SELECTING AN APPROPRIATE EXPERT SYSTEM TOOL[+]

6.1. **Introduction**

For a successful development of an expert system, a number of factors are responsible. Notably, selection of an appropriate expert system problem domain, knowledge acquisition and formal representation, system construction, validation and testing are typically important. Not only the degree of complexity of these tasks are fairly high but also it is fairly easy to mismanage. Different expert system tools have been developed to improve the situation. More specifically, these tools are used to less the burden on a developer and in a more cost-effective way. Although, there are tools to assist in different phases of such development, we are confined here to the matter of selection of tools during actual system construction process after the selection of a problem domain, and the knowledge acquisition process.

In the next section, we discuss some potential inconveniences in selecting a tool for implementation during an expert system development. In section 6.3, we have analysed the features and capabilities of two potential tools at our disposal : (i) Level5 (object), and (ii) Turbo Prolog. Lastly, our conclusions and discussion are summarised.

6.2. **Potential inconveniences**

This section presents some potential inconveniences faced by an expert system designer and discusses the classification of tools. This section also intends to highlight the inconveniences particularly faced by the author in the selection process in connection with the present work.

6.2.1. **No general purpose tool**

The headache of selecting an appropriate tool would have been relieved if we have been provided a general purpose tool for our use. Unfortunately, it is not possible to construct such a general purpose tool simply because human beings commonly utilise a knowledge based approach rather than a general purpose approach for problem solving [1]. As a result, a range of tools are provided applicable in a wide

[+] This is based on the publication [CSI Communications, vol. 20, no.7, January 1997; 24-32; ibid vol. 20, no. 8, February 1997, 19-22] of the author.

variety of domains. For a successful development of an expert system, it is essential that an appropriate tool is being selected.

6.2.2. **Single or multiple tools**

In a particular situation, a single tool may not be adequate to fulfil the requirements the problem domain lays on an expert system. Due to the evolutionary nature of an expert system development, one may find it worthwhile to use one tool for prototype system and the other for target system. Obviously, this adds one confusion - which type(s) of tools would be suitable for prototype development and which would be suitable for target system development. It is rather difficult to offer sound guidelines here. Two typical approaches may be useful : (i) solving the chicken-egg problem [2], and (ii) use one of the large, hybrid object-oriented toolkits [3] throughout the total developmental phases. In the first approach, it is assumed that few expert systems development efforts are well formulated in advance. The problem domain is complex, domain knowledge and expertise are ambiguous and the nature of the problem is such that complete unfolding is only possible after considerable exploration and experimentation. This is where one has to address the chicken-egg problem. One can choose an appropriate tool only after understanding the total requirements the problem domain lays on an expert system; but this may only be possible after experimentation on building a prototype which requires choosing a tool. This is where two types of tools may be considered useful.

During demonstration / research prototypes, one should use a tool having fast prototyping property e.g. Prolog [4]. The second approach assumes that selection of such appropriate implementation tool should only be done after complete requirement analysis of the problem domain lays on an expert system. In most of the situations this may prevail. Even though the total requirement analysis is not complete, it might be advantageous to use a comprehensive tool assuming that the simpler rule-based production systems do not qualify; model-based reasoning would be more appropriate. Potential difficulties in using multiple tools (discussed below) can be reduced here substantially. The objectives of demonstration / research prototypes can be fulfilled using the simple structures of the tool. More complex structures of the tool can be used for production system. Use of multiple tools may lead to the following problems : (i) Financial investment; may not be practical in many cases, (ii) Man-years investment; before using any such tool, one has to be conversant with the intricacies of the tool. The amount of man-years would be increased simply by a factor of two or three depending on the number of tools used. Moreover, once domain experts and knowledge engineers have been trained on a particular system, it is not simply feasible to be retrained on another potentially suitable system; (iii) Interfacing problem; usually the architectural design and functionality of each tool is different, one has to interface such two or more tools. This would require an extra

degree of expertise involving a fair amount of man-years. The original objectives of using tools e.g. less time, less effort are thereby challenged.

