Proof is the resulting list containing the last branch of
   the proof of Goal.
   Glist is the list containing all the substitution to create
   the general proof graph.
   1. Stopping condition, it is the last fact and the result is nil.
   2. General step
   ------------------------------------------------------ */
prove(Goal,Memory,List,GList,List,GList):-
        prove_one_step(Goal,empty,fact,Memory,Memory),!.

prove(Goal,Old_memory, Acc,GAcc,List,Glist) :-
        prove_one_step(Goal,[New_goal,Weight],State,Old_memory,New_memory),
        process(State,[New_goal,Weight],New_memory,Acc,List,GAcc,Glist).

/* ------------------------------------------------------
   process(+State,+[New_goal,Weight],+Memory,?Acc,-List,?GAcc,-Glist)
   process the next step according to the state of the
   resolving clause that creates the new goal to solve.
   The Memory accumulates the predicate already solved.
   The List will contain the resolvents of the proof.
   ------------------------------------------------------ */
process(horn,[[New_goal|New_Ggoal],Weight],Memory,Acc,List,GAcc,Glist):-
        prove([New_goal|New_Ggoal],Memory,[New_goal,Weight|Acc],
        [New_Ggoal,Weight|GAcc],List,Glist).

process(clause,[[New_goal|New_Ggoal],Weight],Memory,Acc,List,GAcc,Glist):-
        prove([New_goal|New_Ggoal],Memory, [New_goal,Weight|Acc],
        [New_Ggoal,Weight|GAcc],List,Glist).

process(fact,[[New_goal|New_Ggoal],Weight],Memory, Acc,List,GAcc,Glist):-
        prove([New_goal|New_Ggoal],Memory,[New_goal,Weight|Acc],
        [New_Ggoal,Weight|GAcc),List,Glist).

process(non_horn_found,[C,Weight],Memory,[Goal|Acc],List,[Ggoal|GAcc],Glist):-
        diamond_process([Goal|Ggoal],[Goal|Acc],[Ggoal|GAcc],[Conj|GConj]),
        resolve_diamond([Conj|GConj],C,[Res|GRes]),
        lightest_weight(Res,_, W),
        prove([Res|GRes],Memory,[Res,W|[Conj,Weight|[Goal|Acc]]],
                [GRes,W|[GConj,Weight|[Ggoal|GAcc]]],List,Glist).

process(non_horn_found,_,_,[Goal|Acc],_,[GGoal,GAcc],_):-
        recorded_(nonhorn,used((B or H),[L])),
        retract_(nonhorn,used((B or H),[L])),
        record_a(nonhorn,used((B or H),[Goal|L])),
        record_a(preuve,branch(_,[Goal|Acc],[GGoal|GAcc],[])),
        !,fail.

process(non_horn_transf,[[New_goal|New_Ggoal],Weight],Memory,Acc,List,GAcc,Glist
        record_a(preuve,branch(_,Acc,GAcc,[])),
        prove([New_goal|New_Ggoal],Memory, [New_goal,Weight|_],
        [New_Ggoal,Weight|_],List1,Glist1),!,
        append(Glist1,GAcc,Glist),
        append(List1,Acc,List).

process(non_horn_transf,[_,_],_,_,_,_,_):-
        recorded_(nonhorn,transf(B,H,P)),
        record_a(nonhorn,used((B or H),[P])),
        retract_(nonhorn,transf(B,H,P)),
        !,fail.

/* ------------------------------------------------------------------------
   prove_one_step(+Goal,-[New_goal,Weight],?State,+Old_memory,-New_memory)
   makes one step of resolution for Goal, finding the next goal to resolve,
   updating the state and the list of predicate that are solved (new_memory).
   ------------------------------------------------------------------------ */
/* Last fact - stopping condition  */

prove_one_step([(no A)|(no G)],empty,fact,Old_memory,New_memory):-
        literal(A),
        recorded_v(w,weight(A,4)),
        resolve([(no A)|(no G)],4,empty,fact,Old_memory,New_memory).

/* General step  */

prove_one_step([(no A)|(no G)],[[New|New_G],W],State,Old_memory,New_memory):-
        literal(A),
        recorded_(w,weight(A,W1)),
        resolve([(no A)|(no G)],W1,[New|New_G],State,Old_memory,New_memory),
        lightest_weight(New,_,W).

prove_one_step([A|G],[[New|New_G],W],State,Old_memory,New_memory):-
        literal(A),
        recorded_(w,weight(A,W1)),
        resolve([A|G],W1,[New|New_G],State,Old_memory,New_memory),
        lightest_weight(New,_,W).

prove_one_step([A|G],[[New|New_G],W],State,Old_memory,New_memory) :-
        lightest_weight(A,X,W1),
        find_in(X,G,GX),
        resolve([X|GX],W1,[Res|GRes],State,Old_memory,New_memory),
        joined_res(Res,X,A,New),
        joined_res(GRes,GX,G,New_G),
        lightest_weight(New,_,W).

/* ------------------------------------------------------------------
   joined_res(+Clause,+Resolve_literal,+Resolving_clause,
              -New_resolving_clause, -Weight)
   creates the logical consequence of two resolvents.
   ------------------------------------------------------------------*/
joined_res([],X,A,R):-
        substract(X,A,R),!.
joined_res(XCond,X,A,(R or XCond)):-
        !,substract(X,A,R).

/* ------------------------------------------------------------------
   find_in(+Literal,+Clause,-Gliteral) finds in the clause the
   general form of Literal.
   ------------------------------------------------------------------*/
find_in(Literal,[],[]):-!.
find_in(Literal,L,L):-
        return_pred(L,Pred1),
        return_pred(Literal,Pred2),
```

```
        same_functor(Pred1,Pred2),!.
find_in(Literal,(L or _), L):-
        return_pred(L,Pred1),
        return_pred(Literal,Pred2),
        same_functor(Pred1,Pred2),!.
find_in(Literal,((L or _) or _),Gliteral):-
        find_in(Literal,L,Gliteral),!.
find_in(Literal,((_ or R) or _),Gliteral):-
        find_in(Literal,R,Gliteral),!.
find_in(Literal,((_ or _) or Rest),Gliteral):-
        find_in(Literal,Rest,Gliteral).
find_in(Literal,(_ or Rest),Gliteral):-
        find_in(Literal,Rest,Gliteral),!.

/* ------------------------------------------------------------------
   diamond_process(+Goal,+Memory,+Acc,-The_result_of_the_link,
                   -New_memory,-Disjunction_form) forms the diamond
    save the branch that is missing.
   ------------------------------------------------------------------ */

diamond_process([Goal|Ggoal],Acc,GAcc,[Conj1|Conj2]):-
/*  find the last branch */
        recorded_(preuve, branch(First,List,[GX,GY|Glist],[])),
        !,
/*  find the common part and the junction node */
        find_junction(Acc,List,Node1,[X,Y|R_list]),
/* find the list of the clauses that have to be linked */
        find_all_branch(Node1,[],Br_list),
/* put these parts together*/
        make_and([Goal,X| Br_list], Conj1),
        make_and([Ggoal,GX | Br_list],Conj2),
/* translate in DNF */
        retract_(preuve, branch(First, List,_, [])),
        record_a(preuve, branch(X,[X,Y|R_list],[Node1],[GX,GY,Glist])).


/* ------------------------------------------------------------
   find_junction(+Acc,+Branch,-Junction_node,-Rest_of_the_branch)
   finds the junction and return the rest of the branch.
   ------------------------------------------------------------*/
find_junction(Acc,Branch,Node_1,R_branch):-
        append(_,[Node_1,N|Y],Acc),
        append(R_branch,[Node_1,N|Y],Branch),
        \+ number(Node_1).

/* ------------------------------------------------------------
   find_all_branch(+Junction_node,+Temporary,-List) finds all the previous
   non_horn branch that have the same junction node.
   ------------------------------------------------------------*/
find_all_branch(Jnode,_,[]):-
        \+ recorded_(preuve,branch(_,_,[Jnode],_)),!.
find_all_branch(Jnode,Temp,[]):-
        Temp \== [],
        recorded_(preuve,branch(X,_,[Jnode],_)),!,
        member(X,Temp).
find_all_branch(Jnode,Temp,[X|Res]):-
        recorded_(preuve,branch(X,_,[Jnode],_)),
        find_all_branch(Jnode,[X|Temp],Res),!.

