Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

L.C. van der Gaag

PROLOG: an expert system building tool

Computer Science/Department of Software Technology Report CS-R8616 April

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

PROLOG: an Expert System Building Tool

L.C. van der Gaag
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

For several years, Lisp has been the most popular programming language for artificial intelligence. PROLOG, however, is rapidly becoming the second most popular AI language; for several applications, PROLOG is even preferred to Lisp. In this paper, the suitability of PROLOG as an expert system building tool is demonstrated: a small expert system shell is discussed, and compared to the DELFI-2 system, after the example of which the PROLOG system has been developed.

1980 Mathematics Subject Classification: 69K11, 69D42.
1982 CR Categories: I.2.1, D.3.2.
Key Words & Phrases: expert systems, PROLOG, logic programming.

1. Introduction

Nowadays, several tools for building expert systems are in use:

- high-level programming languages: Lisp, Pascal
- expert system shells: EMYCIN [1], DELFI-2 [2]
- representation languages: PROLOG, OPS5 [3]

Early expert systems were constructed in a high-level programming language. The use of a high-level programming language has some severe drawbacks. Disproportianate attention has to be paid to implementation issues not relevant to the, often complex, expert domain. Furthermore, the domain-dependent expert knowledge and the algorithms for manipulating the knowledge are interwoven; once constructed, the expert system cannot easily be adapted to altered views on the domain.

Expert knowledge, however, is continuously subject to changes due to altering views and new experiences. This view has led to the introduction of the so-called expert system shells [4]. An expert system shell offers an expert system building environment having the facilities for representing and manipulating domain-dependent expert knowledge; the expert just has to provide the knowledge. The principle of expert system shells is reflected in what is sometimes called, the paradigm of expert system design:

$$\text{expert system} = \text{knowledge} + \text{inference}$$

An expert system constructed using an expert system shell, typically comprises two components:

- a knowledge base, containing domain-dependent expert knowledge
- an inference engine, containing domain-independent algorithms for manipulating the expert knowledge

The knowledge base and inference engine are strictly separated. A knowledge base may be replaced by another knowledge base, thus obtaining a different expert system. The algorithms for manipulating an existing knowledge base may be replaced by algorithms for manipulating the same knowledge base, using a different reasoning strategy.

The inference engine is part of a consultation program that furthermore provides a user interface and facilities for explaining the expert system's lines of reasoning, called explanation facilities. Figure 1 depicts the typical architecture of an expert system.

Report CS-R8616
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

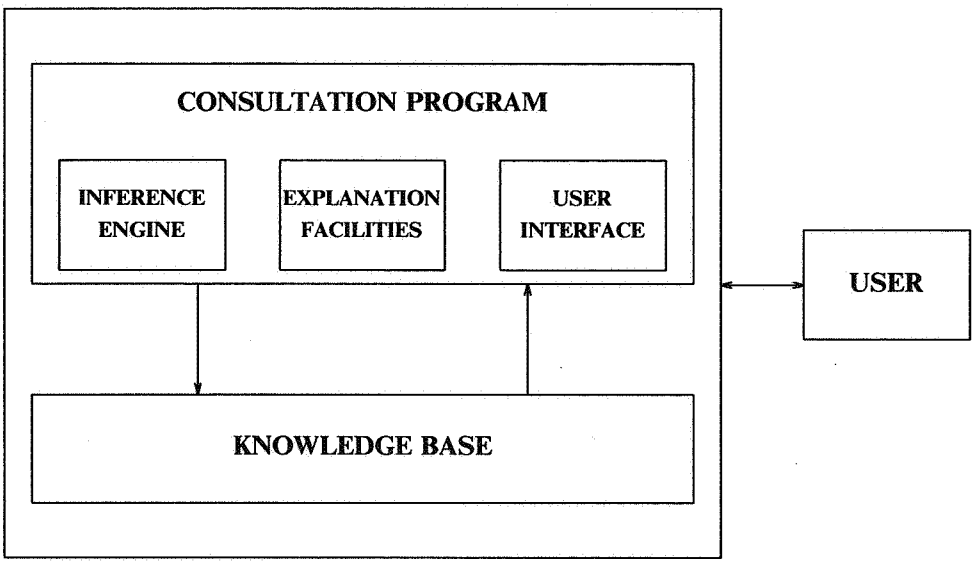

Figure 1. Typical architecture of an expert system

Although the construction of an expert system in a high-level programming language is not to be recommended, these languages are useful for implementing an inference engine. For example, the inference engine of the EMYCIN system is implemented in Lisp; Pascal is used for the implementation of the DELFI-2 system. The knowledge base is constructed using a formal specification language, especially designed for knowledge engineering.