As a worst situation, no such existing higher level tools might be adequate to satisfy the requirements of the problem domain. This is where one has to design his / her own architecture. Here, one has to fix up his / her mind whether the goal of the system development is to develop a system for actual use or to make major advances in the state-of-the-art of expert system technology. As observed by Prerau [5], it is not wise to attempt to achieve both of these goals simultaneously simply because it is really a formidable task. So, a system development for actual use may demand pruning of certain characteristics of the problem domain within the confines of current expert system technology.

6.2.3. **In search of a bird after constructing a cage**

With the advent of increasing number of higher level tools (e.g. shells or toolkits) one can observe an interesting phenomenon [6]. In most cases, peoples are in search of birds after the construction of a cage. That is, the members of a project team are forced to in search of a problem domain which suits the solution techniques of the tool already chosen. This approach, obviously, restricts the natural flow of selection of a problem domain of our society. Therefore, the reverse approach, i.e. selecting an expert system problem domain and then go for selecting an appropriate tool for implementation, is much more natural. Obviously, this may lead to a situation of having no higher level tool suitable for the problem domain.

6.2.4. **Exaggerated claims from vendors / agents**

It is rather very difficult for the end users to distinguish between facts and hyperbole. Vendor literature, demonstrations and reference manuals are subject to exaggerated claims [7]. It may not always be possible to go through the detailed experimental verification of these claims before the actual procurement. Even, sometimes, agents do suppress the potential demerits of the tools for selling his / her product.

6.2.5. **Non-standard terminologies**

Standard terminologies with their standard definitions and actions are really useful for a better comparison of tools and thereby ease the process of selection of such tools. Unfortunately, some tools do not agree in terminologies. For example, in KEE, frames are called units, properties of units are called slots, and properties of slots are called facets. But, in S1, frames are called classes, properties of classes are called attributes, and properties of attributes are called slots. Similarly, the term rule used in ROSIE, ART and RULE MASTER are different in performing actions. So, the

terminologies with non-standard definitions and actions add an extra degree of difficulty, the prospective users face when selecting tools.

6.2.6. **Miscellaneous issues : Price, training and documentation support**

"High price, good quality - more facility" is, however, considered true. But, from functionality and performance point of view, price is not necessarily an indicator of suitability. A tool costing less may be more suitable at per the requirements of the problem domain at hand than a high cost tool. For the ease of use and for the quick exploitation of the potential features of the tool, a comprehensive training should be considered mandatory. The trainer should be an expert preferably a core member of the group responsible for the development of the tool. In different developed countries, the source of most advanced tools, this can easily be observed. Most of the vendors have their online facilities to support the customers; this facility can easily be enjoyed by the customers having advanced communication tools. For us, in India, the situation is not so favourable. Usually, the vendors sell their products through agents; the agents are not well-equipped with the properly trained experts. Even, the agents are, sometimes, reluctant to offer an on-site demonstration. Frequent on-line assistance is again a costly affair; may not be feasible for every customer. The situation may be improved with good documentation including a user's guide, a reference manual and an architecture manual and some demonstration examples with clarification.

6.2.7. **Language, shell or toolkit**

The selection of an appropriate tool should also be dictated by the relative merits and demerits of these three classes of tools. Generally speaking, the shells provide the upper level of a stratum of tools, the lower level is being provided by the languages and the middle level is being provided by the toolkits. The metrics like applicability, abstraction, facilities and costs of hardware, software and training may be considered useful in the comparison process [1] as discussed below.

6.2.7.1. **Applicability**

Languages are applicable quite generally and virtually any type of expert system can be designed. On the other hand, shells are rather specific in this context. The good matching of the requirements the domain lays on an expert system and the facilities the shell offers is the key to success of the development of an expert system. The toolkits should have the generality of the language approach but also contain specific representations and control strategies.

6.2.7.2. **Abstraction**

The level of abstraction is low in the language approach and medium in the shell approach. On the other hand, toolkits provide a rich set of abstraction.

6.2.7.3. **Facilities**

In the toolkit, the facilities are most rich. Shells offer medium facilities. In a language approach, we get limited facilities but, however, any facility missing in a language may be provided by programming.

6.2.7.4. **Costs**

(i) **Hardware**

In the case of languages and shells, the costs of hardwares are generally quite low compared to toolkits. This is simply because toolkits often demand specialised hardwares whereas languages and shells run on PC or on workstations. Although, recent versions of some toolkits run on PC but the memory / backup size is reasonably high.