/* ------------------------------------------------------------
   trans_in_dnf(+Conj, -Disj) transforms a conjunction into a
    disjunction.
   ------------------------------------------------------------*/
trans_in_dnf(X,X):- literal(X),!.
trans_in_dnf((no X),(no Y)):-
        trans_in_dnf(X,Y),!.
trans_in_dnf((X and Y), (no ((no X) or (no Y)))):-
        literal(X),
        literal(Y),!.
trans_in_dnf(((no X) and (no Y)),(no (X or Y))):-
        literal(X),
        literal(Y),!.
trans_in_dnf((X and Y), (no ((no X)or Res))):-
        literal(X),
        trans_in_dnf((no Y),Res),!.
trans_in_dnf(((no X) and Y),(no (X or Res))):-
        literal(X),
        trans_in_dnf((no Y),Res),!.
trans_in_dnf((X and Y),(no (R1 or R2))):-
        trans_in_dnf((no X),R1),
        trans_in_dnf((no Y),R2).
```

```
/* ------------------------------------------------------------------
   RESOLUTION.PL contains the set of predicates which are used to resolve
   a literal with unification.
   ------------------------------------------------------------------ *

/* ------------------------------------------------------------------
   resolve(+Literal,+Weight,-Resolvent,-Resolving_type,+Old_memory,
           -New_memory,Gsublist)  resolution of a literal. The resolving type
         is either  fact, horn, non_horn, or clause. Memory is used to
         keep track of the literals that were solved in order to detect
         loops in the resolution.
   ------------------------------------------------------------------*/

/* ------------------------------------------------------------------
                              facts
   ------------------------------------------------------------------ */

resolve([(no P)|_],4,empty,fact,_,_):-
        recorded_(4,P).
resolve([P|_],4,empty,fact,_,_):-
        recorded_(4,(no P)).

/* ------------------------------------------------------------------
                        resolution of literal P
   ------------------------------------------------------------------ */

resolve([(no P)|(no G)],W,[B|GB],horn,Old_memory,[(no P)|Old_memory]):-
        find_horn(W,G,GB),
        copy_term((GB or G),Temp),
        Temp = (B or P),
        empty_inter(B, Old_memory).
resolve([(no P)|(no G)],W,[(B or Res)|(GB or GRes)],Param,Old_memory,[(no P)|Old
        setof(B1,find_nh(W,G,B1),L),
        noempty(L),
        member([(GB or H),R],L),
        state_nh(P,GB,H,R,Param,GRes),
        copy_term((GB or GRes or G),Temp),
        Temp = (B or Res or P).
resolve([(no P)|(no G)],W,[(H or B)|(GH or GB)],clause,Old_memory,[(no P)|Old_me
        find_clause(W,G,[GH,GB]),
        copy_term((GH or GB or G),Temp),
        Temp = (H or B or P),
        empty_inter((H or B),Old_memory).
resolve([(no P)|(no G)],W,[C|GC],clause,Old_memory,[(no P)|Old_memory]):-
        copy_term(G,GTemp),
        find_clause(W,GTemp,GC),
        copy_term((GC or GTemp),Temp),
        Temp = (C or P),
        empty_inter(C,Old_memory).
resolve([P|G],W,[C|GC],clause,Old_memory,[P|Old_memory]):-
        copy_term(G,GTemp),
        find_clause(W,(no GTemp),GC),
        copy_term((GC or GTemp),Temp),
        Temp = (C or P),
        empty_inter(C,Old_memory).

/* ------------------------------------------------------------------
   resolve_diamond(+Clause_from_diamond,+Input_clause,-Res)
   find the result of applying the Non-horn clause to the clause form
   by the diamond.
   ------------------------------------------------------------------ */

resolve_diamond([Clause|GClause],[Input_clause|GInput],[Res|GRes]):-
        liste_literals(Clause,L),!,
        substractn(L,Input_clause,Res),
        liste_literals(GClause,L2),
        substractn(L2,GInput,GRes).

/* ------------------------------------------------------------------
   find_horn(+Weight,+Predicate,[-Resulting_clause])
   finds a valid Horn clause that contains the predicate P and
   returns the Horn clause and the resulting clause - P.
   The weight of the clause is >= to the weight of P.
 ------------------------------------------------------------------*/
find_horn(W,P,B):-
        recorded_(W, (B or P)),
        horn_clause(B, P).
find_horn(W,P,B):-
        W < 4,
        W1 is W + 1,
        find_horn(W1,P,B).

/* ------------------------------------------------------------------
   find_nh(+Weight,+Predicate,[-Non-horn_clause,-Resulting_clause])
   finds a valid non-horn clause that contains the predicate P and
   returns the non_horn clause and the resulting clause - P.
   The weight of the clause is >= to the weight of P.
  ------------------------------------------------------------------*/
find_nh(W,P,[(B or H),R]):-
        recorded_(W,(B or H)),
        non_horn_clause(B,H),
        in(P,H,R).
find_nh(W,P,[(B or H),R]):-
        W < 4,
        W1 is W + 1,
        find_nh(W1,P,[(B or H),R]).

/* ------------------------------------------------------------------
   find_clause(+Weight,+Predicate,-Resulting_clause) finds a
   clause of weight W that contains the Predicate.
   The weight of the clause is >= to the weight of P.
 ------------------------------------------------------------------*/
find_clause(W,P,R):-
        recorded_(W, C),
        in(P,C,R),
        \+ horn_clause(C).
find_clause(W,P,R):-
        W < 4,
        W1 is W + 1,
        find_clause(W1,P,R).

/* ------------------------------------------------------------------
   empty_inter(+Exp, +Liste) succeeds if the intersection
   between the expression and the list is empty.
 ------------------------------------------------------------------*/
empty_inter(_,[]):-!.
empty_inter(Exp,Liste):-
        !,
        common(Exp,Liste,R),
        R == [].
empty_inter(_,_):-!,fail.

/* ------------------------------------------------------------------
   common(+Exp,+Liste,-Common_liste) accepts an expression
   and a liste and returns in a liste the common elements in
   the expression and the liste.
 ------------------------------------------------------------------*/

common((H),Liste,[H]):-
        literal(H),
        my_member(H,Liste),!.
```

```
common((H),_,[]):-
        literal(H),!.

common((H or Body),Liste,[H|Res]):-
        literal(H),
        my_member(H,Liste),!,
        common(Body,Liste,Res).
common((_ or Body),Liste,Res):-
        !,common(Body,Liste,Res).

/* ------------------------------------------------------
   state_nh(+Predicate,+Left_handside,+Right_handside,+Rest,
            -Non_horn_state, -Rest)
   finds the state of a non-horn clause.
   1. If this non-horn clause has already being used then one
      more part has been found => state= non_horn_found
   2. If this non-horn clause has already being used then it is
      not a transformation => fail (occurs when backtracking).
   3. If it is the first time this non-horn is used then
      record the non-horn clause in the database.
      State = non_horn_transform.
   ------------------------------------------------------*/
state_nh(P,L,H,_,non_horn_found,H):-
        recorded_(nonhorn,used((L1 or H1),[P1])),
        P \== P1,
        clause_eq((L or H),(L1 or H1)),!.
state_nh(_,L,H,_,non_horn_transf,_):-
        recorded_(nonhorn, used((L1 or H1),_)),
        clause_eq((L or H),(L1 or H1)),!,fail.
state_nh(P,L,H,R,non_horn_transf,R):-
        !,record_a(nonhorn,transf(L,H,P)).

/* ------------------------------------------------------
   clause_eq(+C1,+C2) return true if the two clauses have the
   same predicates in any order.
   Transforms the two clauses into list of terms and do a seteq
   on the two lists.
   ------------------------------------------------------*/
clause_eq(C1,C2):-
        copy_term(C1,C1temp),
        copy_term(C2,C2temp),
        liste_literals(C1temp,L1),
        liste_literals(C2temp,L2),
        seteq(L1,L2),!.
```

```
/* TOOLS.PL */

/* ------------------------------------------------------------------------
   RECORDTOOL contains predicates that use data base references.
   ------------------------------------------------------------------------ */

/* tools for the internal database */
/* ------------------------------------------------------
   I redefined recorda as record_a etc... in order to define
   a corresponding retract_ and to define my own recorded_ to
   subsumes Quintus unification.
   ------------------------------------------------------ */

record_a(Key, Term):-
        asserta('$recorded'(Key, Term)).
record_z(Key, Term):-
        assertz('$recorded'(Key, Term)).
recorded_(Key, Term):-
        clause('$recorded'(Key, Term), _).

recorded_v(Key,Term):-
        %Term1 = weight((max(A,B,C)),X)
        most_general(Term,Term1),
        setof(Term1,'$recorded'(Key,Term1), L),
        notempty(L),
        member(T,L),
        subsumes(Term,T),!.
recorded_v(Key,Term):- !, recorded_(Key,Term).

recorded_w(Key,Term):-
        most_general(Term,Term1),
        setof(Term1,'$recorded'(Key,Term1), L), % create a list
        notempty(L),
        member(T,L),                    % get one member of the list
        arg(1,T,PT),                    % get the predicate only
        arg(1,Term,PTerm),              % from weight(Pred,Weight)
        subsumes(PT,PTerm),!.           % check whether two terms are
                                        % alphabetic variants.

retract_(Key, Term):-
        retract('$recorded'(Key, Term)).

clean_up :-
        retractall('$recorded'(_,_)).


notempty([]):-!,fail.
notempty(_).