As for expert system design, the representation languages lack the drawbacks of high-level programming languages. For instance, the separation of knowledge and inference is achieved rather naturally using PROLOG for the implementation of an expert system. In this paper, a small expert system shell in PROLOG is discussed in order to illustrate the suitability of a representation language as an expert system building tool.

2. Issues in expert system design

The main issues in expert system design are knowledge representation and knowledge manipulation. Several techniques for representing expert knowledge in a knowledge base are in use. An overview of these techniques is presented in [5]. Three methods are employed frequently:

- semantic nets [6]
- frames [7]
- production rules [8]

Most present-day expert systems use production rules for knowledge representation. This representation technique is discussed in section 2.1.

As for knowledge manipulation in a production rule based expert system, one of two basic techniques is applied in the construction of an inference engine [9]:

- top down inference
- bottom up inference

Diagnostic expert systems, expert systems for diagnosing an observed malfunctioning of a system, often apply top down inference as a knowledge manipulation scheme. This knowledge manipulation scheme is discussed in section 2.2.

2.1 Production rules

The production rule formalism is used to encode expert knowledge in conditional statements:

if
  certain conditions are fulfilled,
then
  certain conclusions are drawn

Conditions and conclusions are statements concerning objects and their properties. There are several ways to express conditions and conclusions, and their interdependence more formally. In this paper, the formalism described below is adopted:

if <condition part> **then** <conclusion part>, where

<condition part> ::= <clause> {**and** <clause>}*
<clause> ::= <condition> {**or** <condition>}*
<conclusion> ::= <conclusion> {**and** <conclusion>}*

<condition> ::= <predicate> <object> <attribute> <value>
<conclusion> ::= <action> <object> <attribute> <value>

Conditions and conclusions are composed of four elements:

- a three-place predicate or an action having three arguments
- an object, being the subject the statement refers to
- an attribute, being the property under discussion
- a constant value

In a condition, the predicate compares the value the specified attribute has adopted, with the specified constant value. For example, the condition

lessthan patient age 50

contains the predicate "lessthan". The statement refers to a patient and the property being discussed is his or her age. The predicate "lessthan" succeeds is the patient under consideration is younger than 50 years.

In a conclusion, the action operates on its arguments. The action "conclude", for instance, assigns the specified constant value to the attribute of the object. In

conclude patient diagnosis hepatitis_A

the constant value "hepatitis_A" is assigned to the attribute "diagnosis" of the patient in contemplation. This conclusion states that this patient's disease is diagnosed as "hepatitis_A".

Production rules have proved to be a suitable representation scheme for encoding expert knowledge in a diagnostic expert system. There is, however, a serious complication: in a real-life environment the expert knowledge is often incomplete and interspersed with uncertainties. The production rule formalism is therefore extended in order to provide the expert with a means for expressing the uncertainties of his beliefs. There are several schemes for modelling and dealing with uncertainties:

- theory of probability
- Dempster Shafer theory [10]
- fuzzy sets and fuzzy logic [11]
- certainty factor theory [12]

In this paper, the discussion will be restricted to the certainty factor theory. The certainty factor theory is simple and to the purpose. Although the theory lacks a mathematical foundation, it is employed in several well-known expert system shells.

A certainty factor (cf) attached to a statement is a number between -1 and +1 that reflects the

measure of the expert's belief in the statement. Positive certainty factors indicate there is evidence that the statement is correct; the larger the certainty factor, the greater is the belief in the statement. When the certainty factor equals +1, the statement is known to be correct. A negative certainty factor suggests that the statement is incorrect. The smaller the certainty factor, the greater is the belief that the statement is false. When the certainty factor equals -1, the statement is known to be incorrect. The certainty factor cf = 0 indicates the statement is neither confirmed nor discomfirmed.

In the production rule formalism, certainty factors are attached to the conclusions. A typical production rule in a medical expert system concerning diseases of the liver and biliary tract is, for instance:

if
same patient sex female **and**
greaterthan patient age 50 **or**
lessthan patient age 12 **and**
same patient signs spider_angiomas **or**
same patient signs palmar_erythema **or**
same patient signs Kayser_Fleischer_rings **or**
same patient signs butterfly_erythema
then
conclude patient cholestasis intrahepatic 0.40 **and**
conclude patient cholestasis extrahepatic 0.20

This rule contains conditions concerning a patient. The properties considered are the patient's sex, age and clinical signs. The rule contains two conclusions in relation to the patient's cholestasis. If the conditions are fulfilled there is some evidence (a certainty factor 0.40) that the patient's cholestasis is intrahepatic, and there is less evidence (a certainty factor 0.20) that the cholestasis is extrahepatic.

