"Good work in Artificial Intelligence concerns the automation of things we know how to do, not the automation of things we would like to know how to do." -Alan Perlis.

Acknowledgements.

I wish to acknowledge the support and sympathy of many people and friends who helped me in the development of this project. Thanks to my project supervisor, Paul Powell (Lecturer in the Institute of Technology, Sligo) for his guidance, understanding and time dedicated to me. I wish to thank also the rest of lecturers of the Degree in Science of Computing at the Institute of Technology, John O'Donnell, John Kelleher and Tom McCormack, who provided a source of many useful ideas applied in this Project.

Particular mention must be made to the students of the Degree in Science of Computing, whose collaboration and friendship far exceeded what can be considered as hospitality with foreigners.

Finally, my family in Granada, although more than 2,000 miles away from here, were a continuous source of support and courage for me.

Main Project: Development of an Expert System Shell.

Technical Manual

1. Introduction.

1.1. Purpose.

The following document describes the analysis and development process carried out during the elaboration of the Main Project of the Bachelor in Science of Business Computing. The Project is included in the scope of Artificial Intelligence. The objective of the Project is the development of an application, which will be used as a shell for the creation, manipulation, and query of expert systems. Special efforts were put in developing an user interface that will be easy to use and independent of the *Knowledge Base* used, taking advantage of the actual windows environment that visual programming languages (such as C++ Builder, Delphi, Visual C++ or Visual Basic) provide.

1.2. Artificial Intelligence

Artificial Intelligence is a difficult concept to define. Almost each expert in the field of AI proposes a different definition. Some of the most simple and concise are the following:

"Artificial Intelligence is basically a theory of how the human mind works"
"Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better" (Rich, 1983).
"Artificial intelligence is behaviour by a machine that, if performed by a human being, would be called intelligent".

The major areas in which Artificial Intelligence has been applied are:

- Natural Language Processing
- Speech (Voice) Understanding
- Robotics and Sensory Systems
- Computer Vision and Scene Recognition
- Intelligent Computer-Aided Instruction
- Machine Learning (Neural Computing)
- Expert Systems, which are the most popular and successful AI Technology at the moment.

1.3. The Growth of Expert Systems.

There has been, in the past decade and present decade, a virtual explosion of interest in the field known as expert systems (or, alternatively, as knowledge-based systems). Appearing from seemingly out of nowhere, expert systems have quickly evolve from an academic notion into a proven and highly profitable product –one that offers an efficient and effective approach to the solution of an exceptionally wide array of important, real-world problems. In particular, expert systems provide a powerful and flexible means for obtaining the solution to a variety of problems that often can not be dealt by other, more *orthodox* methods. Typically, such problems have always been considered too large and too "messy" for solution by conventional approaches. In this regard, expert systems may be considered to represent a potent new instrument to solve this kind of complex problems.

By the end of the 1980s, the implementation of at least 2000 expert systems had already been documented within the corporate world alone (Ignizio, 1991). However, most of the implementations had occurred in the military sector, within local, state, and federal government. The number of expert systems implementations has increased dramatically during the 1990s. In the United States alone, revenues from expert systems reached $6 billion by 1995 (nearly eight times over the receipts documented in 1989).

1.4. What is an Expert System?

Knowledge-based expert systems, or simply expert systems, are computer programs that emulate the reasoning process of a human expert or perform in an expert manner in a domain for which no human expert exists or the cost of human expertise is too high. Typically they reason with uncertain information. In addition to the descriptive knowledge, Expert Systems also imitate the expert's reasoning processes (procedural knowledge) to solve specific problems

Expert Systems use human knowledge to solve problems that normally would require human intelligence. These expert systems represent the expertise knowledge as data or rules within the computer. These rules and data can be called upon when needed to solve problems. Books and manuals have a tremendous amount of knowledge but a human has to read and interpret the knowledge for it to be used. Conventional computer programs perform tasks using conventional decision-making logic - containing little knowledge other than the basic algorithm for solving that specific problem and the necessary boundary conditions. This program knowledge is often embedded as part of the programming code, so that as the knowledge changes, the program has to be changed and then rebuilt. Knowledge-based systems collect the small fragments of human know-how into a knowledge base which is used to reason through a problem, using the knowledge that is appropriate. A different problem, within the domain of the knowledge base, can be solved using the same program without reprogramming. The ability of these systems to explain the reasoning process through back-traces and to handle levels of confidence and uncertainty provides an additional feature that conventional programming does not handle.

Most expert systems are developed via specialised software tools called shells. These shells come equipped with an inference mechanism (backward chaining, forward chaining, or both), and require knowledge to be entered according to a specified format. These shells qualify as languages, although certainly with a narrower range of application than most programming languages.

Expert systems are limited by the information contained in their database. It is up to the knowledge engineer and the expert to work together to gather the correct information and inference rules to be contained in the knowledge base. Therefore, a productive expert system needs, not only a good inference engine, but also a good knowledge base.

1.5. History of Expert Systems.

The digital computers have been since World War II wherein the British and Americans used them in such tasks as numerical computations and code breaking. However, while the computer possesses an astonishing capability to store and manipulate data, few of us would consider such activities to constitute any form of true intelligence. These activities are considered to be mechanical in nature.

However, among the early users and designers of digital computers were a small group of "radicals": they wondered if one might not be able to use the computer as a means for simulating various aspects of human intelligence. As a result, a conference was held in 1956 in Hanover, New Hampshire, the home of Dartmouth College. In that conference, now known as the Dartmouth Conference, certain forecasts for Artificial Intelligence were established (Simon and Newell, 1958). Specifically, it was predicted that, by 1970, a computer would do the following:

- Be a grandmaster at chess.
- Discover significant, new mathematical theorems.
- Understand spoken language and provide language translations.
- Compose music of classical quality.

Rather obviously, and with our advantage of hindsight, these forecasts were too ambitious. However, these forecasts did serve to encourage wider interest in AI and establish certain goals for those within the field. But they also served to have a very negative impact as, with the passage of time, it became clearer that these forecasts were not going to be met and had been, at least, unrealistic.

The early enthusiasm in AI was, by the mid-1960s, considerably subdued. In fact, AI seemed to virtually vanish at about that time and it did not resurrect until the 1980s. The AI community felt that it was time to reconsider the goals. Developments and research in AI should be as follows:

- More modest.
- More focused.
- Directed toward a narrow sector of expertise, rather than general, overall intelligence.

The name given to this sub-field of AI was *expert systems*, or *knowledge-based systems*.

Some of the most relevant Expert Systems have been:

- DENDRAL: it was the first expert system and was started in 1965. Edward Feigenbaum helped to develop DENDRAL and in the process he created and named the field of Knowledge Engineering.

- MYCIN: it was the first system that deliberately mimicked human behaviour in terms of its questioning strategy. It was also the prototype for all rule based expert systems. Therefore, MYCIN is a continuous point of reference in the development of this project.

- MACSYMA: it is a symbolic mathematics engine that does algebra, calculus, differential equations, etc. It is an example of old style knowledge systems. It uses brute force and exhaustive searching to solve problems.

- HEARSAY I and II: they were the prototypes for speech recognition involving sentences rather than just individual words.

- INTERNIST: it is the largest expert system currently being developed. Ideally it will be a diagnostic tool for the entire field of internal medicine. In the last 25 years they have covered 25% of the field. In this system, diseases, symptoms, etc. are coded as objects, which are allowed to interact according to a series of rules.

- PROSPECTOR: it is an ore-deposit locator. It was the first system to use semantic networks in its knowledge base.

- PUFF: it is a diagnostic tool that interprets the results of respiratory tests. It was designed using an expert system shell derived from MYCIN called EMYCIN. Whereas MYCIN to 50 person-years to complete, PUFF took 5.

Later expert systems were primarily designed by and/or for companies. Instead of being research tools they are more commercially oriented and are expected to turn a profit. In 1990, more than 50% of the businesses listed in Fortune 500 were developing expert systems, either for in house use or to sell.

A major goal of expert system developers is to make the systems more artificially intelligent. The current focus is on designing programs that have a deeper theoretical understanding of their domain, and that can provide deeper explanations. New expert systems utilise meta-knowledge about their own knowledge base, inference engines and general construction to develop more informed heuristics to implement in their rules systems. As programs grow larger and more complex, the program must assume some of the burden of understanding its own behaviour, documenting and justifying itself, and even modifying itself when necessary.

1.6. Expert System Architecture.

The following figure depicts one possible representation of an expert system:

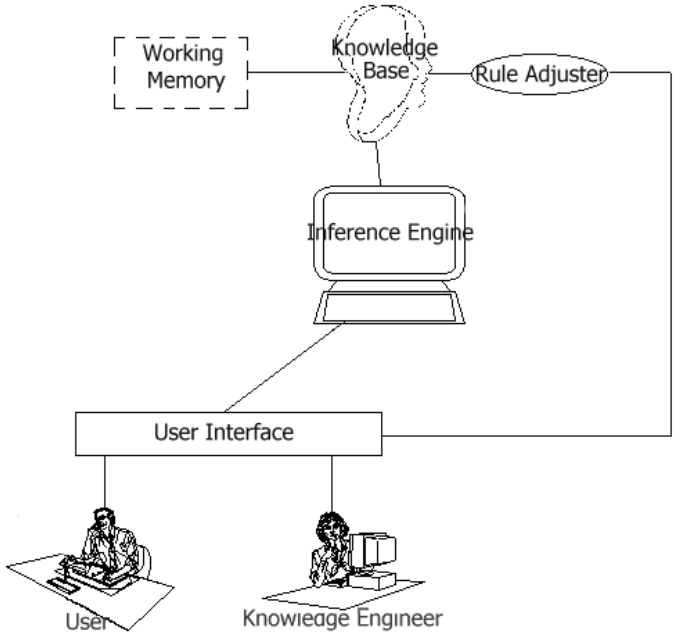


Fig.1. A generic expert system.

Note that access capabilities for two types of human users are provided. One is that individual designated as the *knowledge engineer*. The knowledge engineer is the person responsible for placing the knowledge into the expert system's knowledge base (the portion of the of the expert system shown at the top of Fig.1.) He or she accomplishes this through the *interface* and the *rule adjuster*.

The knowledge engineer is also the interface between the human expert (if there is one) and the expert system. That is, the knowledge engineer somehow must capture the expertise of the human expert and then express this expertise in a format that may be stored in the knowledge base and will be used by the expert system. In the ideal expert system, there would be no need for a knowledge engineer. The domain expert would interact directly with the expert system and would replace the knowledge engineer in the figure.

The second type of individual with access to the expert system is designated, in Fig.1. as simply the *user*. This designation refers to anyone who will be using the expert system as a decision-making aid (i.e., as a consultant). The successful knowledge engineer must always

keep in mind that the expert system is ultimately intended for the benefit of the user, not for that of the knowledge engineer or the domain expert.