/* ------------------------------------------------------------
   most_general(+Specific_term, -General_term)
  Create a term with variables only to detour the unification
  problem.
  ------------------------------------------------------------ */

most_general(Specific, General):-
        Specific =..[P|Args],
        make_it_general(Args,Res),
        General =..[P|Res].

make_it_general([],[]).
make_it_general([A],[A]):-
        var(A).
make_it_general([A|X],[A|Y]):-
        atom(A),
        make_it_general(X,Y).
make_it_general([A|X],[Res|Y]):-
        functor(A,Name,Arity),
        functor(Res,Name,Arity),
        make_it_general(X,Y).

/* ------------------------------------------------------------------------
   LOGICTOOL contains the definitions of the logical predicates
   (operators) and all the predicates used to check if the clauses
   or formulae are valid.
   ------------------------------------------------------------------------ */

/* ------------------------------------------------------
   check_op(+Op1) ??
   ------------------------------------------------------ */
check_op(empty):-  !,fail.
check_op((no X)):-
        literal(X),!.
check_op(((no _) or R)):-
        check_op( R),!.

/* ------------------------------------------------------
   literal(+X) succeeds if X is a literal.
   ------------------------------------------------------ */
literal((_ and _)):- !,fail.
literal((_ or _)):-     !,fail.
literal(((no _) or _)):-!,fail.
literal(X):-            functor(X,_,_).
literal((no X)):-       functor(X,_,_).

/* ------------------------------------------------------
   pliteral(+X) succeeds if X is a literal.
   ------------------------------------------------------ */
pliteral((_ and _)):- !,fail.
pliteral((_ or _)):-    !,fail.
pliteral(((no _) or _)):-!,fail.
pliteral((no _)):-      !,fail.
pliteral(X):-           functor(X,_,_).

/* ------------------------------------------------------
   return_pred(+Literal,-Predicate) returns only the predicate
   of Literal.
   ------------------------------------------------------ */
return_pred((no Literal), Literal):-
       pliteral(literal),!.
return_pred(Literal,Literal):-
        pliteral(Literal).

/* ------------------------------------------------------
   make_and(+Liste, -Conj) accepts a liste of literals and
   returns the conjunction of all literals.
   ------------------------------------------------------ */
make_and([],[]):-!.
make_and([X],X):-!.
make_and([(X)|Xl],(X and Res)):-
        make_and(Xl,Res).

/* ------------------------------------------------------
   make_or(+Liste, -Disj) accepts a liste of literals and
   returns the Disjunction of all literals.
   ------------------------------------------------------ */
make_or([],[]):-!.
make_or([X],X):-!.
make_or([(X)|Xl],(X or Res)):-
        make_or(Xl,Res).

/* chunk(Op, Formula) */
```

```
chunk(or,(X)):-
        literal(X),!.
chunk(or, (X or R)):-
        literal(X),!,
        chunk(or, R).

/* horn_clause */
horn_clause((B or H)):-
        horn_clause(B,H).
horn_clause(B, H):-
        check_op(B),
        pliteral(H).

non_horn_clause(_, P):-
        literal(P),!,fail.
non_horn_clause(B,H):-
        check_op(B),
        chunk(or, H).

/*  ------------------------------------------------------------
    in(+Literal,+Clause,-Rest_of_Clause)
    ------------------------------------------------------------*/

in(Literal,Literal,[]):-
        !,literal(Literal).
in(Literal1,Literal2,[]):-
        Literal1=Literal2,!.
in(Literal,(Literal or Rem), Rem):-!.
in(Literal,(Rem or Literal), Rem):-!.
in(Literal,(Literal2 or Literal),Literal2 ):-
        literal(Literal2),!.
in(Literal,(Literal2 or Remainder), (Literal2 or Rest)):-
        literal(Literal2),!,
        in(Literal,Remainder,Rest).
in(Literal,(Exp1 or Exp2),(Rexp1 or Exp2)):-
        in(Literal,Exp1,Rexp1),!.
in(Literal,(Exp1 or Exp2), (Exp1 or Rexp2)):-
        in(Literal,Exp2,Rexp2),!.

/* ------------------------------------------------------------
   substract, substractn, subtraction, sub_opposite: tools for
   deleting logical terms in a logical expression.
   ------------------------------------------------------------
   substract(+Liste,+Clause,-Difference_of_list_and_clause)
   substract a liste of literals from a clause.
   ------------------------------------------------------------*/
substract(L,C,Res):-
        liste_literals(C,Liste),
        subtraction([L],Liste,R1),
        %remove_dups(R1,R2),
        make_or(R1,Res),!.

/* ------------------------------------------------------------
   substractn(+Liste,+Clause,-Difference_of_list_and_clause)
   substract a liste of literals from a the negation of the clause.
   ------------------------------------------------------------*/
substractn(L,C,Res):-
        liste_literals(C,Liste),
        sub_opposite(L,Liste,R1),
        %remove_dups(R1,R2),
        make_or(R1,Res),!.

/* ------------------------------------------------------------
   substraction(+Liste1,+Liste2,-Difference_of_lists)
   finds the difference of two lists.
   ------------------------------------------------------------*/
substraction([],X,X):-!.
substraction(X,[],X):-!.
substraction([X],[X|X1],X1) :- !.
substraction([X],[Y|Y1],[Y|Res]):-
        X \==Y,!,
        substraction([X],Y1,Res).
substraction([X|X1],L,Res):-
        substraction([X],L,R1),
        substraction(X1,R1,Res).

/* ------------------------------------------------------------
   sub_opposite(+Liste1,+Liste2,-Difference_of_lists)
   finds the difference of two lists eliminating opposite predicates.
   ------------------------------------------------------------*/
sub_opposite([],X,X):-!.
sub_opposite(X,[],X):-!.
sub_opposite([X],[(no X)|X1],X1) :- !.
sub_opposite([(no X)],[X|X1],X1) :- !.
sub_opposite([X],[Y|Y1],[Y|Res]):-
        !,sub_opposite([X],Y1,Res).
sub_opposite([X|X1],L,Res):-
        sub_opposite([X],L,R1),
        sub_opposite(X1,R1,Res).

/* ------------------------------------------------------------
   neg(+Term, -Negated_term) negates a term.
   ------------------------------------------------------------  */
negin((no P),P1):- !,neg(P,P1).
negin((P and Q), (P1 and Q1)):-
        !,negin(P,P1), negin(Q,Q1).
negin((P or Q), (P1 or Q1)):-
        !,negin(P,P1),
        negin(Q,Q1).
negin(P,(no P)).

neg((no P),P):- !.
neg((P and Q),(P1 or Q1)):-
        !, neg(P,P1),neg(Q,Q1).
neg((P or Q),(P1 and Q1)):-
        !, neg(P,P1), neg(Q,Q1).
neg(P,((no P))).

/* ------------------------------------------------------------
   remove_duplicate(+Clause_with_duplicate, -Clause_no_duplicate)
   remove duplicates from a clause.
   ------------------------------------------------------------  */
remove_duplicate((C1 and C2), Res):-
        remove_duplicate(C1,Res1),
        remove_duplicate(C2,Res2),
        append(Res1,Res2,Res),!.
remove_duplicate(Clause,Final):-
        liste_literals(Clause,List).
        remove_dups(List,Nodup),
        make_or(Nodup,Final),!.

/* ------------------------------------------------------------
                    Weighting predicates

   lightest_weight(+Clause,-lightest_literal,-Weight_of_the_lightest)
   finds the lightest literal of a clause and returns the literal and
   the weight.
   ------------------------------------------------------------ */

lightest_weight((no P), (no P),W):-
        literal(P),!,
```

```
        recorded_v(w,weight(P,W)),!.
lightest_weight(P,P ,W):-
        literal(P),!,
        recorded_v(w,weight(P,W)),!.
lightest_weight(E,R,W):-
        E=..[_,A1,A2],!,
        lightest_weight(A1,R1,W1),
        lightest_weight(A2,R2,W2),
        min(W1,R1,W2,R2,W,R),!.

min(W1,R1,W2,_,W1,R1):- W2>=W1,!.
min(_,_,W2,R2,W2,R2):-!.

my_member(_,[]):- !,fail.
my_member(X,[X1|_]):-
        variant(X,X1),!.
my_member(X,[_|T]):-
        my_member(X,T).