2.2. Top down inference

As mentioned before, top down inference is a knowledge manipulation scheme suitable for application in diagnostic expert systems. Top down inference is a goal directed reasoning strategy. Applying goal directed reasoning in an inference engine, the expert system starts with a statement of the goal to achieve, for instance the diagnosis of the disease of a patient in a medical expert system, and through subgoals finally reaches data, necessary to diagnose the disease.

The goal of the consultation of a production rule based expert system is to establish values for certain attributes of an object. These attributes are called goal attributes. Top down inference takes a goal attribute and searches the knowledge base for production rules concluding on this attribute. The evaluation of the selected rules starts with the evaluation of the conditions of the rules. The attribute of a condition becomes the next goal attribute. In top down inference the order of the evaluation of the selected rules and the order of the evaluation of the conditions are not predetermined. For an implementation of top down inference in an inference engine, however, the rules and conditions have to be evaluated in a fixed order. Backward chaining is an implementation of top down inference where the rules and conditions are evaluated sequentially.

During the consultation of an expert system values and corresponding certainty factors will be assigned to some attributes: these attributes are traced. An attribute with its associated value and certainty factor is called a fact. Conform the production rule formalism a fact has the following structure:

<fact> ::= <object> <attribute> <value> <cf>

The evaluation of a condition starts with the inspection of the facts established in the course of the consultation. When facts are known concerning the object and attribute specified in the condition, the production rules need not be considered, because the attribute is traced already. Otherwise, the production rules are used to trace the attribute. When the attribute cannot be traced by applying the

rules in the knowledge base, the user is asked to supply additional information.

When each condition of a production rule is evaluated to be successful, the actions of the conclusions of the rule are performed. The action "conclude", for instance, establishes a new fact, by assigning a constant value to an attribute. With this fact a combined certainty factor is associated.

A certainty factor cf_{fact} is computed using

$$cf_{fact} = cf_{conclusion} \cdot cf_{conditions}$$

where $cf_{conclusion}$ is the certainty factor specified in the conclusion stating the fact. The certainty factors resulting from the evaluation of the conditions are combined into $cf_{conditions}$ using

$$cf_{c_1 \text{ and } c_2} = \min \{cf_{c_1}, cf_{c_2}\}$$

$$cf_{c_1 \text{ or } c_2} = \max \{cf_{c_1}, cf_{c_2}\}$$

where cf_{c_i} is the certainty factor resulting from the evaluation of condition c_i, i = 1,2. This certainty factor cf_{fact} is associated with the established fact, if it is the first stating the specified attribute value.

If a fact stating the same attribute value has been established before, the certainty factor of this old fact and the certainty factor of the new fact are combined into a new certainty factor using

$$cf_{fact} = f(cf_{old}, cf_{new}), \text{ where}$$

$$f(cf_{old}, cf_{new}) = \begin{cases} cf_{old} + cf_{new} \cdot (1 - cf_{old}) & \text{if } cf_{old} \geqslant 0 \text{ and } cf_{new} \geqslant 0 \\ \dfrac{cf_{old} + cf_{new}}{1 - \min\{|cf_{old}|, |cf_{new}|\}} & \text{if } cf_{old} \cdot cf_{new} < 0 \\ -f(-cf_{old}, -cf_{new}) & \text{if } cf_{old} < 0 \text{ and } cf_{new} < 0 \end{cases}$$

Top down inference is discussed in more detail in [13].

3. PROLOG

Before focussing the attention on the application of PROLOG in expert system design, the language and its principles are briefly discussed.

3.1. Introduction

The programming language PROLOG is based on the principles of the first-order predicate logic. The foundation of the language was laid about 1970 by Alain Colmerauer at Marseille University. Yet, the development of the first PROLOG system took four years. The system resulted from the combined efforts of Colmerauer and Robert Kowalski, both engaged in research on logic programming, and researchers on the field of automated theorem proving. The name PROLOG is an abbreviation of *PROgramming in LOGic* as a reference to its principles.

In the early seventies, PROLOG drew little attention in the commercial and industrial world. In several universities, however, PROLOG was employed in a variety of projects. With the development of an efficient compiler for DEC 10/20 PROLOG by David Warren, Fernando Pereira and Lawrence Byrd at Edinburgh University, the programming language emerged as a practical choice for writing programs that involve sophisticated symbolic computation.