(ii) **Software**

The costs of languages and shells are more or less same. But, toolkits are normally more costly than other two.

(iii) **Training**

In general, the shells require a fairly less efforts and short time for learning. But, however, languages require a more extensive efforts and training period.

With due consideration of the observations, we summarize that the toolkit approach appears to be superior to other two approaches on the consideration of applicability, abstraction and facilities offered. But, however, it appears to be inferior to language and shell approaches on the consideration of costs of hardware, software and training. Moreover, although, a toolkit offers a good number of facilities, these may not fulfil all the requirements of the problem domain lays on an expert system. The programming facility, if any, of a toolkit is expected to fulfil such requirements. Once again, one has to be a master of a language like LISP or PROLOG which is provided by a toolkit. One may now have an idea on the sales of expert system software tools per year [8] as depicted in fig. 6.1. From fig. 6.2, one can have an idea how expert system applications have developed : (a) on different platforms, and (b) with different softwares.

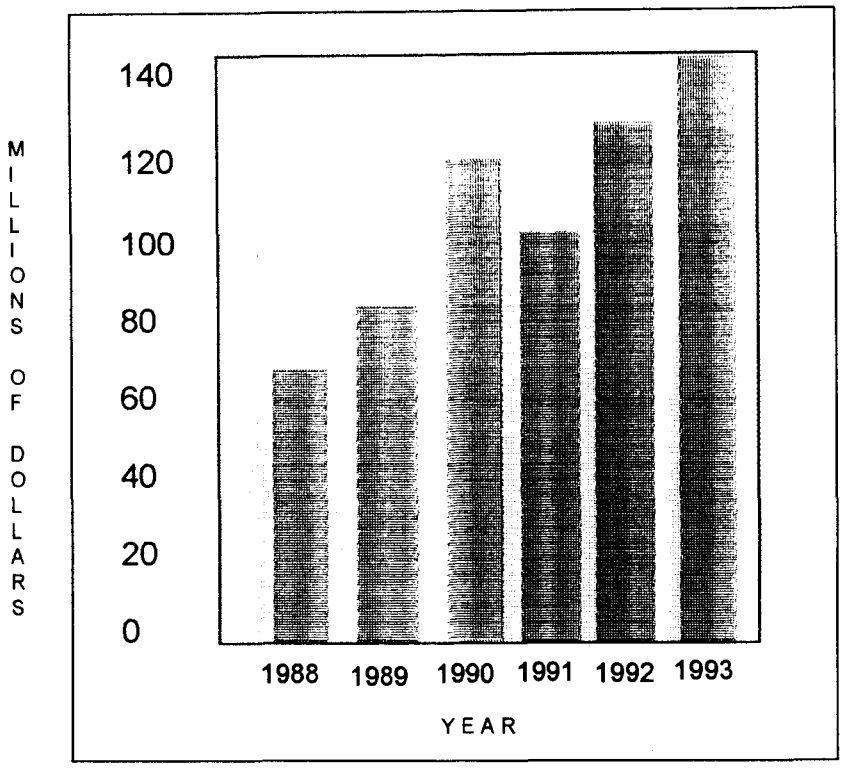

Fig.6.1 Sales of expert system software tools per year

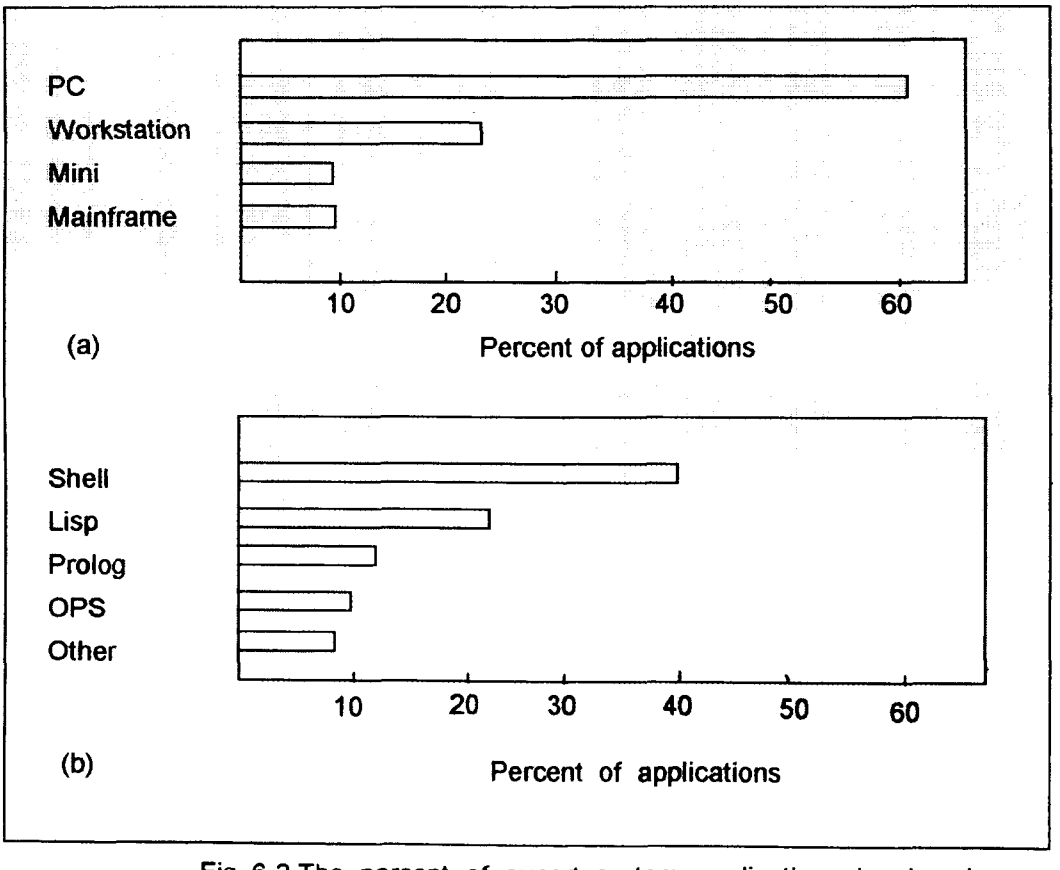

Fig. 6.2 The percent of expert system applications developed
(a) on different platforms; (b) with different softwares

6.2.8. **Left no stone unturned - Is it practically feasible ?**

A decade back literature [9] would be adequate to explain the above head line. Bundy [9] provides a catalog of over 250 software products and AI techniques. Hopefully, this number should be over 450 at the end of 1999. In such a situation, it may be pertinent to ask whether it is practically feasible or not for an end user to turn each and every stone. This should not be an impossible task but this may lead to an unacceptable delay in achieving the ultimate objective of selecting such a tool. Obviously, this demands a fast pruning mechanism.

6.2.9. **Potentially active research field**