/* ------------------------------------------------------------
   copy_list(+Start,+Length,+List,-Sublist) finds the sublist starting
   at element Start and finishing at element End.
   ------------------------------------------------------------*/
copy_list(1,0,List,[]):-!.
copy_list(1,Lenght,[H|T],[H|Res]):-
        NL is Lenght -1,
        copy_list(1,NL,T,Res).
copy_list(Start,End,[H|T],Res):-
        NS is Start -1,
        copy_list(NS,End,T,Res).

/* ------------------------------------------------------
   liste_literals(+Clause,-List_of_predicates)
   returns a list of predicates of a disjunctive clause.
   ------------------------------------------------------*/
liste_literals(X,[X]):-
        literal(X).
liste_literals((X or B),[X|Res]):-
        literal(X), liste_literals(B,Res).
liste_literals((X and  B),[X|Res]):-
        literal(X), liste_literals(B,Res).
liste_literals((X or Y),Res):-
        liste_literals(X,R1),
        liste_literals(Y,R2),
        append(R1,R2,Res).
liste_literals((X and Y),Res):-
        liste_literals(X,R1),
        liste_literals(Y,R2),
        append(R1,R2,Res).

/* ------------------------------------------------------
   liste_literalsn(+Clause,-List_of_negated_literals)
   returns a list with the negation of the literals of
   the clause.
 ------------------------------------------------------*/
liste_literalsn([],[]):-!.
liste_literalsn(X,[(no X)]):-
        pliteral(X),!.
liste_literalsn((no X),[X]):-
        pliteral(X),!.
liste_literalsn((X or B), [(no X)|Res]):-
        pliteral(X),!,
        liste_literalsn(B,Res).
liste_literalsn(((no X) or B), [X|Res]):-
        pliteral(X),!,
        liste_literalsn(B,Res).
liste_literalsn((X or Y),Res):-
        liste_literalsn(X,R1),
        liste_literalsn(Y,R2),
        append(R1,R2,Res).

/* ------------------------------------------------------
   negate(+literal,-negated_literal)
   returns the negation of the literal.
 ------------------------------------------------------*/
negate([],[]):-!.
negate(X,(no X)):-
        pliteral(X),!.
negate((no X), X):-
        pliteral(X),!.
negate((X and Y),((no X) or Res)):-
        pliteral(X),!,
        negate(Y,Res).
negate(((no X) and Y),(X or Res)):-
        pliteral(X),!,
        negate(Y,Res).
negate((X or Y),((no X)  and Res)):-
        pliteral(X),!,
        negate(Y,Res).
negate(((no X) or Y),(X and Res)):-
        pliteral(X),!,
        negate(Y,Res).
/* ------------------------------------------------------
   noempty(+list) returns ok if the list is not empty
 ------------------------------------------------------*/
noempty([]):-!,fail.
noempty([[]]):-!,fail.
noempty(_).
```

```
/* ************************************************************************
GENERALIZER.PL finds a operational concept from the general proof
   in accordance to the operationality criterion chosen by the user.
   There are three ways to define what is operational:
   1. Classical EBG criterion
   2. Last clause of the specified level (2 or 3).
   3. Second tiered operationality, disjunctive preference criterion.
  ************************************************************************/
/* ------------------------------------------------------------
   concept_acquisition(+Proof,+Operationality_Option, -Final_concept)
   finds the final concept from the general proof.
   ------------------------------------------------------------ */
concept_acquisition(GProof,[3,F],Concept):-
        generalize(GProof,[3,F],Gen),
        neg(Gen,Concept).
concept_acquisition(GProof,Option,Concept):-
        generalize(GProof,Option,Gen),
        remove_duplicate(Gen,Gen2),
        neg(Gen2,Concept).

/* ------------------------------------------------------------
   generalize(+Proof,+Operationaliry_Option, -Generalization)
   There are three ways to define what is operational:
   1. Classical EBG criterion
   2. Last clause of the specified level (2 or 3).
   3. Second tiered operationality, disjunctive preference criterion.
 ------------------------------------------------------------ */

generalize(GProof,[1],Generalization):-
        memberchk(3,GProof),!,
        find_last_clause(3,GProof,Generalization).
generalize(GProof,[1],Generalization):-
        find_first_clause(4,GProof,Generalization),!.
generalize(GProof,[2,2],Generalization):-
        memberchk(2,GProof),!,
        find_last_clause(2,GProof,Generalization).
generalize(GProof,[2,2],Generalization):-
        find_first_clause(3,GProof,Generalization),!.
generalize(GProof,[2,3],Generalization):-
        generalize(GProof,[1],Generalization),!.
generalize(GProof,[3,Filename],Generalization):-
        get_potentially_operational_clauses(GProof,Partial_Proof),
        second_tiered_operationality(Filename,Partial_Proof,[Generalization,Dw])

/* ------------------------------------------------------------
   find_last_clause(+Level, +Proof_graph, -Disjunction) finds the
   last clause of the specified level and returns the corresponding
   disjunction.
   ------------------------------------------------------------ */
find_last_clause(Level,GProof,Disj):-
        !,
        nth0(N,GProof,Level),
        GN is N - 1,
        nth0(GN,GProof,Disj).

/* ------------------------------------------------------------
   find_first_clause(+Level, +Proof_graph, -Disjunction) finds the
   first clause of the specified level and returns the corresponding
   disjunction.
   ------------------------------------------------------------ */
find_first_clause(Level,GProof,Disj):-
        rev(GProof,RevProof),!,
        nth0(N,RevProof,Level),
        GN is N + 1,
        nth0(GN,RevProof,Disj).

/* ------------------------------------------------------------
   get_potentially_operational_clauses(+Proof,-Partial_proof)
   returns only clauses of the proof graph within the potentially
   operational levels (currently levels 2 and 3).
        1. Level 3 is there
        1. Level 3 is missing.
   ------------------------------------------------------------ */
get_potentially_operational_clauses(Proof,Partial_proof):-
        nth0(N,Proof,3),!,      % [-,4,-,3,-,2,-,1...]
        nth0(M,Proof,1),
        Length is M-N,
        copy_list(N,Length,Proof,Partial_proof).
get_potentially_operational_clauses(Proof,Partial_proof):-
        nth0(N,Proof,2),!,   % [-,4,-,2,-,1...]
        nth0(M,Proof,1),
        Length is M-N,
        copy_list(N,Length,Proof,Partial_proof).

/* ------------------------------------------------------------
   second_tiered_operationality(+Proof, -Obtained_generalization)
   applies a second operationality criterion to the chosen clauses.
   Currently, the criterion is to prefer disjunctions (negated
   conjunctions of the proof).
   ------------------------------------------------------------ */
second_tiered_operationality(Filename,Proof,Generalization):-
        consult(Filename),
        assess_operationality(Proof,Generalization).
```

```
/*****************************************************************
  OPER is the module that allows the user to define its
  own criterion for operationality.
  The criterion implemented here is the disjunctive preference
  criterion.
 *****************************************************************/

/* ---------------------------------------------------------------
   assess_operationality(+Operational_proof, - Concept)
   Assess operationality at the second_tiered with the disjunctive
   preference criterion.
   ---------------------------------------------------------------*/
assess_operationality(Proof,Concept):-
        assess_operationality(Proof,[],Concept).

assess_operationality([],Concept,Concept):- !.
assess_operationality([H,W|T],[],Concept):-
        calcul_disjunctive_weight(H,0,Dw),!,
        assess_operationality(T,[H,Dw],Concept).
assess_operationality([H,W|T],[C1,Dw1],Concept):-
        calcul_disjunctive_weight(H,0,Dw),
        Dw > Dw1 ,!,
        assess_operationality(T,[H,Dw],Concept).
assess_operationality([H,W|T],Acc,Concept):-
        assess_operationality(T,Acc,Concept).


/* ---------------------------------------------------------------
   calcul_disjunctive_weight(+Clause,-Disj_weight) gives a weight of
   20 to conjunction (cd-disjunction) and a weight of 1 to disjunction
   (cd-conjunction).
   --------------------------------------------------------------- */
calcul_disjunctive_weight(A,Dw,Dw):-
        literal(A).
calcul_disjunctive_weight((A or B),Dw,Total):-
        literal(A),!,
        Dwtemp is Dw + 1,
        calcul_disjunctive_weight(B,Dwtemp,Total).
calcul_disjunctive_weight((A and B),Dw,Total):-
        literal(A),!,
        Dwtemp is Dw + 20,
        calcul_disjunctive_weight(B,Dwtemp,Total).
calcul_disjunctive_weight((A or B),Dw,Total):-
        literal(A),!,
        Dwtemp is Dw + 1,
        calcul_disjunctive_weight(B,Dwtemp,Total).
calcul_disjunctive_weight((A and B),Dw,Total):-
        calcul_disjunctive_weight(A,Dw,T1),
        calcul_disjunctive_weight(B,Dw,T2),!,
        Total is T1+T2+20.
calcul_disjunctive_weight((A or B),Dw,Total):-
        calcul_disjunctive_weight(A,Dw,T1),
        calcul_disjunctive_weight(B,Dw,T2),!,
        Total is T1+T2+1.
```

```
/* PROINTERFACE.PL */
/*************************************************************************
 LODIC (Learning of DIsjunctive Concepts)

 LODIC is a learning system design to allow the use of non-Horn clauses in
 the domain theory and in the acquisition of the final concept. Non-Horn clauses
 are the means by which uncertainty is introduced in domain application.