The *interface* handles all input to the computer and controls and formats all output. A well-designed interface would be one that exhibits ease of use (so called user friendliness), even for the novice user. The interface also handles all communication with the knowledge engineer during the development of the expert's system knowledge base. Another property that is sometimes exhibited in expert systems is that of explanation.

The *inference engine* is employed during a consultation session. During consultation, it performs two primary tasks. First, it examines the status of the knowledge base and working memory so as to determine what facts are known at any given time, and to add any new facts that became available. Second, it provides for the control of the session by determining the order in which inferences are made. The inference engine serves to merge facts with rules to develop, or infer, new facts.

The *knowledge base* is the very heart of any expert system. A knowledge base will typically contain two types of knowledge, that is, facts and rules. The facts within a knowledge base re
present various aspects of a specific domain that are known *prior* to the consultation session of the expert system. The rules within a knowledge base are simply heuristics. If the knowledge base has been constructed through interaction with a human expert, these rules represent the knowledge engineer's perception of the heuristics that are employed by the expert in decision making.

The *working memory* of an expert system changes according to the specific problem at hand. The contents of the working memory consist of facts. However, unlike the facts within the knowledge base, these facts are those that have been determined for the specific problem under consideration *during* (and at the conclusion of) the consultation session or query.

The final module to be discussed is the *rule adjuster*. In most existing expert systems, the rule adjuster serves merely as a rule editor. That is, it enters the rules specified by the knowledge engineer into the knowledge base during the development phase of the expert system. In more ambitious expert systems, the rule adjuster may be used in an attempt to incorporate *learning* into the process. In such instances, we would teach the expert system by providing it with a set of examples and then critique its performance. If its performance is unsatisfactory, the rule adjuster automatically revises the knowledge base. If satisfactory, the rule adjuster may simply reinforce the existing knowledge base.

1.7. Expert System Shells.

By this time, the difference between an expert system and an expert system shell is clear: an expert system includes all of the components discussed above minus the knowledge base. Using a shell, it is up to the knowledge engineer to develop the knowledge base and to then insert this knowledge base into the architecture to form a complete expert system, as intended for a specific domain. The use of a shell thus frees the knowledge engineer from the need to repeatedly develop all the supporting elements of an expert system, and thus to focus his or her attention on the development of the knowledge base.

It must remain clear that this project is about developing and implementing an Expert System Shell. Therefore the main focus and effort is on the design of the inference engine and the rest of the elements associated with it (interface, rule adjuster, knowledge base structure,

explanation facility, working memory...) rather than the contents of the knowledge base. The system will work independently of the knowledge base used, being the knowledge engineer's responsibility to develop a well-structured and consistent knowledge base.

The architecture of the generic expert system depicted in Fig.1., should serve to indicate at least some of the differences between this approach and that of algorithmic procedures and heuristic programming. In particular, note that the knowledge base is separated from the inference engine. In other words, and unlike algorithms and heuristic programming, an expert system separates the heuristic rules from the solution procedure. The knowledge base contains a description, or model, of *what we know* (i.e. about deriving a solution to a given problem). The inference engine contains a description of *what we do* to actually develop the solution. While the knowledge base changes from domain to domain, the inference engine remains the same.

1.8. Limitations.

Some limitations arise when comparing expert systems with conventional human expertise:

- Fragile Systems: small environment changes can force revision of all of the rules.
- Vague Rules: rules can be hard to define. It is a difficult task for an expert to express his knowledge in terms of rules.
- Conflicting Experts: With multiple opinions, who is right?
- Unforeseen events: events outside of domain can lead to nonsense decisions.
- Expert systems can handle only narrow domains, otherwise the knowledge base will contain millions of rules.
- Expert systems do not possess common sense. A human expert uses common sense to reach solutions, but this concept can not be built (at the moment) in a computer.
- Expert systems have limited ability to learn and these limitations lead to loss of faith on them.

Neural Nets and Fuzzy Logic are new approaches that can solve some of this limitations in the future. Neural Nets provide great ability to learn, while Fuzzy Logic represents the best approach nowadays to human uncertainty and imprecision.

2. Requirements Definition.

The objective of the Project can be summarised in one sentence as: "Design and implement a rule-based expert system shell, applying it to an specific field (i.e. choose a car, diagnose heart diseases …)"

The requirements for the Main Project are the usual for an Expert System Shell. We paid special attention to the easiness of use.

- Production rules based expert system shell. The knowledge in the knowledge base is stored as rules.

  Formal definition:

  Knowledge base ::= [Rule]*,
  Rule ::= IF [Premise]* THEN Conclusion,
  Premise ::= Attribute, Value.
  Conclusion ::= Attribute, Value.
  (Attribute and Value are strings of characters)

- Friendly User Interface. Facilitate interaction with the user using windows-style, menus, graphics,… Avoid conversational menus (typical of the first expert systems).
- Knowledge Base Editor. The user must be able to create and modify the knowledge base just using the application, with no need of an external database management system.
- Support of uncertainty (certainty or confidence factors) during the consultation or query session.
- Explanation facility, with the ability to explain why and how a particular conclusion was reached.
- Capability of inserting known facts at the beginning of the consultation session, so that the typical "question-and-answer" process of expert systems is reduced.
- Capability of changing single values in consecutive consultations in order to observe differences in the conclusions reached (*differential analysis*).
- Use of graphics and any additional multimedia feature which aids the user recognition and association with the *real* world.

3. System Specification.

Characteristic	Value
Programming Language	C++
Programming Language Environment	Borland C++ Builder, Version 1.0.
Operating System	Microsoft Windows 95

3.1. Programming Language.

The programming language used in the development of this project is C++ (Object Oriented C). The programming paradigm is, therefore, procedural[1]. However, a lot of expert systems have been built following the declarative programming paradigm, using programming languages such as Prolog, Clips or Lisp. Those programming languages are typical of Artificial Intelligence and provide a virtually built-in inference engine for the expert system.

Procedural languages require the programmer to specify the precise, step-by-step set of instructions to be carried out in the solution of a problem. Declarative languages, on the other hand, only require that one provide the program with the facts and relationships (e.g. data and knowledge base) that exist for a problem; the solution to the problem is then accomplished by the language's own internal inference engine.

The question then is: Why use C++, where we have to implement our own inference engine, instead of using Prolog, for instance, which already gives the programmer a built-in inference engine? Several reasons lead to the use of a procedural language:

- First of all, the purpose of the project is academic, so that the implementation of an inference engine will show the programmer how an expert system works internally.

- Using a declarative programming language restricts the flexibility of the expert system. For instance, Prolog employs backward chaining and backtracking. Search is conducted depth first and all rules are scanned. We can not change naturally the way Prolog works[2], and consequently, we will lose some control over the expert system inference mechanism.

  Nevertheless, the development of a rule-based expert system in Prolog is, quite possibly, easier than with any other language. As an indication, consider the following production rule:

  Rule: If inflation is high
  Then invest in gold.

  The same rule in Prolog may be written as:

```
gold_investment(yes):-   write('Inflation :'),
                         read(Inflation),
                         Inflation=high.
```

[1] Some authors distinguish between the Procedural Paradigm and the Object Oriented Paradigm. However, for Expert Systems development this distinction is not significant.
[2] Using a Prolog program we can change form Backward to Forward Chaining, but internally, Prolog will still work backwards.

- Building our own expert systems software package gives us control over the entire consultation/inference process, as well as the ability to modify, revise, or enhance any portion of the package that we wish.

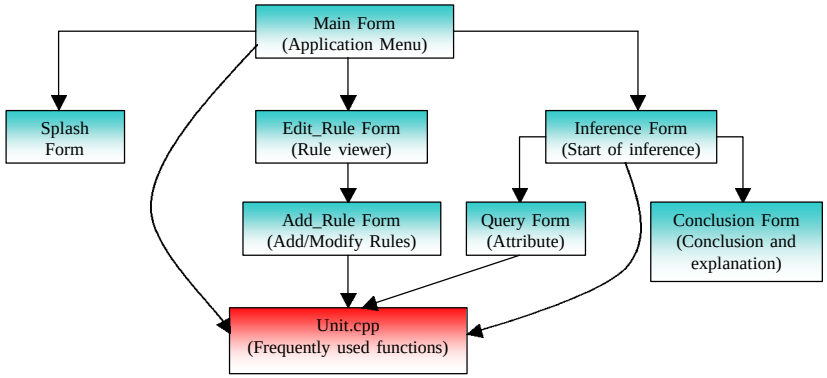
- The manipulation of tables (Paradox tables or dBase tables) is easier with C++ Builder. In Prolog or Lisp the database would have to be specially built for the application and would not be standard.

- The inclusion of multimedia features (graphics, sound,...) is easier with visual programming packages such as Visual Basic or C++ Builder.

  Another question is why use C++ Builder instead of Visual C++ or Visual Basic. The answer is simply "I used C++ Builder because is the environment that I know better". In fact, this system could be easily translated into MS Visual C++ or MS Visual Basic. C++ Builder is powerful and easy to use, although is not so well known or standard as the Microsoft's packages.

4. System Architecture.

The system architecture is guided by the way C++ Builder (and most visual programming packages) works, i.e. using forms. The various components of the expert systems shell (knowledge base editor, inference process) correspond to several forms which are accessible from the main form. The hierarchical structure of the system is the following:



5. The Knowledge Base.

5.1. Knowledge Acquisition.

As it was mentioned in the introductory section, the main focus of this project is not in the contents of the knowledge base, but in the operation of the expert system shell itself. Therefore; the knowledge base rules may not be optimal and there are not meta-rules (rules that give information about rules) that increase the efficiency of the inference process. However, in order to test the expert system shell and to provide a tangible application in the real world, a relatively large knowledge base was built.

It would seem that the most obvious way in which one may acquire a knowledge base is to go directly to the human expert. However, there are some reasons why this may not work, or at least not provide totally satisfactory results (Ignizio, 1991):

- For some problems, there simply may not be an expert. E.g. Investing in the stock market.
- The alleged experts may actually be exhibiting poor to mediocre performance. All too often, the term expert is loosely applied to anyone who simply *gets the job done*.
- The experts may not wish to reveal their secrets.
- The experts often are just unable to articulate the approach that they use. Many experts, in fact, simply and honestly do not really understand how they actually make their decisions.

Moreover, if we use an expert the knowledge engineer has to plan carefully several meetings with him, extracting all the knowledge needed for the knowledge base. A special situation is the use of multiple domain experts. This situation can be specially frustrating if not properly handled. A good approach (Surko, 1989) is using a rule base cloned from one expert and then build a prototype expert system and let the other domain experts critique the results.

Knowledge acquisition can also be accomplished via rule induction: convert an existing database into a set of production rules. A popular approach to rule induction is the algorithm ID3 (Quinlan, 1983) in which rules are generated from trees.

5.2. Knowledge Representation.

The knowledge that is contained within an expert system consists of:

- *A priori* knowledge: the facts and rules that are known about a specific domain prior.
- Inferred knowledge: the facts concerning a specific case that are derived during, and at the conclusion of, a consultation with the expert system.