Nowadays, PROLOG is the centre of interest. Current areas of PROLOG applications involve [14]:

- natural language processing
- compiler writing [15]
- mathematical logic and theorem proving
- database design [16,17]
- expert system design [18,19]

3.2. Logic programming

A program written in a procedural programming language like Pascal is a specification of a series of steps to be undertaken successively in order to arrive at a solution of the considered problem. The description of the problem is embedded in this specification. In logic programming, the description of the problem and the methods to solve the problem are separated explicitly. Robert Kowalski [9] formulated this separation in the paradigm of logic programming:

algorithm = logic + control

An algorithm is usefully split up into two components: a logical component defining *what* the algorithm solves, and a control component describing *how* the solution is achieved.

In the logical component, the problem is expressed in facts and rules in regard to objects and relations between objects, relevant to the problem. The logical component can thus be viewed as a set of propositions. These propositions are expressed in the clausal form of logic. The control component contains a theorem proving algorithm.

The major discovery of logic programming is that the clauses themselves can provide the basis of the procedures required to compute relations expressed in the clausal form of logic.

3.3. PROLOG

The programming language PROLOG is a practical tool for logic programming. A PROLOG system comprises two parts: a PROLOG database and a PROLOG interpreter. The PROLOG database gives shape to the logical component of an algorithm: the database contains the propositions of the logical component expressed in the Horn clause formalism discussed in section 3.3.1. In a PROLOG system the control component of an algorithm is provided for in the PROLOG interpreter briefly reviewed in section 3.3.2. The PROLOG interpreter is based on a theorem proving algorithm called unification [20]. Figure 2 reflects in summary as to how the paradigm of logic programming and PROLOG are interrelated.

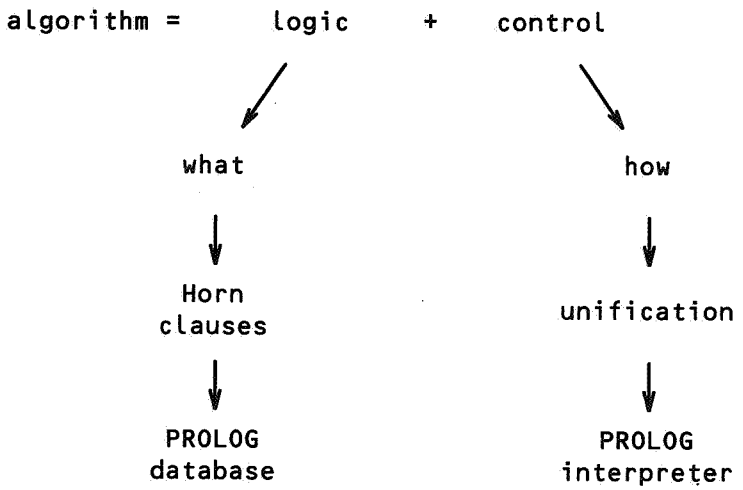

Figure 2. The relationship between PROLOG and logic programming

A reader interested in the principles of logic programming, is referred to [9]. An introduction to the programming language PROLOG is given in [21,22]; the more experienced PROLOG programmer may extend his knowledge of PROLOG with [23,24].

3.3.1. The Horn clause formalism

In PROLOG, knowledge is represented in the Horn clause formalism. A Horn clause is an expression of the form

$$B :- A_1,...,A_n$$

where $B, A_1,...,A_n$ are atomic formulae, $n \geq 0$. An atomic formula is an expression of the form

$$P(t_1,...,t_m)$$

where P is an m-place predicate symbol, $m \geq 0$, and $t_1,...,t_m$ are terms. A term is a constant symbol, a variable or a function of terms.

The atomic formulae $A_1,...,A_n$ are the conditions of the clause, and B is the conclusion. The interpretation of a Horn clause is

$$"B \text{ (is true) is } A_1 \text{ and ... and } A_n \text{ (are true)}"$$

If there are no conditions attached to the conclusion, then the conclusion is interpreted as stating the fact

$$"B \text{ (is true)}"$$

A set of Horn clauses, the logical component of an algorithm, constitutes the PROLOG database.