It may not be possible to select the best one but a better one due to the evolutionary nature of this potentially active research field. One may find a better tool tomorrow satisfying more need of the problem domain. But, to develop a system for practical use one has to be confined to the present ES-technology where all the requirements may not be satisfied.

6.2.10. **Any unique framework ?**

Because of the so many turbulant features just creating the unstability in the selection process, the tool evaluation and selection can not entirely be mechanical. Here, human expertise and judgement will certainly play a significant role especially for the pruning process. But, obviously, this might lead to different solutions of the same problem which again demands a more formal mechanism. To what extent this formalisation would be achieved ? Answering this question and suggesting a formal method are really formadible tasks. It is, rather little bit easier to suggest a general framework for the problem. Rothenberg [2] suggests a framework with eight fold steps emphasising "matching a tool to its intended use" rather than simply "matching a tool to a problem". This framework might be worthwhile in many cases but it involves a larger number of criteria, may not lead to a manageable situation.

For example, 'contexts' dimension might demand five different tools suitable for five different contexts such as conceptualization, prototyping, development, fielding and operation / maintenance; leading to an unmanageable situation. But, however, pruning and prioritizing should make the situation manageable which again may depend on individual's experience and judgement. This might suggest another more simple framework. However, the matter is yet to be settled. But, it may be of general acceptance that such an evaluation and selection should primarily depend on the requirements the domain lays on an expert system and on the tool capabilities rather than tool features. This demands a thorough analysis of the problem domain, the

problem itself and the anticipated project including, even, the potential users of the proposed system.