There are several approaches to represent knowledge in a expert system:

- Semantic networks.
- Frames.
- Neural Networks.
- Production rules.

The knowledge mode of representation chosen for the development of our expert system shell was by the use of **production rules** or rule-based systems. There are several reasons which lead me to that choice:

1. The majority of existing expert systems (and especially expert system shell), employ rule bases.
2. Rules represent a particularly natural mode of knowledge representation. Consequently, the time required to learn how to develop rule bases is minimised.
3. Rule bases can be relatively easily modified. In particular, additions, deletions and revisions to rule bases are relatively straightforward processes[3].

Production rules are of the IF-THEN variety. To represent the ELSE statement we only have to split the rules in two.

E.g.: IF StudentScore is 50 or more
THEN admit the student
ELSE do not admit the student

This rule is equivalent to:
IF StudentScore is 50 or more
THEN admit the student

IF StudentScore is less than 50
THEN do not admit the student

Similarly the use of OR in the premises can be avoided by splitting the premises and keeping the same conclusion.

As noted, each premise and conclusion clause contains attributes and values. In the example above, an attribute would be "StudentScore" and a value "50 or more".

The rules were internally represented in the Project by Paradox tables. There are four tables which make up a complete rule:

- Premises.db: there is one entry for each premise[4] in each rule

Field Name	Type	Size
RID	Short Integer	
PID	Short Integer	
Attribute	String	40 chars.
Value	String	40 chars.
Status	String	1 char.
CF	Number	

Where:
RID: rule identifier. Each rule has an unique identifier which acts as the primary key.
PID: premise identifier. Identifies each premise in a rule.
Attribute: name of the attribute.
Value: value given to the attribute.
Status: status of the premise (Active, true, or false)
CF: numeric value of the certainty factor (between 0 and 100)

- Prem2.db: each entry corresponds to all the premises of a rule.

[3] This also depends on the "quality" of the knowledge base. If we have a well-designed rule bases, the process will be transparent.
[4] By premise, we understand each of the pairs {Attribute, Value} which are in the IF clause of the rule.

Field Name	Type	Size
RID	Short Integer	
Premises	String	255 chars.

Where:
RID: rule identifier. Each rule has an unique identifier which acts as the primary key.
Premises: a string that represents all the premises in the rule in the format "IF [Attribute1=Value1] AND [Attribute2=Value2]…AND [AttributeN=ValueN]" .

- Conclusions.db: there is one entry for each conclusion.

Field Name	Type	Size
RID	Short Integer	
Images	Graphic	

Where:
RID: rule identifier. Each rule has an unique identifier which acts as the primary key.
Attribute: name of the attribute.
Value: value given to the attribute.
Status: status of the rule (Active, Inactive, Triggered, or Fired)
CF: numeric value of the certainty factor (between 0 and 100)

- Images.db: each entry corresponds to an image of each conclusion.

Field Name	Type	Size
RID	Short Integer	
Image	Graphic	

Where:
RID: rule identifier. Each rule has an unique identifier which acts as the primary key.
IMAGE: graphic associated with the conclusion of the rule with identifier RID.

A rule is retrieved from the database by joining these tables using the RID field. Although the tables are physically separated, they will appear logically as an unit in the application.

5.3. Rule Editor.

The rule editor or rule adjuster as shown in Fig.1. will allow the knowledge engineer to add, delete or modify rules. The rules are shown in the grid by using a SQL query. C++ Builder provides a special class for SQL queries called TQuery. In order to launch the query we give to the object property "SQL Text" the following value:

```
SELECT Prem2.RID, Prem2.Premises,
Conclusion.Attribute,Conclusion.Val,Conclusion.CF
FROM Prem2, Conclusion
WHERE Prem2.RID = Conclusion.RID
```

The results for a simple knowledge base are chosen in Fig..

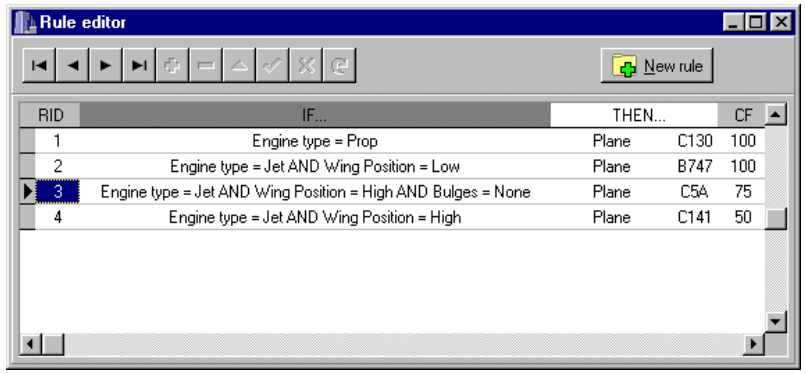


Fig.2.; The Rule Editor window form.

For adding, deleting or modifying rules we use another form that facilitates the task of generating a rule and avoids syntactic errors. It is based in the use of combo boxes. There are two combo boxes for the premises and another for the conclusion. The contents of the "value" combo boxes change dynamically depending on the current selection in the "attribute" combo boxes. The updating is made again by the use of the object "TQuery", specifically using the function implemented in UNIT.CPP "Update_combo":

```
//---------------------------------------------------------------------------
// void UpdateCombo (TComboBox *ComboBox,TQuery *Query, TDBComboBox
*DBComboBox,
//  AnsiString Query_text, AnsiString Field):
//    Fill a combo box with the results of a query.
//---------------------------------------------------------------------------
void UpdateCombo(TComboBox *ComboBox,TQuery *Query, TDBComboBox *DBComboBox,
AnsiString Query_text, AnsiString Field)
{
    Query->Close();
    Query->SQL->Clear();
    Query->SQL->Add(Query_text);
    Query->Prepare();
    DBComboBox->DataField=Field;
    Query->Open();

    DBComboBox->Items->Clear();
    DBComboBox->Items->Add(DBComboBox->Text);
    while(Query->FindNext()){
            DBComboBox->Items->Add(DBComboBox->Text);
    }
    ComboBox->Items=DBComboBox->Items;
    ComboBox->Text=ComboBox->Items->Strings[0];
}
```

The call from the "AddRuleForm" is , therefore, the following:

```
//---------------------------------------------------------------------------
// void __fastcall TAddRuleForm::ComboBox1Change(TObject*Sender)
// If the contents of the first combo box (attribute) change
// the second (values)
//---------------------------------------------------------------------------
void__fastcallTAddRuleForm::ComboBox1Change(TObject *Sender)
{
```



```
      UpdateCombo(ComboBox2,Query1,DBComboBox1,"select distinct val from
      premises where attribute=\""+ComboBox1->Text+"\"", "val");
}
```

The function which adds a rule takes all the inputs given in the form and then post them to the database (note that the user does not have to know the physical structure of the database). The Rule Identifier field is automatically generated by the application, selecting the maximum RID of the posted rules and adding to it 1:

```
      Query1.SQLText=select MAX(RID) from Premises;
      (…)
      DBEdit1->Text = (AnsiString) (atoi(DBEdit1->Text.c_str())+ (int) 1);
```

Once all the fields are filled, we can post the rule to the database. Before, posting it we must check that the value of the Certainty Factor is in the valid range, otherwise the rule will not be posted and the application will show an error message to the user. We use the following function for that purpose (AddRuleBtnClisck):

```
//---------------------------------------------------------------------------
// void __fastcallTAddRuleForm::AddRuleBtnClick(TObject*Sender)
//---------------------------------------------------------------------------
void__fastcallTAddRuleForm::AddRuleBtnClick(TObject*Sender)
{
      int i;
      TMsgDlgButtonsBtns;
      float cf;
      char error[256], str[5];
   // check certainty factor
   Btns<<mbOK;
   strcpy(str, Edit1->Text.c_str());
   cf=atof(str);
   if(cf>100||cf<0){
      wsprintf(error, "The valid range for Certainty Factors is between 0 a
      nd 100");
      MessageDlg(error,mtError,Btns,NULL);
      return;  // do not commit changes
   }

   ConclTable->Open();
   PremTable->Open();

   // post current rule
   for (i=0; i<CurrentRow; i++){ // post premises
      PremTable->Append();
      PremTableRID->Value=atol(DBEdit1->Text.c_str());
      PremTablePID->Value= i+1;
      PremTableAttribute->Value=StringGrid1->Cells[0][i];
      PremTableVal->Value=StringGrid1->Cells[1][i];
      PremTableStatus->Value="A";
      PremTableCF->Value=100.0;
      PremTable->Post();
  }

   // post conclusion
   ConclTable->Append();
   ConclTableRID->Value= atof(DBEdit1->Text.c_str());
   ConclTableAttribute->Value=ComboBox3->Text;
   ConclTableVal->Value=ComboBox4->Text;
   ConclTableStatus->Value="A";
   ConclTableCF->Value=atof(Edit1->Text.c_str());
   ConclTable->Post();

   //we use RID posted to avoid posting 2 times if we click the "Ok" button
   RIDposted=atol(DBEdit1->Text.c_str());
   PremTable->Close();
   ConclTable->Close();
```

```
    CurrentRow=0;
}
```

The "Prem2.db" table (the table with all the premises of each rule together) has also to be updated. For doing so, we use the following function, which goes over the table "Premises.db" looking for rows with the same RID to update the "Prem2.db" table.

```
//---------------------------------------------------------------------------
// void __fastcall TAddRuleForm::RefreshTable()
// Post current rule, close window and refresh rule viewer.
//---------------------------------------------------------------------------
void __fastcall TAddRuleForm::RefreshTable()
{
    int CurrentRID;
    AnsiString premises;
    PremTable->Open();
    LPremTable->EmptyTable();
    LPremTable->Open();

    PremTable->First();
    CurrentRID=PremTableRID->Value;
    premises=(AnsiString)PremTableAttribute->Value+" = "+(AnsiString)
       PremTableVal->Value;
    PremTable->Next();

    // loop through the files of the table
    while (! PremTable->Eof){
       if (PremTableRID->Value==CurrentRID){
        premises=premises+ " AND "+(AnsiString) PremTableAttribute->Value+
             " = "+(AnsiString) PremTableVal->Value;
       }
       else{
          // post current rule
          LPremTable->Append();
          LPremTableRID->Value= CurrentRID;
          LPremTablePremises->Value= premises;
          LPremTable->Post();

          // next rule
          CurrentRID=PremTableRID->Value;
          premises=(AnsiString)PremTableAttribute->Value+" =
              "+(AnsiString)PremTableVal->Value;
       }
       PremTable->Next();
    }//end of while loop

    // post current rule
    LPremTable->Append();
    LPremTableRID->Value= CurrentRID;
    LPremTablePremises->Value= premises;
    LPremTable->Post();

    PremTable->Close();
    LPremTable->Close();
}
```

The processes for editing and deleting an existing rule are approximately the same. Note that the state of the database must be consistent, so that if we modify a particular part of a rule in one table, the others will have to be appropriately modified.