 For a complete explanation of the system, please consult:
       Maude Salembier-Pelletier, Learning of disjunctive concepts with
       Explanation-Based Learning, Master Thesis, University of Ottawa, 1990.

 Required files:
 ---------------
 weight.pl :    This file contains the weighting procedure that creates the
                weight database
 prover.pl :    This file contains the theorem prover that constructs the
                explanation required by EBG. The theorem prover is the
                inference engine of EBG.
 resolution.pl: This file contains the procedures to resolve a step of
                the proof.
 tools.pl   :   This file contains a collection of programming tools.
                -The record tools to record and retrieve entry in the database.
                -The logical tools that perform logical processing of terms.
 generalizer.pl : contains procedures to derived the final concept description.

 block.pl:      contains a demo of the block example.


 Inputs from the user:
 ---------------------
       LODIC prompts you for the name of the domain theory and also for the
       operationality criterion.
       To use the demo, enter block. To use your own domain theory, make your
       own file in the required format and enter the name. The domain theory
       of the block example is a good example.

 STARTING up:
 ------------
       To start up the program, enter the following command:
       ebg(+General_goal, +Specific_goal, -Specific_proof, -General_proof,
       -Final_concept).

 Notation convention:
 --------------------
       In a comment:
       +X : means an input variable,
       -X : means an output variable,
       ?X : means a temporary variable.

 *************************************************************************/

%%%%%%%% using external built-in predicates %%%%%%%%%%%%%%%%%%%%%%%%%
:- use_module(library(subsumes), [subsumes/2,variant/2]).
:- use_module(library(same_functor), [same_functor/2]).
:- use_module(library(basics),[append/3,member/2,memberchk/2]).
:- use_module(library(lists), [remove_dups/2,rev/2,nth0/3]).
:- use_module(library(change_arg),[change_arg/4]).
:- use_module(library(sets),[seteq/2]).

%%%%%%%% operators needed by the program %%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- op(1130, xfy, or).
:- op(1120, xfy, and).
:- op(1100, fy, no).
:- op(1150, xfy, =>).
```

```
%%%%%%%% files that construct the program %%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- ensure_loaded(proweight).
:- ensure_loaded(proprover).
:- ensure_loaded(proresolution).
:- ensure_loaded(tools).
:- ensure_loaded(generalizer).
:- ensure_loaded(tests).

/*************************************************************************/


/* ------------------------------------------------------------------------
   ebg(+General_goal,-General_proof,-Final_concept)
   explanation-based generalization program finds a specific explanation,
   generalizes it, and gives as an output the final concept derived from
   the initial concept in an operational form.
   ------------------------------------------------------------------------*/

ebg(Goal,Proof,Concept):-
        ask_file(File),
        consult(File),
        treatment_of_literal,
        write('All the literals and clauses have being weighted'), nl,
        write('Starting the proof now ...'), nl,
        proof((no Goal),Proof),
        write('The proof is done.'),nl,
        write_the_output(Proof),
        ask_operationality(Option_list),
        concept_acquisition(Proof,Option_list,Concept), nl,
        write('*** The final concept definition is :  '),
        write(Concept),
        clean_up.



/* ****************** Interface to the user ********************  */


ask_file(File) :-
        write('Please, give the name of the file that contains the example '),
        nl,
        write('in the form of clauses like "a and b => c or d" '), nl,
        write('File name: '),
        read(File),nl.

/* ------------------------------------------------------------------------
   ask_operationality(-Option_list) finds the operationality criterion
   to consider for the acquisition of the final concept.
   ------------------------------------------------------------------------ */

ask_operationality(Option_list):-
        write('Enter the operationality criterion option:'), nl,
        write('1. Classical EBG criterion'), nl,
        write('2. Level-based operationality'), nl,
        write('3. Two-tiered operationality'),nl,
        read(Option),
        treat_option(Option,Option_list).

/* ------------------------------------------------------------------------
   treat_option(+Option, -Option_list) builds a list with the information
   needed for the operationality criterion based on the option selected
   ------------------------------------------------------------------------ */

treat_option(1,[1]).
treat_option(2,[2,2]).
```

```
treat_option(3,[3,Filename]):-
        write('Enter the filename of the operationality assessment procedure'),
        nl,
        read(Filename).

/* ------------------------------------------------------------------
    Procedures for writing the output to the user.
   ------------------------------------------------------------------ */

write_the_output(Proof):-
        write('*** The general proof ***' ), nl,nl,
        rev(Proof, RevProof),
        write_proof(RevProof),
        write('            NIL'),nl,nl.


write_proof([]):-!.
write_proof([X,Y|T]):-
        write('     '),
        write(X),
        write(' -- '),
        write(Y), nl,
        write('              | '),nl,
        write_proof(T).
```

```
/* PROWEIGHT.PL */
/* ------------------------------------------------------------
   treatment_of_literal performs the loop to treat all the facts,
    rules, and the goal.
   ------------------------------------------------------------ */
treatment_of_literal :-
        attribute_weight, treatment_of_literal.
treatment_of_literal .

/* ------------------------------------------------------------
   attribute_weight(+Term) attributes a weight to all the literals
   of the input file.
   First case is the goal
   ------------------------------------------------------------ */
attribute_weight:-
        goal(X),
        record_z(w,weight(X,0)),
        retract(goal(X)).
/* ------------------------------------------------------------
   2nd case : it is a fact.
   Facts have a weight of 4.
   Facts are needed to build the fact base .
   ------------------------------------------------------------ */
attribute_weight:-
        fact(X), ! ,
        record_z(w,weight((X),4)),
        record_z(4,(X)),
        retract(fact(X)).

/* ------------------------------------------------------------
   3rd case : it is the goal concept.
   Literals of the goal concept have a weight of 1 and assert the
   rule with the lightest literal weight.
   ------------------------------------------------------------ */
attribute_weight:-
        goal_concept(X), !,
        retract(goal_concept(X)),
        is_clause(X,_,Rest),
        treat_goal_concept(Rest),
        transform_into_clausal(X,Clause),
        record_a(0,Clause).


/* ------------------------------------------------------------
   4th case : it is a rule of the domain theory.
   Literals have a weight of 2 (potentially-operational) or
   3 (partially instantiated from the fact-base).
   ------------------------------------------------------------ */
attribute_weight:-
        rule(X), !,
        transform_into_clausal(X,Clause),
        treat_rule(Clause),
        weigh_and_record_rule(Clause),
        retract(rule(X)).

/* ------------------------------------------------------------
   is_clause(+Rule,-Head,-Body) returns the head and body of a rule.
   ------------------------------------------------------------ */
is_clause((Body => Head),Head,Body) :- !.
is_clause(Head,Head,true).

/* ------------------------------------------------------------
   Treat_goal_concept(+Body_of_rule) attributes a weight of 1 to
   each of the literals that compose the goal concept.
   ------------------------------------------------------------ */
treat_goal_concept((Arg1 and Rest)) :-
        !,
        treat_goal_concept(Arg1),
        treat_goal_concept(Rest).
treat_goal_concept((no Arg)):-
        \+ recorded_(w,weight((Arg),_)),
         record_z(w,weight((Arg),1)).
treat_goal_concept(Arg) :-
        \+ recorded_(w,weight((Arg),_)),
        record_z(w,weight((Arg),1)).
treat_goal_concept(_):-!.

/* ------------------------------------------------------------
   Treat_rule(+Clause) gives a weight to the head of the rule and then
   to the rest of the literals. A weight of two is associated if
   the literal is not part of the fact base, otherwise it is 3.
   ------------------------------------------------------------ */
treat_rule((no Literal)):-
        pliteral(Literal),!,
        check_and_put_in(Literal).
treat_rule(Literal):-
        literal(Literal),!,
        check_and_put_in(Literal).
treat_rule((Exp1 or Exp2)):-
        treat_rule(Exp1),!,
        treat_rule(Exp2).

/* ------------------------------------------------------------
   transform_into_clausal(+Logic_exp, -Clausal) transforms a logic
   expression of the form (a and b => c) into a disjunctive clausal
   form (not a or not b or c).
   ------------------------------------------------------------ */
transform_into_clausal((Cond => Head),(Rcond or Head)):-
        transform_clausal(Cond,Rcond),!.
transform_into_clausal(X,_):-
        write('This formula is not well formed: '),
        write(X), nl.