6.3. **ES-building tools' capabilities**

It might be more important to focus on the capabilities of a tool, rather than the specific features the tool provides for achieving or supporting those capabilities. Highlighting capabilities means highlighting the functionalities of a tool rather the specific implementation of a functionality. Users are generally interested in different capabilities (and it is also more convenient for less experienced end users) without knowing the technical features supporting those capabilities. Some representative potential capabilities and the corresponding supporting features are identified here as presented in table 6.1 which includes the suggestions of Rothenberg [2].

Table 6.1. Capabilities of tools with supporting features.

Capability	Examples of Supporting Features
Arithmetic processing	Arithmetic operators, extended floating point
Built-in-functions	Mathematical, statistical, string, type conversion
Certainty handling	Certainty factors, fuzzy logic
Concurrency	Distributed processing, parallel processing
Consistency checking	Knowledge base syntax checking
Data type handling	Numeric, string, time, simple, compound
Documenting development	Assumption/rationale history, code / data annotation
Explanation	Execution trace, knowledge base browsing
Inference & control	Iteration, forward / backward chaining, inheritance
Integration	Calling other languages, interprocess calls
Internal access	Tool parameter setting functions, source code
Knowledge acquisition	Rule induction, model building aids
Knowledge-base editing	Structure editors, graphic rule lattice
Life Cycle	Tool support for target system life cycle support
Menus	Goals, Reasoning, tools
Meta-knowledge	Rules controlling inference, self-organizing data
Optimization	Intelligent look-ahead, caching, rule compilation
Presentation (I/O)	Text, graphics, Windows, forms, mouse, keyboard
Representation	Rules, frames, procedures, objects, simulation

Let us now consider two potential tools at our disposal : (i) a large-hybrid-object-oriented toolkit - Level5 Object, and (ii) Turbo-prolog for further analysis in the light of the above discussed tool capabilities. We shall be trying to draw a comparison of these two tools according to their capabilities at per the above identified representative list. This discussion is based on Level5 object (Rel 3.0) for Micro-soft Windows and Turbo-prolog (Rel 1.1).

6.3.1. Level5 Object

6.3.1.1. What is Level5 Object ?

Level5 object is a software development tool kit from Information Builders, USA. It is the application development tool to combine client / server technology, object-oriented programming, graphical user interfaces, and knowledge based systems [10].

Level5 object is a software development tool kit. Even if one has little programming experience, he / she can use it to create complex applications in an easy, consistent, and maintainable fashion.

Level5 object is an environment. It contains all the tools necessary to solve a very wide range of problems. The Level5 object toolbox contains an integrated set of editors that help one rapidly create software solutions. From rapid prototypes to large, mission-critical applications, Level5 object is a proven winner as demanded by its developer.

Level5 object is a development tool and a delivery vehicle. Once an application has been created using the Development System, it can be delivered to end-users with the Run-Only System. The Run-Only System provides a variety of delivery choices emphasizing smaller. faster systems. These systems can be encrypted to provide enhanced security.

Level5 object uses a high-level language called PRL. PRL is designed to be simple to learn and read, and is similar to natural English. Although we rarely see it because of the interactive editors, it is there to provide maximum flexibility and accuracy when we develop an application. All of the elements of the software we wish to create can be expressed in the PRL language. The text of the application can be sent to other hardware platforms and operating systems, compiled, and executed. We can therefore create our software solutions where it is most convenient and inexpensively deliver them to other hardware platforms.

Level5 object is well-connected. It contains built-in access to over 60 local and remote data-bases and servers, access to all the common local database formats and SQL servers, interfaces to external programs, communication paradigms, text files, timers, and custom interfacing options.

Level5 object is an expert system. One can use it to create "smart" systems. It can solve real world problems. Using simple rules that can reason or pattern matching triggers that can react to situations, Level5 object can provide consistent, educated answers to the people who need it.

Level5 object contains the following integrated array of powerful tools :

- True objects providing the efficiency of object-oriented programming.
- Graphical User Interface (GUI) development editors, forms and display builders, and control over all aspects of the user interface.
- Complex logic capabilities, business rules, triggers, agendas, procedural and non-procedural modules.
- Robust and seamless database access, SQL, object-oriented databases, and client / server architectures.
- Complete set of integrated debugging tools, stepping, breakpoints, traces, and reasoning.
- 100% portability to other hardware and operating system platforms.
- Compiled execution for efficient application speed and size.