6. The Inference Engine.

The inference engine serves as the inference and control mechanism for the expert system and, as such, is an essential part of the expert system as well as a major factor in the determination of the effectiveness and efficiency of such systems. Inference is the knowledge processing element of an expert system.

The inference strategy used here and in most expert systems is known as *modus ponens*. Simply stated, *modus ponen* means that if the premise of a rule is true, then its conclusion is also true. Thus, if A infers B and A is true, then B is true. This may be represented as A $\rightarrow$ B.

6.1. SQL Compatible Tables Vs. Memory Arrays.

During the inference process the facts inferred and the state of the rules and premises have to be stored somewhere. There are two possibilities: using tables in the same way stored the rules in the knowledge base or using memory bidimensional arrays. Each approach has its advantages and drawbacks:

SQL Compatible Approach.

 "Purer" approach in the meaning that the data can be manipulated solely by SQL queries and updates.

 Homogeneity with the knowledge base.

 It requires a lot of writings to the hard drive, because rule and premise status is continually changing. Thus, efficiency during the inference process decreases.

 It requires duplication of the database at the beginning of each consultation because, otherwise the set of rules of the knowledge base (static during the inference process) would be changed. In the same way, at the end of the consultation the original set of tables would have to be restored. Therefore, efficiency decreases and double hard drive space is required (this can be specially problematic for large knowledge bases)

Memory Bidimensional Arrays.

 Data is stored in RAM memory, therefore the modification of rule and premises status is efficient.

 No duplication of data is needed. At the beginning of each session an empty set of arrays will be generated to store data generated during the inference process.

 C++ Builder facilitates the manipulation of bidimensional arrays of strings by providing the class TStringGrid.

 It is not homogenous with the rule representation in the knowledge base.

 It is not possible to execute SQL queries over the data stored in the arrays.

The approach chosen was using memory arrays because of the enormous loss of efficiency that continuous hard-drive writings and reading involve. The ideal data structure would have been a SQL compatible data structure stored in RAM memory.

6.1. Information before starting the inference process.

Before starting the inference process, we optionally can supply initial facts to the inference engine, as well as specifying the Threshold Value (which will be discussed later) for the propagation of certainty factors in rules. In this way, we avoid most of the annoying typical expert system *Question-and-answer* interface. The inference engine will try to reach a conclusion with the facts given in the initial stage, and only if it can not, then it will ask the user for the value of an specific attribute necessary to fill a *gap*.

6.2. Search Strategies.

The purpose of an expert system is to develop and recommend a proposed solution to a given problem. To accomplish this task, the expert system must conduct a *search* for the solution; and it is the responsibility of the inference engine in particular to perform this search in an efficient and effective manner.

In the search process we are faced with a number of alternatives (i.e. potential solutions) and a variety of constraints. For example, when faced with the problem of determining just what automobile to purchase, our alternatives theoretically include literally all of the different automobiles in existence. However, we are also faced with certain constraints, such as price limitations, style, availability… Such constraints serve to filter out the number of potential automobiles from which we will make our selection.

The search strategy implied in the selection of a car may be described in more technical terms as a *forward chaining* process. That is, we begin the process with certain data concerning the type of automobile desired, its style, cost, speed, mileage, and so on. These data, along with our constraints, serve to filter out the majority of the potential alternatives and thus we ultimately arrive at only a few automobiles from which we make our final selection.

We could approach the automobile selection problem from an entirely different direction by first specifying a particular car for purchase, and the determining whether or not it meets our needs. When using this approach, we are said to be employing *backward chaining*.

The two fundamental search strategies by an expert system (specifically the inference engine) are then forward and backward chaining. Forward chaining proceeds from premises (or data) to conclusions, and is said to be *data driven* (Turban, 1992). Backward chaining proceeds from a conclusion (hypothesis), and then seeks evidence that supports that conclusion. Backward chaining is often called a *goal driven* approach and proceed from the right of a rule to the left[5].

Ultimately, the conclusion reached by both approaches must be the same, but their search efficiency is dependent on the nature of the problem faced. Specifically, if one has a few premises and many conclusions, then forward chaining is generally the best search

[5] E.g. Prolog inference engine works backward chaining.

strategy. Otherwise, with many premises and relatively few conclusions, we should normally employ backward chaining.

In the Project, both forward and backward algorithms were implemented and it is up to the user at the beginning of the consultation session to choose which approach to use. Normally, in a commercial expert system just one approach is implemented. However, the implementation of bot of them allow us to compare and give us a better understanding of the different search methods available during the inference process.

6.3. The Forward Chaining Algorithm.

The implementation of the forward chaining algorithm can be described as a sequence of eight steps (Ignizio, 1991).

1. **Initialization.**
   Establish three empty tables: the Working Memory table, the Attribute-Queue table, and the Rule/Premise Status table (that it is, in fact, the joining of the premise and conclusions tables by the field "RID").

   The Working memory table will be used to record all assertions (i.e., all the facts deduced during the consultation). The Attribute-Queue table records, in order, all attributes for which a value has been assigned or is being sought. The attribute at the top of this table is the attribute presently under consideration. The Rule/Premise Status table records the rule status (i.e. Active "A", Discarded "D", Triggered "T", or Fired "F") as well as the status of each premise clause (i.e. Active "A", False "F", or True "T"). All active premises clauses and rules are initially loaded in the Rule/Premise Status table.

   Structure:

```
// define structure Attribute-Value tuple
struct AV_tuple{
    char attribute[40];
    char value [40];
    float cf;
};
struct Attribute{
    char attribute[40];
     int RID;
};
AV_tuple * WorkMem_Table; // Working Memory Table
Attribute * AttQueue_Table; // Attribute-Queue Table

(…)
PremisesTb->Open();
ConclTb->Open();

ResetQuery(Query1, "select * from conclusion where status=\"A\"");
LoadGrid(Query1,ConclusionsGrid);

ResetQuery(Query1,"select premises.RID,
    premises.PID,premises.Attribute,premises.Val,premises.Status,premises.
    CF from premises,conclusion where conclusion.status=\"A\" and
    premises.RID=conclusion.RID");
LoadGrid(Query1,PremisesGrid);
```

2. **Start Inference.**

   - If the user gave no entries at the beginning of the consultation, ask him an initial fact. Record this attribute at the top of the Attribute-Queue table. Also, record this attribute and its associated value at the bottom of the Working Memory table.

- Otherwise, take the last fact given by the user at the beginning of the consultation, record the attribute at the top of the Attribute-Queue table. Also, record the attribute and its associated value at the bottom of the Working Memory table.

3. **Rule scan.**
   Examine the Rule/Premise Status table. If no rules are active, STOP. Otherwise, scan the active rule-set premise clauses for all occurrences of the attribute on the top of Attribute-Queue table, and record any changes in status of the premise clauses of the active rule set.

   (a) If the premise[6] of any rule is false, discard that the rule.

   (b) If the premise of any rule is true, then mark the associated rule as being triggered. Place its conclusion attribute and RID at the bottom of the Attribute-Queue table.

   (c) If no rules are presently in the triggered state (i.e. via a check of the Rule/Premise Status table, go to STEP 5. Otherwise, go to STEP 4.

4. **Rule firing.**
   Cross out the topmost attribute on the Attribute-Queue table. Change the status of the rule associated with the new topmost attribute from triggered to fired. Place conclusion at the bottom of the Working-Memory Table.

5. **Queue Status.**
   Cross out the topmost attribute on the Attribute-Queue table. Proceed to STEP 6.

6. **Convergence Check.**
   Scan the Rule/Premise Status table for any active rule. If no such rules are found STOP. Otherwise Proceed to STEP 7.

7. **Query.**
   - If there are still initial facts given by the user that were not used, take one.
   - Otherwise, query the user for the value of an attribute in any of the rule's free premise clauses.

   Go to STEP 8.

8. **Updating.**
   Place the associated attribute (and rule number) on the top of the Attribute-Queue table. Also, place conclusion at the bottom of the Working-Memory Table.
   Return to STEP 3.

[6] By "premise", I refer to the entire premise of the rule, which may be composed of several premise clauses.

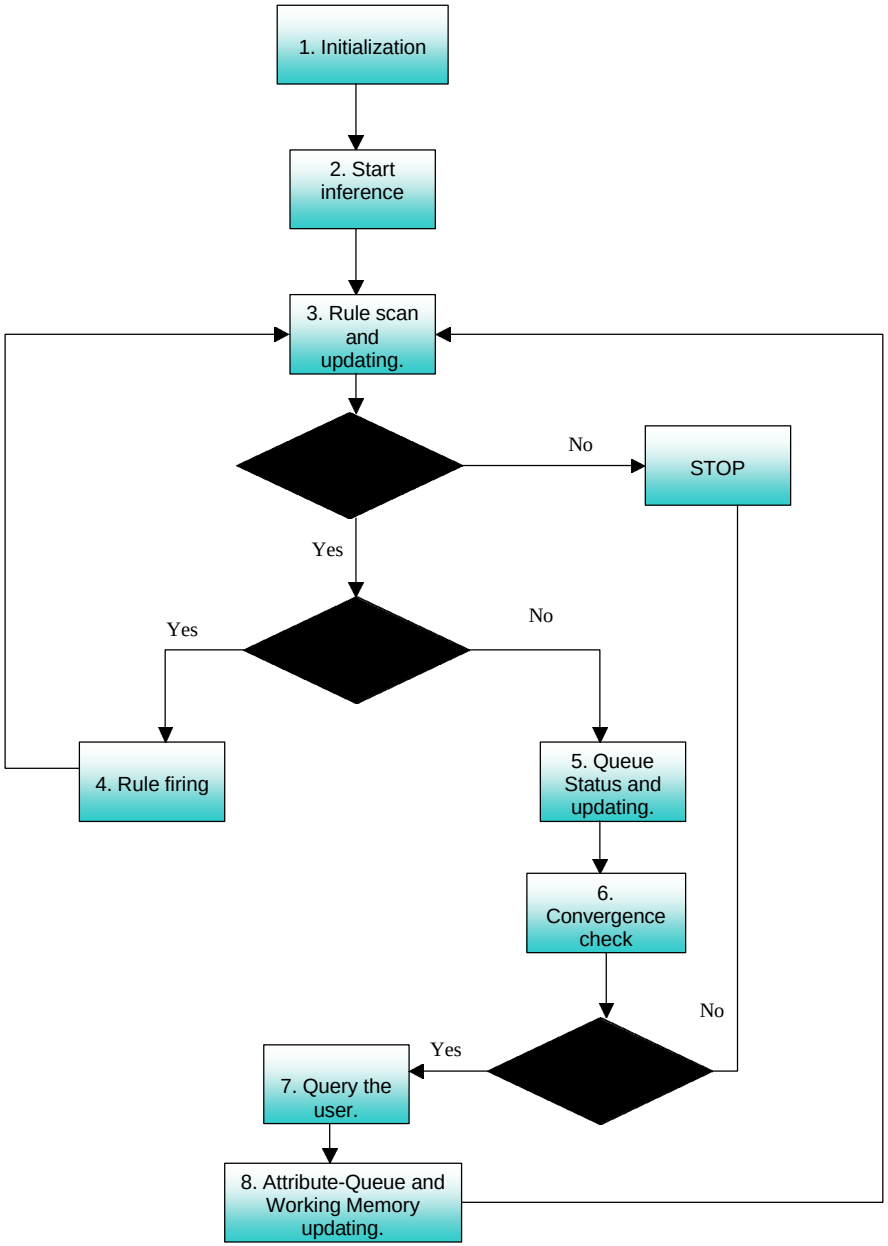


Fig.3. Forward Chaining Algorithm flow chart.