A PROLOG directive is a Horn clause where the conclusion is missing. In a directive the symbol :- is replaced by the symbol ?-:

$$?- A_1,...,A_n.$$

The Horn clause formalism can be looked upon as a programming language. A program is a set of Horn clauses, where each clause is a procedure. In the clause

$$B :- A_1,....,A_n.$$

B is the procedure heading, where the procedure name is determined by the predicate of B. $A_1,....,A_n$ is the body of the procedure, consisting of the procedure calls A_i, $i=1,...,n$. The main program is a PROLOG directive.

3.3.2. The PROLOG interpreter

A PROLOG directive is executed by executing its procedure calls. A procedure call is executed by applying a powerful and general parameter-passing mechanism, implemented by a term matching operation called unification. Both the procedure call and the procedure headings of the procedures in the database are treated as terms. A call and a procedure heading, two terms, match if their predicates are spelt the same way and is their corresponding formal and actual arguments are unifiable. If in a pair of corresponding arguments both formal and actual argument describe variables, the unification binds them together to represent the same object. If in a pair of corresponding arguments the formal argument is a variable, it is instantiated to the actual argument, and vice versa. If both formal and actual argument describe a constant, the arguments match if they are spelt the same way.

When executing a procedure call, the PROLOG interpreter searches through the PROLOG database for the first procedure heading matching the call. Then, the procedure calls of the body of the matching clause are executed. When unification fails, its effects are undone: all variables which were instantiated by the attempt at unification are restored to their original unbound state. The procedure call is then matched against the procedure heading of the next clause. If a procedure call matches none of the procedure headings, backtracking is initiated. The succeeded procedure calls are looked at, one by one, in reverse order of activation, undoing the effects of unification, until a procedure call is met for which an alternative match can be found. The PROLOG interpreter then proceeds.

4. A PROLOG EXPERT SYSTEM SHELL

The paradigm of logic programming bears strong resemblance to the paradigm of expert system design. Both paradigms describe a separation of what and how. From this point of view the knowledge in an expert system corresponds with the logic of an algorithm, and the inference corresponds with the control. Moreover, on the level of implementation, the PROLOG database is similar to the knowledge base and the PROLOG interpreter analogizes the inference engine. Whether PROLOG is suitable as an expert system building tool therefore depends on two issues:

- the flexibility in representing expert knowledge in the PROLOG database, provided by the PROLOG system
- to what extend the PROLOG interpreter can serve as an inference engine

In this paragraph, these two issues are dealt with in the light of a PROLOG expert system shell, currently being developed at the Centre for Mathematics and Computer Science.

4.1. Knowledge representation

When the Horn clause formalism is compared with the production rule formalism a near resemblance is noticeable. Conditions and conclusions are easily expressed in atomic formulae. For instance, the condition

greaterthan patient age 50

consisting of the three-place predicate "greaterthan", the object "patient", the attribute "age" and the constant value "50", is represented as an atomic formula in

```
greaterthan(patient,age,50).
```

In this paper, a slightly different representation is used, having the advantage that the hierarchical relation between an object and its attributes, and the relation between an attribute and its (possible) values are expressed explicitly:

```
greaterthan(patient(age(50))).
```

In this representation, the predicate "greaterthan" has just one argument: `patient(age(50))`.

The conclusion

conclude patient cholestasis intrahepatic 0.40

is represented as an atomic formula in

```
conclude(patient(cholestasis(intrahepatic,0.40))).
```

Thus, the production rule

```
if
   same patient pain colicky and
   same patient cholestasis extrahepatic
then
   conclude patient diagnosis common_bile_duct_stone 0.70
```

may be represented in the Horn clause formalism in a straightforward manner:

```
conclude(patient(diagnosis(common_bile_duct_stone,0.70))) :-
   same(patient(pain(colicky))),
   same(patient(cholestasis(extahepatic))).
```

The fact

patient diagnosis common_bile_duct_stone 0.35

is represented in the Horn clause formalism as

```
patient(diagnosis(common_bile_duct_stone,0.35)).
```

Anticipating the extension onto the PROLOG interpreter described in the next section, the representation of rules and facts as Horn clauses is slightly modified.

As mentioned before, an inference engine applying top down inference considers facts before production rules when evaluating conditions. As the PROLOG interpreter considers Horn clauses in the order in which they are specified in the PROLOG database, it is sufficient to add facts to the database before the corresponding production rules as long as facts and conclusions are represented the same way. If the PROLOG database contains

```
patient(diagnosis(common_bile_duct_stone,0.35)).
patient(diagnosis(common_bile_duct_stone,0.70)) :-
   same(patient(pain(colicky))),
   same(patient(cholestasis(extrahepatic))).
```

the fact `patient(diagnosis(common_bile_duct_stone,0.35)).` is considered before the rule concluding on the diagnosis of the patient.