6.3.1.2. **What kinds of problems are best solved with Level5 object** ?

Level5 object has consistently shown itself to be an effective tool for solving certain classes of application problems. Here is a short list of the kinds of problems one can solve with Level5 object. However, because it is a general purpose tool with a broad range of capabilities, one may not be limited to this list.

Scheduling

These applications solve difficult scheduling problems that exceed the capabilities of conventional off-the-shelf solutions. Level5 object's event-driven architecture allows one to create scheduling applications that adjust "on-the-fly" to the changing events of our day-to-day business activities.

Resource and **Constraint Management**

These applications juggle resources and resource constraints to find a best-fit scenario that can mean the business efficiency difference between one and one's competition. Modern business is moving to "just-in-time" operations in order to operate with the least cost and the highest productivity. Innovative organizations use Level5 object as the critical decision-making link in these resource management applications.

Regulation Compliance and **Conformance**

Worldwide, government organizations are using Level5 object to create "smart" forms systems that enforce consistent compliance with government regulations and ensure higher quality decision making by the work force. By codifying regulations and guidelines into business rules and embedding them into automated business process applications, many organizations have improved the quality and fairness of the services they provide.

Diagnostics

Level5 object has the unique ability to resolve complex diagnostic problems across a broad range of industries. Level5 object has proven itself as the quickest and easiest path to a solution, whether one is developing fault detection systems, real-time process monitoring and control applications, or business evaluation and decision support applications.

Client / Server

Level5 object has built-in interfaces to all the common and important local and remote database servers on mainframes and mid-tier systems. Its robust and flexible support of the user interface environment on the client side, as well as its ease of use and development features, make Level5 object an excellent tool to field client / server applications and cooperative processing.

6.3.1.3. **Capabilities of Level5 object**

Let us now have a look on the capabilities of Level5 object as follows :

Arithmetic Processing

Level5 object provides arithmetic processing to the attributes of numeric type of different objects. The types of operators such as assignment, numeric, relational along with numeric functions and numeric value editor etc. are used for the purpose.

Built-in-functions

It provides on-system functions that operate on numeric, string, time, interval attributes. The typical functions are : Mathematical, Reference, Statistical, String, Time / Interval, Trigonometric. Value type conversion.

Certainty handling

Level5 provides a way to build reasoning with partial or uncertain information into a knowledge base. When confidence prompting is 'on', the user will assign a confidence value. In response to customer requests, confidence in Level5 object can be expressed as a numeric value (-2..100). -2 indicates 'unknown' and -1 indicates 'undetermined'. Confidence values can also be assigned from within the rules of a knowledge base, which allows developers to quantify the degree to which they are confident in the accuracy of a conclusion or the degree of accuracy required to make a conclusion. This version supports product space confidence. With this technique, confidence is calculated by multiplying the confidence value of the antecedent and the value of the conclusion. Confidence management strategies other than product space can be developed with the help of confidence function.

Consistency checking

Level5 object closely monitors all respects of knowledge base, it maintains complete referential integrity across rule, file and object management. Level5 goes beyond monitoring to prevent any attempt by a developer to change or accidentally delete classes, data, attributes, rules, instances or demons referenced elsewhere in a knowledge base.

Data type handling

It describes the type of information represented by an attribute of a class using attribute type. The attribute types are : colour, compound, instance reference, interval, multicompound, numeric, picture, rectangle, simple, string and time. In addition, the attribute types may be of single value or an array of different sizes.

Documenting development

In Level5 object, the process of documentation can be achieved using the display editor. Named rules and objects facilitate the documentation process in the system.

Explanation

The Level5 report system is a comprehensive explanation facility that provides complete access to the inferential reasoning process while running a knowledge base. In the case of the human mind, how a decision is reached is usually as important as the determination itself. Similarly, Level5 maintains full audit trails of how it arrives at its conclusions. Analytical reasoning facilities, such as session monitor, historical traces, single-stepping and breakpoint, enables developers and when necessary, end

users to view all the current states of the inference engine; examine and change the state of any fact in a knowledge base; review the answers provided to Level5 quaries; and follow the line of reasoning being pursued. By activating a debug window while running an application, developers can observe and trace the reasoning process. In single-step mode, Level5 pauses after each event the inference engine processes, allowing the developer to view the action before resuming the session.