The implementation of the forward chaining algorithm in C++ was written as follows:

```
/*************************************************************************/
/*************************************************************************/
/**                    FORWARD CHAINING ALGORITHM                       **/
/*************************************************************************/
/*************************************************************************/

//---------------------------------------------------------------------------
//void __fastcall TInferenceForm::FwdButtonClick(TObject *Sender)
//---------------------------------------------------------------------------
void __fastcall TInferenceForm::FwdButtonClick(TObject *Sender)
{
    TMsgDlgButtons Btns;
    char error[256];

       CleanGrids(); // empty results from former queries
       MainForm->ThresholdValue=atoi(Edit2->Text.c_str());
       PageControl1->SelectNextPage(true);
       ForwardChaining();

       if (RulesGrid->Cells[0][1]==""){ // no conclusions found
       // check certainty factor
              Btns<<mbOK;
              wsprintf(error, "No conclusions found");
              MessageDlg(error,mtError,Btns,NULL);
              return;
       }
       else{
           ConclBtn->Enabled=true;
       }
}
//---------------------------------------------------------------------------
//void __fastcall TInferenceForm::ForwardChaining(void)
//           Forward Chaining algorithm. Consists of 8 steps.
//---------------------------------------------------------------------------
void __fastcall TInferenceForm::ForwardChaining(void)
{
// STEP 1: Initialisations
    AnsiString Val;
    int position;

    PremisesTb->Open();
    ConclTb->Open();

    ResetQuery(Query1, "select * from conclusion where status=\"A\"");
    LoadGrid(Query1,ConclusionsGrid);

    ResetQuery(Query1,"select premises.RID,
    premises.PID,premises.Attribute,premises.Val,premises.Status,premises.CF
    frompremises,conclusion whereconclusion.status=\"A\" and
    premises.RID=conclusion.RID");
    LoadGrid(Query1,PremisesGrid);

    aq_ind=0; // index for Attribute-queue table
    top_most=0; // index for Attribute-queue table
    wm_ind=0; // index for Working-memory table

// STEP 2: Start Inference
    if(CurrentRow==0){//if the user gave no entries, we need an initial fact
       PremisesTb->First(); //get first entry
       Val= MainForm->Query(PremisesTbAttribute->Value);

       // save results
       strcpy(WorkMem_Table[wm_ind].attribute,PremisesTbAttribute-
       >Value.c_str());
       strcpy(WorkMem_Table[wm_ind].value,Val.c_str());
```



```
       WorkMem_Table[wm_ind].cf=MainForm->CF;
       FactsGrid->Cells[0][wm_ind]=PremisesTbAttribute->Value;
       FactsGrid->Cells[1][wm_ind]=Val;
       FactsGrid->Cells[2][wm_ind]=(AnsiString) MainForm->CF;
       wm_ind++;

       strcpy(AttQueue_Table[aq_ind].attribute,PremisesTbAttribute-
       >Value.c_str());
       AttQueue_Table[aq_ind].RID=PremisesTbRID->Value;
       aq_ind++;
    }
    else{ // if the user gave entries, take as the initial fact
       strcpy(WorkMem_Table[wm_ind].attribute,InitialFactsGrid-
       >Cells[0][CurrentRow-1].c_str());

       strcpy(WorkMem_Table[wm_ind].value,InitialFactsGrid-
       >Cells[1][CurrentRow-1].c_str());

       WorkMem_Table[wm_ind].cf=atof(InitialFactsGrid->Cells[1][CurrentRow-
       1].c_str());
       MainForm->CF=WorkMem_Table[wm_ind].cf;
       FactsGrid->Cells[0][wm_ind]=InitialFactsGrid->Cells[0][CurrentRow-1];
       FactsGrid->Cells[1][wm_ind]=InitialFactsGrid->Cells[1][CurrentRow-1];
       FactsGrid->Cells[2][wm_ind]=InitialFactsGrid->Cells[2][CurrentRow-1];
       wm_ind++;

       strcpy(AttQueue_Table[aq_ind].attribute,InitialFactsGrid-
       >Cells[0][CurrentRow-1].c_str());
       AttQueue_Table[aq_ind].RID=-1; // it does not come from a rule, but
       from the user
       aq_ind++;
       CurrentRow--;
    }

// STEP 3: Rule scan and check for convergence
STEP3: if (ForwardChainingStep3()){
              if (CheckField(ConclusionsGrid, 3, "T")==-1){// check for
       triggered rules
                     ForwardChainingStep5(); // no triggered rules
                     position=ForwardChainingStep6();
                     if (position!= -1){// convergence check. True if there
                     are active rules still.
                            ForwardChainingStep7(position); // Query value
                            goto STEP3;
                     }
              }
              else{
                     ForwardChainingStep4(); // there are triggered rules
                     goto STEP3;
              }
    }
    // else End
}

//---------------------------------------------------------------------------
//bool __fastcall TInferenceForm::ForwardChainingStep3(void)
// Forward Chaining algorithm. 3rd. step. If there are no active rules or
// triggered rules returns FALSE.
//---------------------------------------------------------------------------
bool __fastcall TInferenceForm::ForwardChainingStep3(void)
{
       char TopAttribute[40];
    char AttrVal[40];
    int i;

       if (CheckField(ConclusionsGrid, 3, "A")==-1){ // check for active
rules
       if (CheckField(ConclusionsGrid, 3, "T")==-1) // check for active rules
```

```
       return false; // no active or triggered rules
        else return true;
    }
       strcpy(TopAttribute,AttQueue_Table[top_most].attribute);
       // top of Attribute queue table

    // update state of premises and conclusions
    for (i=0; i<wm_ind;i++){
        if(strcmp(WorkMem_Table[i].attribute,TopAttribute)==0)
             strcpy(AttrVal,WorkMem_Table[i].value);
       }
       UpdateRuleSet((AnsiString)TopAttribute,(AnsiString)AttrVal,
       PremisesGrid,ConclusionsGrid);

       return true; // there were active rules
}

//---------------------------------------------------------------------------
//void__fastcallTInferenceForm::ForwardChainingStep4(void)
// Forward Chaining algorithm. 4th. step. Rule firing
//---------------------------------------------------------------------------
void__fastcallTInferenceForm::ForwardChainingStep4(void)
{
    int position;

       // cross topmost attribute on Attribute-Queue Table
       top_most++;

       // change rule associated with topmost attribute from triggered to
       // fired
       position=UpdateGrid((AnsiString)AttQueue_Table[top_most].RID,"F",
       ConclusionsGrid); // change conclusion state to "fired"

       // place conclusion at the bottom of the Working-Memory Table
       strcpy(WorkMem_Table[wm_ind].attribute,ConclusionsGrid-
       >Cells[1][position].c_str());

       strcpy(WorkMem_Table[wm_ind].value,ConclusionsGrid-
       >Cells[2][position].c_str());
       MainForm->CF=WorkMem_Table[wm_ind].cf;

       WorkMem_Table[wm_ind].cf = atof (ConclusionsGrid-
       >Cells[4][position].c_str());

       // show rule fired in Grid
       ShowRule(Query1,(AnsiString)AttQueue_Table[top_most].RID,
       RulesGrid);

       FactsGrid->Cells[0][wm_ind]=ConclusionsGrid->Cells[1][position];
       FactsGrid->Cells[1][wm_ind]=ConclusionsGrid->Cells[2][position];
       FactsGrid->Cells[2][wm_ind]=ConclusionsGrid->Cells[4][position];
       wm_ind++;
}

//---------------------------------------------------------------------------
//void__fastcallTInferenceForm::ForwardChainingStep5(void)
// Forward Chaining algorithm. 5th. step. Queue status.
//---------------------------------------------------------------------------
void__fastcallTInferenceForm::ForwardChainingStep5(void)
{
    // cross topmost attribute on Attribute-Queue Table
       top_most++;
}

//---------------------------------------------------------------------------
//int__fastcallTInferenceForm::ForwardChainingStep6(void)
//Forward Chaining algorithm. 6th. step. Convergence check and rule marking.
//---------------------------------------------------------------------------
```



```
int__fastcallTInferenceForm::ForwardChainingStep6(void)
{
       return (CheckField(ConclusionsGrid, 3, "A"));
}

//---------------------------------------------------------------------------
// void __fastcall TInferenceForm::ForwardChainingStep7(intposition)
// Forward Chaining algorithm. 7th. step. Query.
//---------------------------------------------------------------------------
void__fastcallTInferenceForm::ForwardChainingStep7(int position)
{
       int premise;
    AnsiString value;

    if(CurrentRow>0){ // there are still facts given by the user
          // place attribute of premise in position in the Grid on the top
          // of the Attribute-queue table
          if (top_most !=0){
             top_most--;
             strcpy(AttQueue_Table[top_most].attribute,InitialFactsGrid-
             >Cells[0][CurrentRow-1].c_str());
             AttQueue_Table[top_most].RID=-1;
          }
          else{
             MoveAQTable();
             strcpy(AttQueue_Table[top_most].attribute,InitialFactsGrid-
             >Cells[0][CurrentRow-1].c_str());
             AttQueue_Table[top_most].RID=-1;
          }

// place attribute plus its value at the bottom of the Working-Memory Table
       FactsGrid->Cells[0][wm_ind]=InitialFactsGrid->Cells[0][CurrentRow-1];
       FactsGrid->Cells[1][wm_ind]=InitialFactsGrid->Cells[1][CurrentRow-1];
       FactsGrid->Cells[2][wm_ind]=InitialFactsGrid->Cells[2][CurrentRow-1];

       strcpy(WorkMem_Table[wm_ind].attribute,FactsGrid-
       >Cells[0][wm_ind].c_str());

       strcpy(WorkMem_Table[wm_ind].value,FactsGrid-
       >Cells[1][wm_ind].c_str());

       WorkMem_Table[wm_ind].cf = atof(FactsGrid->Cells[2][wm_ind].c_str());
       MainForm->CF=WorkMem_Table[wm_ind].cf;
       wm_ind++;
       CurrentRow--;
    }

    else{
       premise=GetFreePremise(ConclusionsGrid->Cells[0][position],
       PremisesGrid);

       value= MainForm->Query(PremisesGrid->Cells[2][premise]);
       ForwardChainingStep8(premise,PremisesGrid,value);
    }
}

//---------------------------------------------------------------------------
//void__fastcallTInferenceForm::ForwardChainingStep8
//               (int position, TStringGrid *Grid, AnsiString Value)
//Forward Chaining algorithm. 8th. step. Update A-Q table and Working Memory
// Table.
//---------------------------------------------------------------------------
void__fastcallTInferenceForm::ForwardChainingStep8(int position,
TStringGrid *Grid, AnsiString Value)
{
    // place attribute of premise in position in the Grid on the top of the
    // Attribute-queue table
    if (top_most !=0){
```

```
      top_most--;
      strcpy(AttQueue_Table[top_most].attribute,Grid-
      >Cells[2][position].c_str());
      AttQueue_Table[top_most].RID=atoi(Grid->Cells[0][position].c_str());
    }
    else{
      MoveAQTable();
      strcpy(AttQueue_Table[top_most].attribute,Grid-
      >Cells[2][position].c_str());
      AttQueue_Table[top_most].RID=atoi(Grid->Cells[0][position].c_str());
    }

  //place attribute plus its value at the bottom of the Working-Memory Table
  strcpy(WorkMem_Table[wm_ind].attribute,Grid->Cells[2][position].c_str());
  strcpy(WorkMem_Table[wm_ind].value,Value.c_str());
  WorkMem_Table[wm_ind].cf = MainForm->CF;

  FactsGrid->Cells[0][wm_ind]=Grid->Cells[2][position];
  FactsGrid->Cells[1][wm_ind]=Value;
  FactsGrid->Cells[2][wm_ind]=(AnsiString)MainForm->CF;
  wm_ind++;
}
```