Production rules and facts themselves can provide the basis of the procedures required to evaluate the rules, because rules and facts are represented as Horn clauses. Production rules, for instance, can see to the computation of certainty factors and the insertion of facts into the PROLOG database. The principle of expert system shells, however, is the strict separation of knowledge and inference. Therefore, an inference engine is developed to see to control issues like the computation of certainty factors.

After the evaluation of a production rule, not only the certainty factors of the conclusions should be known to the inference engine, but also the certainty factors resulting from the evaluation of the conditions. Because all variables in PROLOG have a scope of just one clause, the certainty factors are assembled in a list of certainty factors in an extra argument in the conclusion:

```
patient(diagnosis(common_bile_duct_stone,0.70,[CF1,CF2])) :-
   same(patient(pain(colicky,CF1,_))),
   same(patient(cholestasis(extrahepatic,CF2,_))).
```

The production rule formalism permits more than one conclusion in a production rule; the Horn clause formalism, however, does not. Therefore, a production rule comprising more than one conclusion, is split up into as many Horn clauses as there are conclusions in the rule. For instance, the production rule

if
  lessthan patient age 25 **and**
  same patient cholestasis intrahepatic **and**
  same patient signs Kayser_Fleischer_rings
then
  conclude patient diagnosis Wilson's_disease 0.95 **and**
  conclude patient diagnosis primary_biliary_cirrhosis 0.05

is represented in two Horn clauses:

```
patient(diagnosis(wilson_s_disease,0.95,[CF1,CF2,CF3])) :-
   lessthan(patient(age(25,CF1,_))),
   same(patient(cholestasis(intrahepatic,CF2,_))),
   same(patient(signs(kayser_Fleischer_rings,CF3,_))).
patient(diagnosis(primary_biliary_cirrhosis,0.05,[CF1,CF2,CF3])) :-
   lessthan(patient(age(25,CF1,_))),
   same(patient(cholestasis(intrahepatic,CF2,_))),
   same(patient(signs(kayser_Fleischer_rings,CF3,_))).
```

In conclusion it is noted that in the representation of conditions connected by an **or** the PROLOG ";" cannot be used consequence upon the computation of certainty factors. A predicate "or" is introduced having two arguments, a list of the conditions connected by an **or**, and the certainty factor into which the certainty factors resulting from the evaluation of these conditions are combined. The production rule

if
  same patient complaint bruises **or**
  same patient signs purpura
then
  conclude patient liverdisorder decompensated 0.90

is represented in the Horn clause

```
patient(liverdisorder(decompensated,0.90,[CF1])) :-
   or([same(patient(complaint(bruises,_,_))),
   same(patient(signs(purpura,_,_)))],CF1).
```

4.2. A PROLOG inference engine

Though, the backtracking scheme incorporated in the PROLOG interpreter is essentially an implementation of top down inference, it is not applicable as an inference engine straight-away. Some features inherent in inference engines have to be added to the PROLOG interpreter: the computation of certainty factors and the insertion of facts into the PROLOG database have been mentioned before. Another addition is the notion that the user is asked to supply additional information when an attribute cannot be traced by applying the production rules in the knowledge base. Furthermore, the definitions of the predicates mentioned in the conditions of the rules have to be supplied to the PROLOG interpreter.

The tracing of goal attributes is monitored essentially by five Horn clauses:

```
trace_fact(Fact) :-
     clause(Fact,true),!.
trace_fact(Fact) :-
     initialize(Fact,Clause,Object,Attribute),
     trace_value(Clause,Object,Attribute),!,
     clause(Fact,true).

trace_value(Fact,Object,Attribute) :-
     clause(Fact,true),!,fail.
trace_value(Rule,Object,Attribute) :-
     call(Rule),
     process(Rule),
     fail.
trace_value(Fact,Object,Attribute) :-
     ask(Fact,Object,Attribute).
```

Initially, the variable "Fact" is instantiated with a statement of the goal to achieve, for instance, the clause

```
patient(diagnosis(Value,CF,_)).
```

In this example, the goal of the consultation is to establish values and corresponding certainty factors in relation to the goal attribute "diagnosis" of the object "patient".

The procedure "trace_fact" consists of two entries. The first entry examines whether the fact searched for has been established before and is already present in de knowledge base, by means of the PROLOG predicate "clause": a procedure call `clause(X,Y)` locates the first clause in the database whose procedure heading matches `X` and whose body matches `Y`. The body of a clause stating a fact, is the term `true`.