Inference and Control

Level5 can process information in a variety of ways : back chaining, parallel inference processing, dynamic agendas, rules, blackboard techniques and object-oriented programming. This versatility gives developers ultimate flexibility over the way in which data is presented and processed. It uses rules, demons, methods of WHEN NEEDED and WHEN CHANGED styles. These methods may contain some repetitive structures like IF....THEN.....ELSE; WHILE; DO... UNTIL and FOR.... TO. In addition, LOOP-statement is also available but not applicable to two those methods of WHEN NEEDED and WHEN CHANGED styles.

The class property INHERITS allows one to transfer the structure and behavior of a parent class to a child class. Consequently, a child class receives all of the capabilities of its parent classes and uses not only the attributes of the parent class, but also its methods, rules and demons. Therefore, one can reuse an application's code down the class hierarchy, increasing productivity and accuracy.

In Level5 object, a class can inherit from more than one parent. This feature, called 'multiple inheritance' allows one to create hierarchical chains of inheritance.

Integration

Direct database access enables Level5 to read and write to file types directly from within the knowledge base. Level5 object supports dBASE III and III PLUS, Lotus 1-2-3 WKS and WK1, SQL and ASCII file formats. Using object-oriented database management techniques, developers build and maintain powerful applications that are able to access large, heterogeneous data structures. By committing much of the underlying complexity of data access to system classes, developers can limit end-user access to only pertinent data and make the process of sourcing attribute values from databases virtually transparent. In addition, its DDE system class provides direct program-to-program communication capabilities. In this release Level5 object functions only as a client.

Internal access

Level5 object provides different commands and facets for different parameter values at the begining of a consultant session or when running an application e.g. RESIZE, INIT, REINIT etc. It also provides the source code of a knowledge base in a language what is known as Production Rule Language (PRL).

Knowledge acquisition

Level5 object does not provide any facility for automatic knowledge acquisition for a system development.

Knowledge-base editing

Level5 object is equipped with five editors such as Objects editor, Database editor, Display editor, Methods / Rules / Demons editor and Windows editor. With these editors one can edit different parts of the total knowledge base.

Life Cycle

It is in the nature of systems that share a common life cycle pattern. After a system has been in operation for a number of years, it gradually decays and becomes less and less affective because of the changing environment to which it has to adapt. For the time being it may be possible to overcome problems by amendments and minor modifications to the system but eventually it will be necessary to acknowledge the need for fundamental changes which demands a new system.

In Level5 object the required minor modifications in knowledge base in terms of objects, attributes, rules, methods, demons etc are possible. It is also possible to delete any object or add new objects as when required. The system automatically takes care of the validity of any deletion or modification in the knowledge base. For a new system development the created objects can be used. This expedites the new system development.

Menus

The menu-driven facilities of Level5 object makes it handy to the system developer as well as to the end users.

Meta-knowledge

Different facets of Level5 object tell the inference engine how to process an attribute. Facets like BREAKPOINT, EXHAUSTIVE, REINIT etc. as well as Rules / Methods / Demons are used to control inference in an application.

Optimization

Separate compilation of source knowledge base is not required in Level5 object. The object code is automatically created after addition of source text. It supports compiled execution for efficient application speed and size. One can expedite processing using the SMARTDrive concept. SMARTDrive is a disk-caching program provided with Windows. In addition, writing the software in C increases performance speed.

Presentation (I / O)

Considering I / O presentation, Level5 object supports text, graphics, windows, forms, mouse and keyboard. Where mouse is not available, it runs fully using a keyboard.

Representation

Level5 object is a fully object-oriented system. The objects are managed using objects editor. It also supports rules (single or group), methods (WHEN NEEDED and WHEN CHANGED) and demons.

6.3.2. **Turbo Prolog**

Prolog is short for programming in logic. The language was originally developed in 1972 by Alain Colmerauer and P. Roussel at the university of Marseilles in France for writing natural language translation systems, and it quickly became one of the preferred languages for other artificial inteliigence applications. It is a computer language that was created especially for answering questions about a knowledge base that consists of rules and facts. Prolog has built-in 'backward chaining' and also utilizes another technique known as backtracking. Backward chaining is a technique in which a conclusion or consequence is assumed to be true and then knowledge base of rules and facts is examined to see if it supports the assumption. If the assumption turns out 'not' to be correct, backtracking is said to get rid of the original assumption and replace it with a new one.