6.4. The Backward Chaining Algorithm.

The implementation of the backward chaining algorithm can be described as a sequence of six steps (Ignizio, 1991).

1. **Initialization.**
   Establish three empty tables, the Working Memory Table, the Goal Table, and the Rule/Premise Status table. The Working Memory Table and the Rule/Premise Status table do the same function as in the forward chaining algorithm. The goal table records, in order, those attributes for which a values is sought.

2. **Start inference.**
   Save facts given by the user at the beginning and update Rule/Premise Status table. Specify a final goal (i.e., a conclusion clause attribute). Place the associated goal attribute at the top of the Goal table.

3. **Rule scan.**
   Scan the conclusion clauses of the active rules (i.e., rules that have not yet been fired or discarded) to find any occurrence of the goal attribute presently on the top of the Goal table.

   (a) If the Goal table is empty, STOP

   (b) If only one such rule may be found, go to step 6. If several such rules may be found, and any of these are triggered, select any one of the triggered rules and proceed to step 6. Otherwise, select one rule from among the rules found that contains the subject goal attribute in its conclusion clause set, and go to step 6.

   (c) If no active rules are found that contain the subject goal attribute in their conclusion clause set, then go to step 4.

4. **Query.**
   Query the user for the goal attribute on top of the Goal Table, record his or her response, remove the top goal attribute from the Goal table and place it, plus its value (as supplied by the user), in the Working Memory table. Go to step 5.

5. **Rule/premise status update.**
   Using the contents of the Working Memory table, update the Rule/Premise Status table. Specifically, if the premise of any rule is false, discard that rule, and if the premise is true, trigger that rule. Return to step 3.

6. **Rule evaluation.**
   For the rule found in step 3:

   (a) If this rule is triggered, then remove the current topmost goal attribute from the Goal table and place it, plus its value, in the Working Memory table. Change the status of this rule from triggered to fired. Go to step 5. Otherwise (i.e., if this rule is not triggered) proceed to step 6b, below.

   (b) If this rule is not triggered, then select the first unknown premise attribute of the rule and place it, plus the rule number, at the top of the Goal table. Return to step 3.

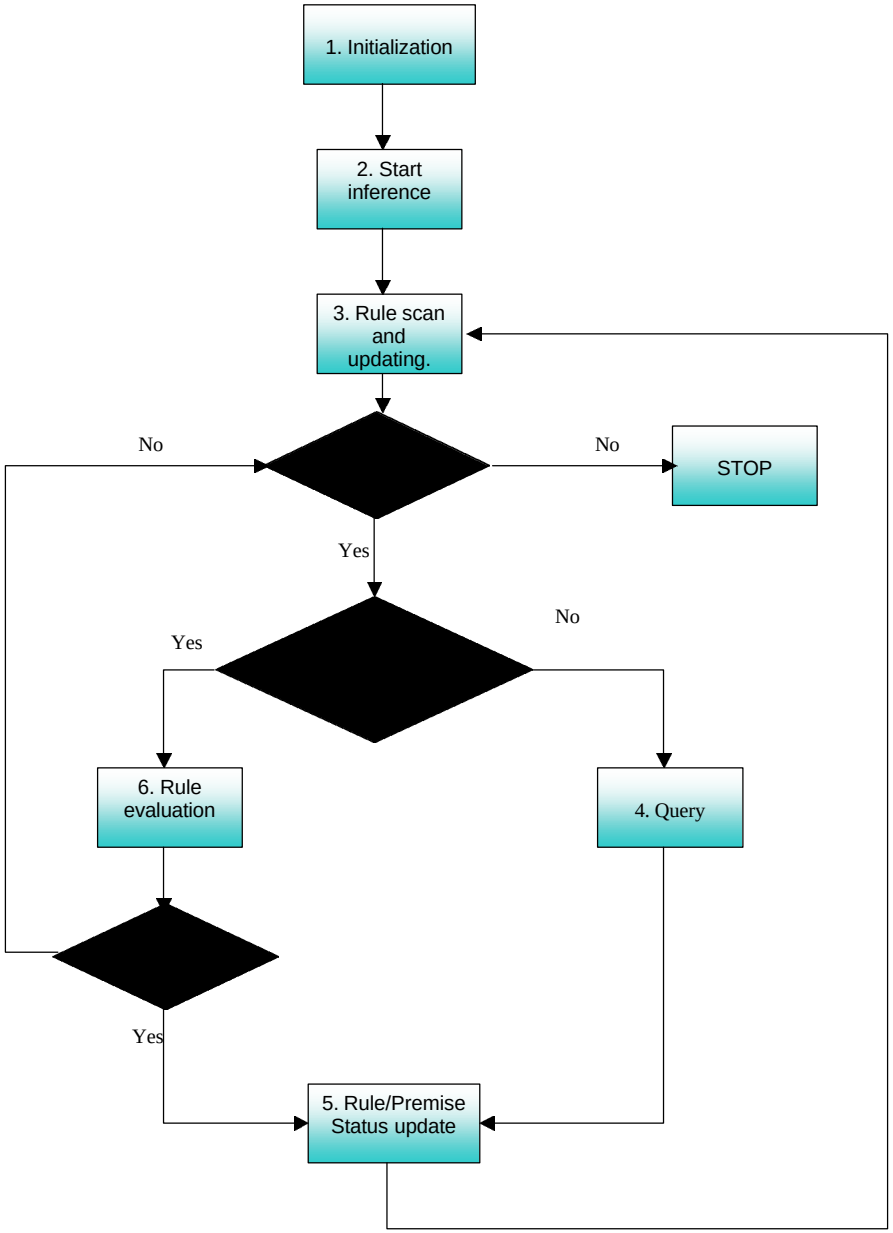


Fig.4. Backward Chaining Algorithm flow chart.