/* ------------------------------------------------------------
   transform_clausal(+Logic_exp, -Clausal) transforms a logic
   expression of the form (a and b => c) into a disjunctive clausal
   form (not a or not b or c).
   ------------------------------------------------------------ */
transform_clausal((no Literal), Literal):-
        pliteral(Literal), !.
transform_clausal(Literal,((no Literal))):-
        pliteral(Literal),!.
transform_clausal((X and Y), (RX or RY)):-
        !,neg(X,RX),
        neg(Y,RY).
transform_clausal((X or Y), (RX or RY)):-
        !,neg(X,RX),
        neg(Y,RY).

/* ------------------------------------------------------------
   weigh_and_record_rule(+Clause) weigh a clause and record the weight
   and the rule.
   ------------------------------------------------------------ */
weigh_and_record_rule(Clause):-
        copy_term(Clause,Ctemp),
        lightest_weight(Ctemp,_,W),!,
        record_a(W,Clause).

/* ------------------------------------------------------------
   check_and_put_in(+Literal) checks if the literal is already
   part of database. If not it records it.
   ------------------------------------------------------------ */
```

```
check_and_put_in(Literal):-
        recorded_(w,(weight(Literal,_))),!.
check_and_put_in(Literal):-
        record_a(w,weight(Literal,2)).
```

```
/* PROPROVER.PL */
/****************** LODIC project *****************/
/***    theorem prover  for the propositional     ***/
/*****************************************************/

/* ------------------------------------------------------
     proof(+Goal,-Proof) returns the list Proof containing
   the center clauses of the graph resolution of Goal.The
   elements are alternatively a clause and its weight.
   ------------------------------------------------------ */

proof(Goal,Proof) :-
        lightest_weight(Goal,_,W),
        prove(Goal, [],[Goal,W], Proof).

/* ------------------------------------------------------
   prove(+Goal, +Memory, ?Acc, -Proof) :
   Goal is the goal to be resolved.
   Memory contains the list of literals that have been resolved
   so far.
   Acc is an accumulator for this current proof.
   Proof is the resulting list containing the last branch of
   the proof of Goal.
   1. Stopping condition, it is a fact.
   2. General step
   ------------------------------------------------------ */

prove(Goal, Memory,Liste, Liste):-
        prove_one_step(Goal,empty, fact,Memory,Memory),!.

prove(Goal, Old_memory, Acc, Liste) :-
        prove_one_step(Goal,[New_goal, Weight], State,Old_memory,New_memory),
        process(State,[New_goal,Weight],New_memory,Acc,Liste).

/* ------------------------------------------------------
   process(+State,+[New_goal,Weight], +Memory, -Liste)
   process the next step according to the state of the
   resolving clause that creates the new goal to solve.
   The Memory accumulates the literal already solved.
   The Liste will contain the resolvents of the proof.
   ------------------------------------------------------ */

process(horn,[New_goal,Weight],Memory,Acc,Liste):-
        prove(New_goal,Memory,[New_goal, Weight|Acc], Liste).

process(clause,[New_goal,Weight],Memory, Acc,Liste):-
        prove(New_goal, Memory, [New_goal, Weight|Acc], Liste).

process(fact,[New_goal,Weight],Memory, Acc,Liste):-
        prove(New_goal,Memory,[New_goal, Weight|Acc],Liste).

process(non_horn_found,[C,Weight],Old_memory,[Goal|Acc],Liste):-
        diamond_process(Goal, Old_memory,[Goal|Acc], New_goal,New_memory,Disj),
        resolve_and(New_goal,C,Old_memory,New_memory,Res),
        lightest_weight(Res,_,W),
        prove(Res,New_memory,[Res,W|[Disj,Weight|[Goal|Acc]]],Liste).

process(non_horn_found,_,_,[Goal|Acc],_):-
        recorded_(nonhorn,used((B or H),[L])),
        retract_(nonhorn,used((B or H),[L])),
        record_a(nonhorn,used((B or H),[Goal|L])),
        record_a(preuve,branch(_,[Goal|Acc],[],[])),
        !,fail.

process(non_horn_transf,[New_goal,Weight],Memory, Acc,Liste):-
        record_a(preuve,branch(_,Acc,[],[])),
        prove(New_goal, Memory,[New_goal,Weight|_],List1),!,
        append(List1, Acc, Liste).

process(non_horn_transf,[_,_],_,_,_):-
        recorded_(nonhorn,transf(B,H,P)),
        record_a(nonhorn,used((B or H),[P])),
        retract_(nonhorn,transf(B,H,P)),
        !,fail.


/* ------------------------------------------------------------------
   diamond_process(+Goal,+Memory,+Acc,-The_result_of_the_link,
                   -New_memory,-Disjunction_form) forms the diamond
    save the branch that is missing.
   ------------------------------------------------------------------ */

diamond_process(Goal,Old_memory,Acc,Conj,Old_memory,Disj):-
/*  find the last branch */
        recorded_(preuve, branch(First,List, [], [])),
        !,
/*  find the common part and the junction node */
        find_junction(Acc, List, Node1,[X,Y|R_list]),
/* find the list of the clauses that have to be linked */
        find_all_branch(Node1,[],Br_list),
/* put these parts together*/
        make_and([Goal,X| Br_list], Conj),
/* translate in DNF */
        trans_in_dnf(Conj,Disj),
        retract_(preuve, branch(First, List,[], [])),
        record_a(preuve, branch(X,[X,Y|R_list], [Node1], [Disj])).


/* ------------------------------------------------------------
   find_junction(+Acc,+Branch,-Junction_node,-Rest_of_the_branch)
   finds the junction and return the rest of the branch.
   ------------------------------------------------------------*/

find_junction(Acc,Branch,Node_1,R_branch):-
        append(_,[Node_1,N|Y],Acc),
        append(R_branch,[Node_1,N|Y],Branch),
        \+ number(Node_1).

/* ------------------------------------------------------------
   find_all_branch(+Junction_node,+Temporary,-List) finds all the previous
   non_horn branch that have the same junction node.
   ------------------------------------------------------------*/
find_all_branch(Jnode,_,[]):-
        \+ recorded_(preuve,branch(_,_,[Jnode],_)),!.
find_all_branch(Jnode,Temp,[]):-
        Temp \== [],
        recorded_(preuve,branch(X,_,[Jnode],_)),!,
        member(X,Temp).
find_all_branch(Jnode,Temp,[X|Res]):-
        recorded_(preuve,branch(X,_,[Jnode],_)),
        find_all_branch(Jnode,[X|Temp],Res),!.

/* ------------------------------------------------------------
   trans_in_dnf(+Conj, -Disj) transforms a conjunction into a
    disjunction.
   ------------------------------------------------------------*/
trans_in_dnf(X,X):- literal(X),!.
trans_in_dnf((no X),(no Y)):-
        trans_in_dnf(X,Y),!.
trans_in_dnf((X and Y), (no ((no X) or (no Y)))):-
        literal(X),
        literal(Y),!.
```

```
trans_in_dnf(((no X) and (no Y)),(no (X or Y))):-
        literal(X),
        literal(Y),!.
trans_in_dnf((X and Y), (no ((no X)or Res))):-
        literal(X),
        trans_in_dnf((no Y),Res),!.
trans_in_dnf(((no X) and Y),(no (X or Res))):-
        literal(X),
        trans_in_dnf((no Y),Res),!.
trans_in_dnf((X and Y),(no (R1 or R2))):-
        trans_in_dnf((no X),R1),
        trans_in_dnf((no Y),R2).


/* ------------------------------------------------------------------
   prove_one_step(+Goal,-[New_goal,Weight],?State,+Old_memory,-New_memory)
   makes one step of resolution for Goal, finding the next goal to resolve,
   updating the state and the list of literal that are solved (new_memory).
   ------------------------------------------------------------------ */
/* Last fact - stopping condition  */

prove_one_step((no A), empty,fact,Old_memory,New_memory):-
        literal(A),
        recorded_(w,weight(A,4)),
        resolve((no A),4,empty,fact,Old_memory,New_memory).