The second entry of "trace_fact" initializes the tracing of the goal attribute specified in the instantiation of the variable "Fact".

The procedure "trace_value" consists of three entries. If at least one fact related to the specified attribute and object is known, the inspection of the knowledge base is ended, because the attribute is traced already.

Otherwise, the production rules concluding on the attribute to be traced are applied by means of the PROLOG predicate "call" in the second entry of the procedure "trace_value". The predicate "call" treats its argument as a procedure call. When its argument matches with a clause, this matching clause is a relevant production rule. Then, the procedure calls in the body of the matching clause are executed, in other words, the conditions of the selected rule are evaluated. When the evaluation of the conditions of a rule has been successful, the conclusion of the rule establishes a new fact. This fact is inserted into the knowledge base with the correct certainty factor by means of the procedure call to the procedure "process". The third procedure call, `fail`, is a PROLOG predicate used to force backtracking in order to apply all the relevant rules.

The body of the third entry of the procedure "trace_value" consists of a procedure call to the procedure "ask", which examines whether the application of the production rules has been succesful. If the attribute is still not traced, the user may be asked to supply additional information.

The predicates mentioned in the conditions are defined as Horn clauses. These definitions therefore provide the procedures for the evaluation of the predicates. Each procedure body comprises a procedure call to the procedure "trace_fact" as in

```
same(Fact) :-
        trace_fact(Fact),
        arg(1,Fact,Argument),
        arg(2,Argument,CF),!,
        CF > 0.2.
```

This Horn clause is the definition of the predicate "same". Because of the procedure call to "trace_fact", a condition serves virtually as a call to the Horn clauses representing the relevant production rules.

The condition `same(Fact)` evaluates to be successful if the fact `Fact` is or can be inferred from the knowledge base and the certainty factor associated with the fact is greater than 0.2. The three final procedure calls shape the test on the certainty factor.

5. A COMPARISON WITH THE DELFI-2 SYSTEM

The PROLOG expert system shell described in section 4. is developed after the example of the DELFI-2 system. Several knowledge bases have already been developed using the DELFI-2 expert system shell. So as to evaluate the PROLOG expert system shell and to compare it with the DELFI-2 system one of these knowledge bases is translated into PROLOG Horn clauses. The knowledge base chosen for this purpose is the knowledge base called Hepar. Hepar holds expert knowledge on liver and biliary disease [25] and is still being augmented and improved by P. J. F. Lucas at the Centre for Mathematics and Computer Science and A. R. Janssens at Leyden University Hospital. The examples used throughout this paper are taken from Hepar.

The two systems were run with Hepar on a VAX 11/780 computer and a PC. Though a proper comparison of the systems cannot be made because of the different characters of the systems, a few striking differences were elicited.

The development of the PROLOG expert system shell took an experienced PROLOG programmer about two weeks. Even if the time spent on research is left out of consideration, this is a substantially smaller period of time than the months spent on the development of the DELFI-2 system. Most of the discrepancy is owed to the fact that the larger part of the inference engine is already provided for in the PROLOG interpreter, and the near resemblance of the Horn clause formalism to the production rule formalism.

These two observations also account for the size of the PROLOG expert system shell: including a primitive user interface, the extensions onto the PROLOG interpreter cover about seven pages. This size renders the PROLOG expert system shell easy to maintain. A comparison with the size of the DELFI-2 system would not be appropriate, because the DELFI-2 system is a rather elaborated system, whereas the PROLOG expert system shell is just a prototype system lacking a satisfactory user interface and explanation facilities.

Furthermore, a consultation of the PROLOG expert system takes about five times the duration of the same consultation of the DELFI-2 system. Part of the inefficiency of the PROLOG expert system is owed to the Horn clause representation of production rules containing more than one conclusion. At the time of the translation of the knowledge base, Hepar held 171 production rules. Because rules having more than one conclusion had to be represented in more than one Horn clause, the PROLOG knowledge base holding the same knowledge consisted of 273 production rules. In the DELFI-2 system, the conditions of a production rule having several conclusions are evaluated once, whereas these conditions are evaluated several times in the PROLOG expert system. Although this observation accounts for part of the inefficiency of the PROLOG expert system, considerable inefficiency originates from the used interpreters for the PROLOG language.