Source code of the backward chaining algorithm:

```
/*****************************************************************************/
/*****************************************************************************/
/**                    BACKWARD CHAINING ALGORITHM                          **/
/*****************************************************************************/
/*****************************************************************************/
//---------------------------------------------------------------------------
// void __fastcall TInferenceForm::BackButtonClick(TObject *Sender)
//---------------------------------------------------------------------------
void __fastcall TInferenceForm::BackButtonClick(TObject*Sender)
{
       CleanGrids(); // empty results from former queries
       PageControl1->SelectNextPage(true);
       MainForm->ThresholdValue=atoi(Edit2->Text.c_str());
       BackwardChaining();
       ConclBtn->Enabled=true;
}

//---------------------------------------------------------------------------
// void __fastcall TInferenceForm::BackwardChaining(void)
//           Forward Chaining algorithm. Consists of 6 steps.
//---------------------------------------------------------------------------
void __fastcallTInferenceForm::BackwardChaining(void)
{
// STEP 1: Initialisations
    AnsiString Val;
    int position, i;
    PremisesTb->Open();
    ConclTb->Open();
    bool iterate;

    ResetQuery(Query1, "select * from conclusion where status=\"A\"");
    LoadGrid(Query1,ConclusionsGrid);

    ResetQuery(Query1, "select premises.RID,
    premises.PID,premises.Attribute,premises.Val,premises.Status,premises.CF
    from premises,conclusion where conclusion.status=\"A\" and
    premises.RID=conclusion.RID");

    LoadGrid(Query1, PremisesGrid);

    goal_ind=0; // index for Goal table
    top_goal=0; // index for top of Goal table
    wm_ind=0; // index for Working-memory table

    // copy contents of initial facts given by the user in the Facts grid
    CopyGrid(InitialFactsGrid, FactsGrid);

    // STEP 2: Start Inference
    // save results given by the user in the initial facts and update rules
    for(i=0;i<CurrentRow;i++){
             MainForm->CF=atof(InitialFactsGrid->Cells[0][i].c_str());
             UpdateRuleSet(InitialFactsGrid->Cells[0][i],InitialFactsGrid-
             >Cells[1][i],PremisesGrid,ConclusionsGrid);
    }

    ConclTb->First(); //get first entry
    // save results
    strcpy(Goal_Table[top_goal].attribute,ConclTbAttribute->Value.c_str());
    Goal_Table[top_goal].RID=ConclTbRID->Value;
    goal_ind++;

  // STEP 3: Rule scan and check for convergence
  //scan active rules for occurences of the attribute in the top of the goal
  // table, giving prority to triggered rules
   do{
          iterate=BackwardChainingStep3();
```

```
    }while(iterate);

}

//---------------------------------------------------------------------------
//bool __fastcallTInferenceForm::BackwardChainingStep3(void)
// Backward Chaining algorithm. 3rd. step. Returns false if the goal
// table is empty
//---------------------------------------------------------------------------
bool__fastcallTInferenceForm::BackwardChainingStep3(void)
{
      int position;

      if (top_goal==goal_ind){ // if both indexes are equal => the goal
      table is empty
            return false;
      }
      else{
            position=GetRule(ConclusionsGrid,(AnsiString)
            Goal_Table[top_goal].attribute);
            if (position != -1){
                  BackwardChainingStep6(position);
            }
            else{
                  BackwardChainingStep4();
                  BackwardChainingStep5();
            }
       return true;
      }
}

//---------------------------------------------------------------------------
//void __fastcallTInferenceForm::BackwardChainingStep4(void)
// Backward Chaining algorithm. 4th. step. Query.
//---------------------------------------------------------------------------
void__fastcallTInferenceForm::BackwardChainingStep4(void)
{
    AnsiString value;

    value= MainForm->Query((AnsiString)Goal_Table[top_goal].attribute);

    // place topmost attribute plus its value on the Working Memory Table
    strcpy(WorkMem_Table[wm_ind].attribute,Goal_Table[top_goal].attribute);
    strcpy(WorkMem_Table[wm_ind].value,value.c_str());
    WorkMem_Table[wm_ind].cf = MainForm->CF;

    FactsGrid->Cells[0][wm_ind]=(AnsiString)Goal_Table[top_goal].attribute;
    FactsGrid->Cells[1][wm_ind]=value;
    FactsGrid->Cells[2][wm_ind]=(AnsiString)MainForm->CF;
    wm_ind++;

    // cross topmost attribute on Goal Table
    top_goal++;
}

//---------------------------------------------------------------------------
// void __fastcallTInferenceForm::BackwardChainingStep5(void)
// Backward Chaining algorithm. 5th. step. Rule/premise status update.
//---------------------------------------------------------------------------
void__fastcallTInferenceForm::BackwardChainingStep5(void)
{
      UpdateRuleSet((AnsiString)WorkMem_Table[wm_ind-1].attribute,
      (AnsiString)WorkMem_Table[wm_ind-1].value,PremisesGrid,
      ConclusionsGrid);
}

//---------------------------------------------------------------------------
// void __fastcall TInferenceForm::BackwardChainingStep6(intposition)
```



```
// Backward Chaining algorithm. 6th. step. Rule evaluation.
//---------------------------------------------------------------------------
void__fastcallTInferenceForm::BackwardChainingStep6(intposition)
{
      int premise_nr;

      if(ConclusionsGrid->Cells[3][position]=="T"){
      // if rule is triggered => fire it
      // place topmost attribute plus its value on the Working Memory Table
            strcpy(WorkMem_Table[wm_ind].attribute,ConclusionsGrid-
            >Cells[1][position].c_str());
            strcpy(WorkMem_Table[wm_ind].value,ConclusionsGrid-
            >Cells[2][position].c_str());
            WorkMem_Table[wm_ind].cf = MainForm->CF;

            // show rule fired in Grid
            ShowRule(Query1,ConclusionsGrid->Cells[0][position],
            RulesGrid);

            FactsGrid->Cells[0][wm_ind]=(AnsiString)
            Goal_Table[top_goal].attribute;
            FactsGrid->Cells[1][wm_ind]=ConclusionsGrid-
            >Cells[2][position].c_str();
            FactsGrid->Cells[2][wm_ind]=(AnsiString)MainForm->CF;
            wm_ind++;

      // cross topmost attribute on Goal Table
            top_goal++;

       // update rule-premise table
            ConclusionsGrid->Cells[3][position]="F";//fired
      }
      else{
            // select first unknown premise attribute of the rule
            premise_nr=GetFreePremise(ConclusionsGrid->Cells[0][position],
            PremisesGrid);

            // place it at the top of the Goal table
            if (top_goal !=0){
                  top_goal--;
                  strcpy(Goal_Table[top_goal].attribute,PremisesGrid-
                  >Cells[2][premise_nr].c_str());
                  Goal_Table[top_goal].RID=atoi(PremisesGrid-
                  >Cells[0][premise_nr].c_str());
            }
            else{
                  MoveGoalTable();
                  Strcpy(Goal_Table[top_goal].attribute,PremisesGrid-
                  >Cells[2][premise_nr].c_str());
                  Goal_Table[top_goal].RID=atoi(PremisesGrid-
                  >Cells[0][premise_nr].c_str());
            }
      }
}
```

7. Enhancements.

7.1. Uncertainty.

Up until now, our focus has been on deterministic rules. However, as the structure of the database reveals, the system also provides support for uncertainty. The real world in fact is characterised by uncertainty and it is desirable that this characteristic is available in any expert system.

In general, there are two primary sources of uncertainty that may be encountered in the expert system:

- Uncertainty with regard to the validity of a knowledge base rule.
- Uncertainty with regard to the validity of a user response.

The method employed to represent uncertainty is using Certainty Factors. There are several methods of using certainty factors in handling uncertainty in knowledge-based systems. We use a numeric value associated with each rule and each premise that ranges from 0 to 100 (Turban, 1992). 0 represents absolute certainty that a rule or premise is false, while 100 represents absolute certainty that the rule/premise is true. Note that certainty factors are not probabilities.

The propagation of certainty factors is made following the approach used in MYCIN:

- The certainty factor of the whole premise is the minimum of the CFs of each premise.

E.g.: IF sky_color = grey (CF = 70)
AND month = April (CF = 60)
THEN weather = rainy.

Thus, the CF of the premise will be the minimum of the two, that is 60.

- The certainty factor of the conclusion will be calculated by multiplying the actual value of the conclusion CF by the CF of the premise.

CF(rule)' = (CF(rule) * CF(premise))/100

In the application, the certainty factor of the premise (IF part) is calculated by storing in a variable the temporal minimum of the premises' certainty factors while we update the status of the premises:

The fragment of code that calculates the accumulated CF of the premises is:

```
float TemporaryCF=100;

for (i=0; Grid->Cells[0][i] !="EOF"; i++){
      if (Grid->Cells[0][i]==RID){
            if (Grid->Cells[4][i]=="F"){ // mark the rest of premises in
      the rule        as false to increase efficiency
            mark=true;
           break;
            }
            else{
```



```
                  if (Grid->Cells[4][i]=="A")
                        break;
                  else{ // true premise => calculate accumulated CF
                        if (atoi (Grid->Cells[5][i].c_str())<(int)
                        TemporaryCF){ // new minimum

                              TemporaryCF=atof(Grid-
                        >Cells[5][i].c_str());
                        }
                  }
            }
      }
} //end of for loop
```

The calculation of the certainty factor of the conclusion is calculated by the function "CalcCF":

```
//---------------------------------------------------------------------------
---
// float CalcCF(float preconditionCF, int Position, TStringGrid *Grid)
// Calculates the Certainty Factor of the rule given the CF of the premises
//---------------------------------------------------------------------------
---
float CalcCF(float preconditionCF, int Position, TStringGrid *Grid)
{
      float conclusionCF, CF;
      char cf[5];
      strcpy(cf, Grid->Cells[4][Position].c_str());
      conclusionCF=atof(cf);
      CF=(conclusionCF*preconditionCF)/100;
      // New CF = (Premise CF * Conclusion CF)/100

      if (CF<MainForm->ThresholdValue){
            return 0; // if it is under the Threshold Value, return 0 as
the C.F.
      }
      else{
            MainForm->CF=CF;
            return (CF);
      }
}
```

7.2. The Threshold Value.

In the function above, note that we use the variable "ThresholdValue". The threshold level was use in MYCIN also, but in that expert system it was a static value (0.2 in a scale from –1 to 1). In this implementation, the user can change the value of the *threshold level* in each consultation. The threshold level discards rules under a certain value of certainty.

7.3. Explanation Facility.

At the end of a consultation, the system is able to justify why a particular solution was reached. The explanation consists in keeping track of the facts inferred and the rules applied and present them to the user at the end of the consultation, if he or she requires them.

In the following figure a particular explanation is showed.

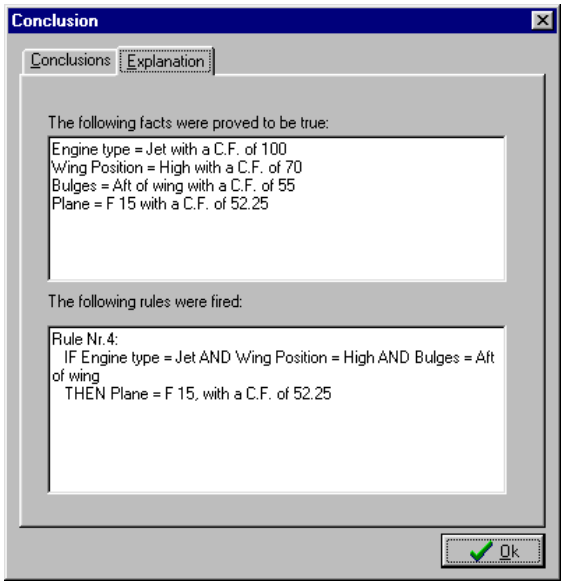


Fig.5. Explanation facility provided by the expert system shell.

8. Testing.

The system was tested during all the design process. Most of the times, the method that I have found more useful and effective is printing the code on paper and tracing the program manually to see what portions of code could be improved or where is a specific flaw of the application.

In fact, for any programmer who reads the code for forward or backward chaining algorithms it would be difficult to know how the algorithms work without using a particular example and apply the algorithm to it step-by-step.

Obviously, the debugging facilities provided by modern software development tools such as C++ Builder also help in the task of testing and debugging the code, specially for small errors in assignations or stopping conditions in loops. A special feature that I really appreciate in C++ Builder (available in most of other visual programming packages) is the inspection of objects as a whole, being able to watch/modify (almost) any property of the object inspected.

User Manual

1. Installation.

To install the Expert System Shell application, simply run the “setup.exe” application from Windows’95 or Windows’98 and follow the steps indicated in the screen (E.g. destination directory). The interface of the set-up application is the standard provided with almost every MS Windows application.

2. Running the Application.

The application may be run using the common steps for every MS Windows application, i.e., either by double-clicking the icon of the application in Windows Explorer, or typing the full path in the “Run” option of the “Start” menu (Installation directory + expert.exe).

3. Start menu of the Expert System.

As in other applications (such as MS PowerPoint), when we run the application we get an initial window which guide us through the main options, i.e. editing the knowledge base, or starting a new consultation. Also the user can simply start the application by clicking “Cancel” or selecting the radio button of “Start Expert System Shell”.