/* General step  */

prove_one_step((no A), [New,W],State,Old_memory,New_memory):-
        literal(A),
        recorded_(w,weight(A,W1)),
        resolve((no A),W1, New,State,Old_memory,New_memory),
        lightest_weight(New,_,W).

prove_one_step(A, [New,W],State,Old_memory,New_memory):-
        literal(A),
        recorded_(w,weight(A,W1)),
        resolve(A,W1,New,State,Old_memory,New_memory),
        lightest_weight(New,_,W).

prove_one_step(A, [New_goal, W], State,Old_memory,New_memory) :-
        lightest_weight(A,X,W1),
        resolve(X,W1,Res,State,Old_memory,New_memory),
        accord_res(Res,X,W1,A,New_goal,W).

accord_res(empty,X,_,A,R,W):-
        substract(X,A,R),
        lightest_weight(R, _,W),!.

accord_res(XCond,X,W1,A,(R or XCond),W):-
        !,
        substract(X,A,R),
        lightest_weight((R or XCond),_,W).
```

```
/* ------------------------------------------------------------------
   PRORESOLUTION.PL contains the set of literals which are used to resolve
   a literal.
   ------------------------------------------------------------------ */

/* ------------------------------------------------------------
   facts
   ------------------------------------------------------------ */

resolve((no P), 4,empty, fact, M,M):-
        recorded_(4,P).
resolve(P,4,empty,fact,M,M):-
        recorded_(4,(no P)).

/* ------------------------------------------------------------
   resolution of no P
   ------------------------------------------------------------ */

resolve((no P),W, B, horn, Old_memory,[(no P)|Old_memory]):-
        find_horn(W,P,B),
        empty_inter(B, Old_memory).
resolve((no P), W,(B or Res), Param,Old_memory,[(no P)|Old_memory]):-
        setof(B1,find_nh(W,P,B1),L),
        noempty(L),
        member([(B or H),R],L),
        state_nh(P, B, H, R,Param, Res).
resolve((no P), W, _, clause,Old_memory,[(no P)|Old_memory]):-
        find_clause(W,P,[H,B]),
        empty_inter((H or B),Old_memory).
resolve(P, W, C, clause,Old_memory,[P|Old_memory]):-
        find_clause(W,(no P),C),
        empty_inter(C,Old_memory).
resolve_and(Clause,(B or H),Res):-
        liste_literals(Clause,L),
        substractn(L,(B or H),Res).

/* ------------------------------------------------------
   find_horn(+Weight,+Predicate,[-Resulting_clause])
   finds a valid Horn clause that contains the literal P and
   returns the Horn clause and the resulting clause - P.
 ------------------------------------------------------*/

find_horn(W,P,B):-
        recorded_(W, (B or P)),
        horn_clause(B, P).

/* ------------------------------------------------------
   find_nh(+Weight,+Predicate,[-Non-horn_clause,-Resulting_clause])
   finds a valid non-horn clause that contains the literal P and
   returns the non_horn clause and the resulting clause - P.
 ------------------------------------------------------*/

find_nh(W,P,[(B or H),R]):-
        recorded_(W,(B or H)),
        non_horn_clause(B,H),
        in(P,H,R).

/* ------------------------------------------------------
   find_clause(+Weight,+Predicate,-Resulting_clause) finds a
   clause of weight W that contains the Predicate.
 ------------------------------------------------------*/

find_clause(W,P,R):-
        recorded_(W, C),
        in(P,C,R).

/* ------------------------------------------------------
   empty_inter(+Exp, +Liste) succeeds if the intersection
   between the expression and the list is empty.
 ------------------------------------------------------*/

empty_inter(_,[]):-!.
empty_inter(Exp,Liste):-
        !,
        common(Exp,Liste,R),
        R == [].
empty_inter(_,_):-!,fail.

/* ------------------------------------------------------
   common(+Exp,+Liste,-Common_liste) accepts an expression
   and a liste and returns in a liste the common elements in
   the expression and the liste.
 ------------------------------------------------------*/

common((H),Liste,[H]):-
        literal(H),
        my_member(H,Liste),!.
common((H),_,[]):-
        literal(H),!.
common((H or Body),Liste,[H|Res]):-
        literal(H),
        my_member(H,Liste),!,
        common(Body,Liste,Res).
common((_ or Body),Liste,Res):-
        !,common(Body,Liste,Res).

/* ------------------------------------------------------
   state_nh(+Predicate,+Left_handside,+Right_handside,+Rest,
         -Non_horn_state, -Marked_literals)
   finds the state of a non-horn clause.
   1. If this non-horn clause has already being used then one
      more part has been found => state= non_horn_found
   2. If this non-horn clause has already being used then it is
      not a transformation => fail (occurs when backtracking).
   3. If it is the first time this non-horn is used then
      marked the other literals of the disjuncts and record
      the non-horn clause in the database.
      State = non_horn_transform.
   ------------------------------------------------------*/

/* when finding the last missing literal to be resolved with the nh clause
   the result (last argument) is the head of thenh clause*/

state_nh(P,L,H,_,non_horn_found,H):-
        recorded_(nonhorn,used((L or H),_)),!.
state_nh(_,L,H,_,non_horn_transf,_):-
        recorded_(nonhorn, used((L or H),_)),!,fail.
state_nh(P,L,H,R,non_horn_transf,R):-
        !, record_a(nonhorn,transf(L,H,P)).
```

Appendix B

Typical User Sessions

```
| ?- [interface].
[interface.pl consulted 37.367 sec 74,940 bytes]

yes
| ?- ebg(safe_to_stack(Obj1,Obj2),safe_to_stack(obj1,obj2),SProof,GProof,General

Please, give the name of the file that contains the example
in the form of clauses like "a and b => c or d"
File name: sts_dom.

[consulting /usr1/grad/maude/these/sts_dom.pl...]
[sts_dom.pl consulted 0.383 sec 1,708 bytes]

All the literals and clauses have being weighted
Starting the proof now ...
The proof is done.
*** The specific proof ***

     0 -- no safe_to_stack(obj1,obj2)
          |
     1 -- no lighter(obj1,obj2)
          |
     2 -- no weight(obj1,0.1)or no weight(obj2,5)or no less(0.1,5)
          |
     2 -- (no weight(obj2,5)or no less(0.1,5))or no compute_weight(obj1,0.1)
          |
     2 -- (no less(0.1,5)or no compute_weight(obj1,0.1))or no isa(obj2,endtable)
          |
     3 -- (no less(0.1,5)or no isa(obj2,endtable))or no volume(obj1,1)or
no density(obj1,0.1)or no mult(0.1,1,0.1)
          |
     4 -- no isa(obj2,endtable)or no volume(obj1,1)or no density(obj1,0.1)or
no mult(0.1,1,0.1)
          |
     4 -- no volume(obj1,1)or no density(obj1,0.1)or no mult(0.1,1,0.1)
          |
     4 -- no density(obj1,0.1)or no mult(0.1,1,0.1)
          |
     4 -- no mult(0.1,1,0.1)
          |
         NIL

*** The general proof  ***

     0 -- no safe_to_stack(_466,_483)
          |
     1 -- no lighter(_466,_483)
          |
     2 -- no weight(_466,_11108)or no weight(_483,5)or no less(_11108,5)
          |
     2 -- (no weight(_483,5)or no less(_11108,5))or
no compute_weight(_466,_11108)
          |
     2 -- (no less(_11108,5)or no compute_weight(_466,_11108))or
no isa(_483,endtable)
          |
     3 -- (no less(_11108,5)or no isa(_483,endtable))or no volume(_466,_13718)
or no density(_466,_13726)or no mult(_13726,_13718,_11108)
          |
     4 -- no isa(_483,endtable)or no volume(_466,_13718)or
no density(_466,_13726)or no mult(_13726,_13718,_11108)
          |
     4 -- no volume(_466,_13718)or no density(_466,_13726)or
no mult(_13726,_13718,_11108)
          |
     4 -- no density(_466,_13726)or no mult(_13726,_13718,_11108)
          |
     4 -- no mult(_13726,_13718,_11108)
          |
         NIL
Enter the operationality criterion option:
1. Classical EBG criterion
2. Level-based operationality
3. Two-tiered operationality
|: 1.

*** The final concept definition is :
density(A,B) and isa(C,endtable) and less(D,5) and volume(A,E) and mult(B,E,D)

| ?- ebg(is_climber(A),is_climber(mike),SProof,GProof,Generalization).

Please, give the name of the file that contains the example
in the form of clauses like "a and b => c or d"
File name: club.