6. Conclusion

The programming language PROLOG has some substantial advantages in building rule based expert systems applying top down inference as a knowledge manipulation scheme:

- the separation of knowledge and inference as a principle in the development of expert system shells, is achieved rather naturally consequence on the principles of logic programming
- production rules are represented as Horn clauses in a straightforward manner
- part of the inference engine is already provided for in the PROLOG interpreter

As to the use of a PROLOG expert system in a real-life environment PROLOG, however, has a considerable drawback: its inefficiency compared with, for instance, Pascal. Nonetheless, a PROLOG rule based expert system can be developed in a short notice of time making PROLOG a suitable language for the rapid development of prototype systems.

In this paper, just one of several possible knowledge representation schemes is discussed. Part of the future investigations at the Centre for Mathematics and Computer Science will be directed to the representation of frames and corresponding inference schemes as Horn clauses.

References

1. W. van Melle (1980). *A domain-independent system that aids in constructing knowledge-based consultation programs,* Stanford University, Stanford.
2. H. de Swaan Arons, P. J. F. Lucas (1984). Expert systems in a application oriented environment (in Dutch), *Informatie,* vol. 26,8: 361 - 637.
3. L. Browston, R. Farrell, E. Kant, N. Martin (1986). *Programming expert systems in OPS5,* Addison-Wesley, Reading Massachusetts.
4. F. Hayes-Roth, D. A. Waterman, D. B. Lenat eds. (1983). *Building expert systems,* Addison-Wesley, Reading Massachusetts.
5. P. H. Winston (1985). *Artificial Intelligence,* Addison-Wesley, Reading Massachusetts.
6. R. A. de By (1985). Semantic nets 1, 2 (in Dutch), *Informatie,* vol. 27,9: 782 - 794, vol. 27,10: 890 - 902.
7. R. Fikes, T. Kehler (1985). The role of frame-based representation in reasoning, *Communications of the ACM,* vol. 28,9: 904 - 920.
8. B. G. Buchanan, R. O. Duda (1983). Principles of rule-based expert systems, *Advances in Computers,* vol. 22: 163 - 216.
9. R. A. Kowalski (1979). *Logic for problem solving,* North-Holland, New York.
10. G. Shafer (1976). *A mathematical theory of evidence,* Princeton University Press, Princeton.
11. L. A. Zadeh (1975). Fuzzy logic and approximate reasoning, *Synthese,* vol. 30: 407 - 428.
12. B. G. Buchanan, E. H. Shortliffe (1984). *Rule-based expert systems: the MYCIN experiments of the Stanford Heuristic Programming Project,* Addison-Wesley, Reading Massachusetts.
13. P. J. F. Lucas (1986). Knowledge representation and inference in rule-based systems, *Colloquium Knowledge Based Systems,* CWI, Amsterdam.
14. H. Coelho, J. C. Cotta, L. M. Pereira (1980). *How to solve it with PROLOG,* Laboratorio Nacional de Engenharia Civil, Lisboa.
15. D. H. D. Warren (1980). Logic programming and compiler writing, *Software - Practice and experience,* vol. 10: 97 - 125.
16. H. Gallaire, J. Minker (1978) *Logic and data bases,* Plenum Press, New York.
17. J. W. Lloyd (1983). An introduction to deductive database systems, *The Australian Computer Journal,* vol. 15,2: 52 - 57.
18. P. J. F. Lucas, L. C. van der Gaag (1986). PROLOG and expert systems (in Dutch), *Informatie,* vol. 28,1: 16 - 25.
19. P. A. Subrahmanyam (1985). The software engineering of expert systems: is PROLOG appropriate ? *IEEE Transactions on Software Engineering,* vol. SE-11,11: 1391 - 1400.

20. J. A. ROBINSON (1965). A machine-oriented logic based on the resolution principle, *Journal of the ACM,* vol. 12,1: 23 - 41.
21. L. C. VAN DER GAAG, P. J. F. LUCAS (1985). PROLOG: specification = implementation (in Dutch), *Informatie,* vol. 27,9: 766 - 774.
22. W. F. CLOCKSIN, C. S. MELLISH (1981). *Programming in PROLOG,* Springer-Verlag, Berlin.
23. F. KLUZNIAK, S. SZPAKOWICZ (1985). *PROLOG for programmers,* Academic Press, London.
24. J. A. CAMPBELL (1984). *Implementations of PROLOG,* Ellis Horwood, Chichester.
25. P. J. F. LUCAS, A. R. JANSSENS (1986). Hepar, an expert system for diagnosing disorders of the liver and biliary tract (in Dutch), *MIC '86 Proceedings,* NGI: 283 - 288.