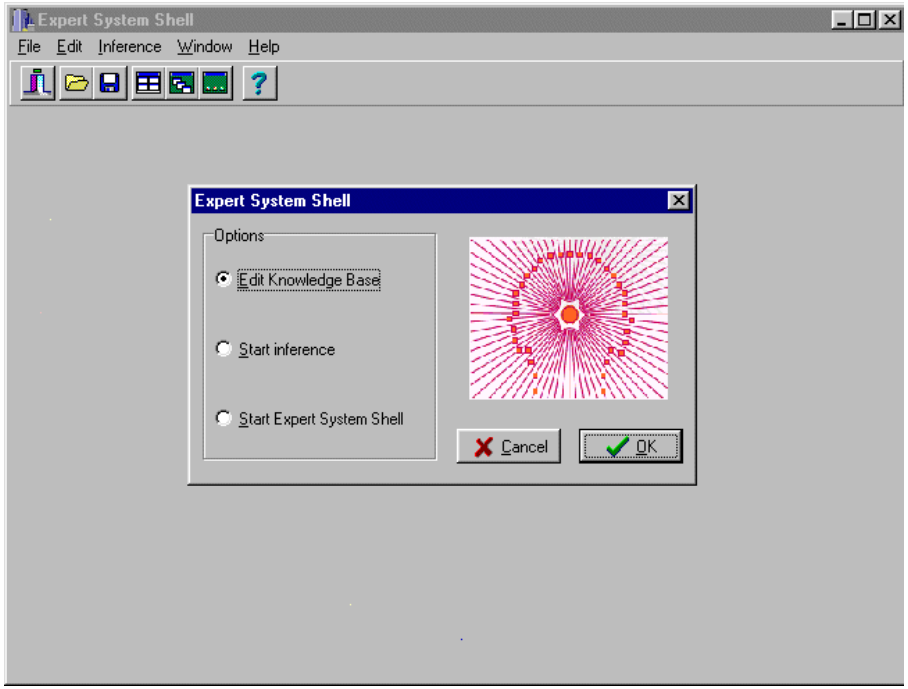


Fig.6. Start menu of the application.

4. Editing the knowledge base.

Before we can edit the knowledge base, we must supply the application with the path where the knowledge base is stored. Then we get a form with all the rules of the knowledge base in a grid. We can edit, delete or add rules from/to the knowledge base.

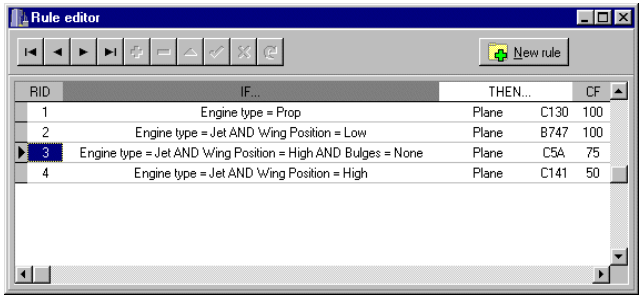


Fig.7. Rule viewer.

In order to edit an existing rule we just have to double-click on the desired rule. The form for adding/editing rules is showed:

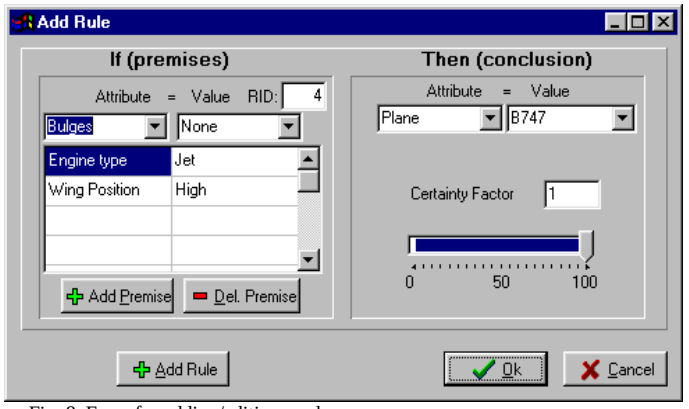


Fig. 8. Form for adding/editing a rule.

In the left side of the form, we can add or delete premises to/from the rule. The combo-boxes are automatically loaded with the possible values for each attribute. The RID number is automatically generated by the application, although we can change manually if there is not any rule with the same RID.

In the right side of the form, we specify the conclusion of the rule and associate a certainty factor to the rule, either using the edit box or the track bar. When we click “Add Rule” or “Ok” the rule is posted (after checking that the fields were filled correctly) and the Rule Viewer form is updated. The difference between the two buttons is that “Ok” posts the rule and closes the window and “Add Rule” just posts the rule.

5. Inference with the Expert System Shell.

To start a consultation session with the expert system shell we select “Inference” from the menu bar and the “Start inference”. If we did not specify a knowledge base the system will ask us for the path of the knowledge base we want to consult.

We can provide data to the system either at the beginning of the consultation or during the consultation. Also, we must associate a certainty factor to each fact that we assert.

The threshold value can be different for each consultation, we can modify it using the track bar or the edit box in the same way we modify certainty factors. The value by default is 0, i.e. no rule is discarded if the certainty factor is low (obviously, since no rule can have a certainty factor under 0).

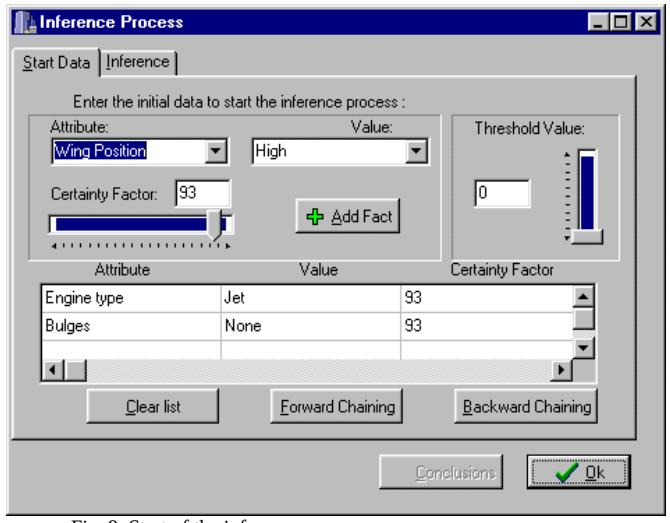


Fig. 9. Start of the inference process.

To start the inference engine we select the inference mechanism (forward or backward) by clicking the appropriate button. Then, the system will try to reach a conclusion from the facts given at the beginning (if any was given) and if it can not then it will ask the user for more information.

The form used to ask the user for a value for a given attribute is shown next.

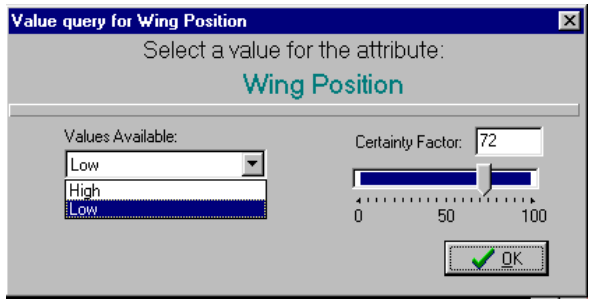


Fig. 10. Form for querying the user about a particular attribute.

During the inference process the expert system shows the known facts (either provided by the user or by inference) and rules that it is firing.

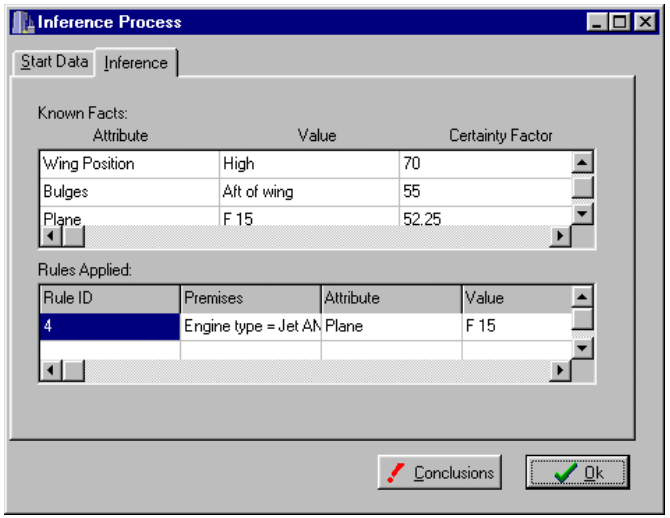


Fig.11. The inference process in action.

If the system reaches a conclusion, the conclusion button is enabled. If we click it the conclusion is shown in an individual form. That explanation facility for the conclusion reached (we just have to select the “tab” labelled “explanation”).

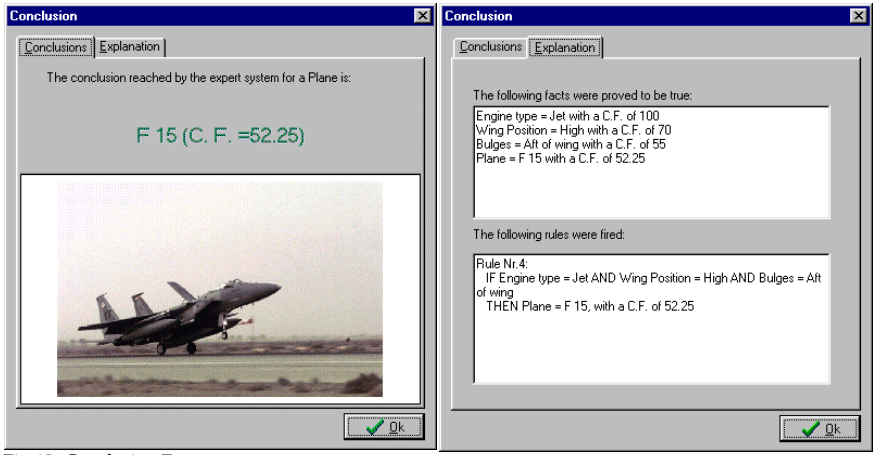


Fig.12. Conclusion Form.

If several conclusions are reached, the expert system will show consecutively each conclusion. We can repeat the inference changing some values and see how the conclusions reached by the expert system vary.

Appendix A: Source Code.

Next, the most relevant units of the expert system implementation are shown. The project is divided in several modules, which are connected as described in section 4 of the technical manual (see page 15). Normally, each file corresponds to a form in the application, except "UNIT.CPP" that is a repository of frequently used functions.

Bibliography.

- Cawsey, A. (1998) *The essence of Artificial Intelligence*. Prentice Hall.
- Covington, M., Nute, D., Vellino, E. (1997) *Prolog Programming in Depth.* Prentice Hall.
- Ignizio, J. (1991) *Introduction to Expert Systems*. Mc. Graw-Hill, Inc.
- Marquez, A. (1997) *Programaciòn con C++ Builder.* Anaya Multimedia.
- Newman, W.M., Lamming M.G. (1995) *Interactive Sytem Design.* Addison-Wesley.
- Quinlan, J.R. (1983) Learning Efficient Classification Procedures and Their Applications to Chess End Games. *Machine Learning: An Artificial Intelligence Approach.*
- Rich, E. (1983) *Artificial Intelligence*. McGraw-Hill.
- Simon, H.A. and Newell, A. (1958) Heuristic Problem Solving: The next advance in Operations Research. *Operation Research,* January-February 1958, pp.1-10
- Sommerville, I. (1992) *Software Engineering.* Addison-Wesley Publishing Company.
- Surko, P., Tips for Knowledge Acquisition, *PC AI*, May-June 1989. pp. 14-18.
- Turban, E. (1992) *Expert Systems And Applied Artificial Intelligence*. Prentice Hall.