[consulting /usr1/grad/maude/these/club.pl...]
[club.pl consulted 0.316 sec 1,376 bytes]

All the literals and clauses have being weighted
Starting the proof now ...
The proof is done.
*** The specific proof ***

     0 -- no is_climber(mike)
          |
     1 -- no clubmember(mike)or no climber(mike)or skier(mike)
          |
     1 -- (no clubmember(mike)or skier(mike))or no clubmember(mike)or
skier(mike)
          |
     1 -- (no clubmember(mike)or no clubmember(mike)or skier(mike))or
likes(mike,snow)
          |
     3 -- (no clubmember(mike)or no clubmember(mike)or likes(mike,snow))or
likes(mike,snow)
          |
     3 -- (no clubmember(mike)or no clubmember(mike)or likes(mike,snow))or
no likes(tony,snow)
          |
     4 -- (no clubmember(mike)or no clubmember(mike)or no likes(tony,snow))or
 no likes(tony,snow)
          |
     4 -- no clubmember(mike)or no likes(tony,snow)or no likes(tony,snow)
          |
     4 -- no likes(tony,snow)or no likes(tony,snow)
          |
     4 -- no likes(tony,snow)
          |
         NIL

*** The general proof  ***

     0 -- no is_climber(_466,
          |
     1 -- no clubmember(_7121)or no climber(_7121)or skier(_7121)
          |
     1 -- (no clubmember(_7121)or skier(_7121))or no clubmember(_7121)or
skier(_7121)
          |
     1 -- (no clubmember(_7121)or no clubmember(_7121)or skier(_7121))or
```

```
likes(_10081,snow)
          |
     3 -- (no clubmember(_7121)or no clubmember(_7121)or likes(_10081,snow))or l
ikes(_11654,snow)
          |
     3 -- (no clubmember(_7121)or no clubmember(_7121)or likes(_11654,snow))or
 no likes(tony,snow)
          |
     4 -- (no clubmember(_7121)or no clubmember(_7121)or no likes(tony,snow))or
no likes(tony,snow)
          |
     4 -- no clubmember(_7121)or no likes(tony,snow)or no likes(tony,snow)
          |
     4 -- no likes(tony,snow)or no likes(tony,snow)
          |
     4 -- no likes(tony,snow)
          |
         NIL
Enter the operationality criterion option:
1. Classical EBG criterion
2. Level-based operationality
3. Two-tiered operationality
|: 2.
Enter the specific level, 2 or 3:
|: 3.
*** The final concept definition is :
clubmember(A)and likes(tony,snow)and no likes(B,snow)


| ?- ebg(carnivore(A),carnivore(whale_1),SProof,GProof,Generalization).

Please, give the name of the file that contains the example
in the form of clauses like "a and b => c or d"
File name: carnivore.

[consulting /usr1/grad/maude/these/carnivore.pl...]
[carnivore.pl consulted 0.317 sec 1,396 bytes]

All the literals and clauses have being weighted
Starting the proof now ...
The proof is done.
*** The specific proof ***

     0 -- no carnivore(whale_1)
          |
     1 -- no eat(whale_1,meat)or no animal(whale_1)
          |
     2 -- no eat(whale_1,meat)or no mammal(whale_1)
          |
     2 -- no mammal(whale_1)or no eat(whale_1,fish_1)or no animal(fish_1)
          |
     2 -- (no mammal(whale_1)or no eat(whale_1,fish_1))or no fish(fish_1)
          |
     3 -- (no mammal(whale_1)or no eat(whale_1,fish_1))or no vertebrate(fish_1)
or no lives_in(fish_1,water)
          |
     4 -- no eat(whale_1,fish_1)or no vertebrate(fish_1)or
no lives_in(fish_1,water)
          |
     4 -- no vertebrate(fish_1)or no lives_in(fish_1,water)
          |
     4 -- no lives_in(fish_1,water)
          |
         NIL

*** The general proof  ***

     0 -- no carnivore(_466)
          |
     1 -- no eat(_466,meat)or no animal(_466)
          |
     2 -- no eat(_466,meat)or no mammal(_466)
          |
     2 -- no mammal(_466)or no eat(_466,_8434)or no animal(_8434)
          |
     2 -- (no mammal(_466)or no eat(_466,_8434))or no fish(_8434)
          |
     3 -- (no mammal(_466)or no eat(_466,_8434))or no vertebrate(_8434)or
no lives_in(_8434,water)
          |
     4 -- no eat(_466,_8434)or no vertebrate(_8434)or no lives_in(_8434,water)
          |
     4 -- no vertebrate(_8434)or no lives_in(_8434,water)
          |
     4 -- no lives_in(_8434,water)
          |
         NIL
Enter the operationality criterion option:
1. Classical EBG criterion
2. Level-based operationality
3. Two-tiered operationality
|: 2.
Enter the specific level, 2 or 3:
|: 2.
*** The final concept definition is :fish(A)and mammal(B)and eat(B,A)
| ?-
| ?- ebg(put_max(A,B,C),put_max(1,a,b),SProof,GProof,Generalization).

Please, give the name of the file that contains the example
in the form of clauses like "a and b => c or d"
File name: block.

[consulting /usr1/grad/maude/these/block.pl...]
[block.pl consulted 0.450 sec 1,224 bytes]

All the literals and clauses have being weighted
Starting the proof now ...
The proof is done.
*** The specific proof ***

     0 -- no put_max(1,a,b)
          |
     1 -- no holds(val(1,a),assign(1,a,_6193))or no max(a,b,a)
          |
     1 -- no max(a,b,a)or no location(1)
          |
     2 -- no location(1)or no is_smaller(a,b)
          |
     2 -- (no location(1)or no is_smaller(a,b))and (no location(1)or
no is_smaller(b,a))
          |
     4 -- no location(1)or no numb(a)or no numb(b)or no location(1)or
no location(1)
          |
     4 -- no numb(a)or no numb(b)or no location(1)or no location(1)
          |
     4 -- no numb(b)or no location(1)or no location(1)
          |
     4 -- no location(1)or no location(1)
          |
     4 -- no location(1)
          |
```

```
          NIL

*** The general proof  ***

     0 -- no put_max(_467,_482,_497)
          |
     1 -- no holds(val(_467,_482),assign(_467,_482,_6169))or no max(_482,_497,_4
          |
     1 -- no max(_482,_497,_482)or no location(_467)
          |
     2 -- no location(_467)or no is_smaller(_482,_497)
          |
     2 -- (no location(_467)or no is_smaller(_482,_497))and (no location(_9322)
or no is_smaller(_497,_482))
          |
     4 -- no location(_467)or no numb(_482)or no numb(_497)or no location(_467)
or no location(_9322)
          |
     4 -- no numb(_482)or no numb(_497)or no location(_467)or no location(_9322)
          |
     4 -- no numb(_497)or no location(_467)or no location(_9322)
          |
     4 -- no location(_467)or no location(_9322)
          |
     4 -- no location(_9322)
          |
          NIL
Enter the operationality criterion option:
1. Classical EBG criterion
2. Level-based operationality
3. Two-tiered operationality
|: 3.
Enter the filename of the operationality assessment procedure
|: oper.
[consulting /usr1/grad/maude/these/oper.pl...]
[oper.pl consulted 0.567 sec 1,756 bytes]

*** The final concept definition is :
location(A)and is_smaller(B,C)or location(D)and is_smaller(C,B)
```

```
(ai3)maude(1) prolog

Quintus Prolog Release 2.5.1 (Sun-3, SunOS 4.1)
Copyright (C) 1990, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- [prointerface].
[prointerface.pl consulted 30.650 sec 71,256 bytes]

yes
| ?- ebg(poisonous,Proof,Concept).

Please, give the name of the file that contains the example
in the form of clauses like "a and b => c or d"
File name: mushroom.

[consulting /usr1/grad/maude/these/mushroom.pl...]
[mushroom.pl consulted 0.167 sec 736 bytes]

All the literals and clauses have being weighted
Starting the proof now ...
The proof is done.

*** The general proof ***

     0 -- no poisonous
            |
     1 -- no toadstool
            |
     2 -- no fungus or mushroom
            |
     2 -- mushroom or no boletus
            |
     4 -- no boletus or no boletus
            |
     4 -- no boletus
            |
           NIL

Enter the operationality criterion option:
1. Classical EBG criterion
2. Level-based operationality
3. Two-tiered operationality
|: 2.

*** The final concept definition is :  no mushroom and boletus
| ?-
| ?- ebg(treatment,P,C).

Please, give the name of the file that contains the example
in the form of clauses like "a and b => c or d"
File name: diseases.

[consulting /usr1/grad/maude/these/diseases.pl...]
[diseases.pl consulted 0.250 sec 1,120 bytes]

All the literals and clauses have being weighted
Starting the proof now ...
The proof is done.
*** The general proof ***

     0 -- no treatment
            |
     1 -- no diagnostic_of_mctd
            |
     2 -- no tests
            |
     2 -- no sle
            |
     2 -- no sle and no mctd
            |
     4 -- no arthritis or no rash or no fever
            |
     4 -- no rash or no fever
            |
     4 -- no fever
            |
           NIL

Enter the operationality criterion option:
1. Classical EBG criterion
2. Level-based operationality
3. Two-tiered operationality
|: 3.
Enter the filename of the operationality assessment procedure
|: oper.
[oper.pl consulted 0.533 sec 1,772 bytes]

*** The final concept definition is :  sle or mctd
```