Like procedural languages, prolog does not use algorithms. Algorithms are objective procedures that when followed, guarantee a solution. Object-Oriented languages, such

as prolog [11], use heuristics. Heuristics are rules of thumb that are useful in reaching a goal. They do not guarantee a solution and are useful when no algorithms exists. Prolog uses only data about objects and their relationships. Prolog also emphasizes symbolic processing.

There is no standard for the prolog language but the closest thing to an accepted standard is the prolog described in Clocksin and Mellish's book which is generally known as C and M prolog [12]. Most commercial prologs have evolved as super sets of this core language. Turbo prolog is both a superset and a subset of C and M prolog, yet it does not support many features that are supported in the core language. In addition, turbo prolog supports some features (such as input and output) by structures more similar to those of the C language than those of C and M prolog. Programs written in C and M prolog will not run under Turbo prolog without modification. Similarly, turbo prolog programs must be extensively modified to run under C and M prolog.

6.3.2.1. **Features and Capabilities**

Let us now have a look on the features and capabilities of Turbo Prolog language.

Knowledge in turbo prolog can be expressed in terms of either facts or rules. The basic units for building facts or rules are predicates, i.e., expressions that say simple things about the individuals in our universe. For instance, the piece of information 'the left speaker of the stereo system is not emitting sound (is dead)' is represented in turbo prolog as

IS (left_speaker, dead).

The factual expression in prolog is called a clause.

In the above example, left_speaker and dead are objects. An object is the name of an element of an entity in the real world. Turbo prolog permits the user to use six different object types viz, char, integer, real, string, symbol and file. The word IS is the relation in the example. A relation is a name that defines the way in which a collection of objects (or objects and variables referring to objects) belong together.

The entire expression before the period is called a predicate. A predicate is a function with a value of true or false. Predicates express a property or a relationship. The word before the parentheses is the name of the relation. The elements within the parenthses are the arguments of the predicate, which may be objects or variables.

Here are a few examples of predicates :

```
Car(Maruti)
Country(India)
M_tone(Patient)
```

- **Rules**

Rules permit prolog to infer new facts from existing facts. A rule is an expression that indicates that the truth of a particular fact depends upon one or more other facts. Let us consider the following example.

```
IF  the Muscle tone of limbs is flaccid AND
Heart-rate is low AND
Respiratory effort is slow irregular AND
Reflex stimulation is grimace AND
Colour is periphery blue but body pink
THEN the patient probably has to be undergone with type II resuscitation.
```

The rule could be expressed as the following turbo-prolog clause :

```
rescus(Patient, type II) if
        m_tone(Patient, flaccid) and
        h_rate(Patient, low) and
        res_eff(Patient, slow_irregular) and
        ref_sti(Patient, grimace) and
        colour(Patient, p_blue_body_pink).
```

The conditions upon which conclusion depends are stated next, each connected by the word and. The conclusion can be viewed as a prolog goal. The goal is true if all the conditions specified for the goal are true.

Every rule has a conclusion (or head) and an antecedent (or body). The antecedent consists of one or more premises. The premises in the antecedent form a conjunction of goals that must be satisfied for the conclusion to be true. If all the premises are true, the conclusion is true; if any premise fails, the conclusion fails.

Since rules are such an important part of any prolog program, a shorthand has been developed for expressing rules. In this abbreviated form the previous rule becomes

```
rescus(Patient, type II) :-
        m_tone(Patient, flaccid),
        h_rate(Patient, low),
        res_eff(Patient, slow_irregular),
        ref_sti(Patient, grimace),
        colour(Patient, p_blue_body_pink).
```

The :- operator is called a break. A comma expresses **and** relationship and semi-colon expresses an **or** relationship. The process of using rules and facts to solve a problem is called formal reasoning.

- **The Turbo-Prolog Program**

Turbo prolog program consists of two or more sections. The main body of the program, the clauses section, contains the clauses and consists of facts and rules. The relations used in the clauses section are defined in the predicates section. Each relation in each clause must have a corresponding predicate definition in the predicates section. The only exceptions are built-in predicates that are an integral part of turbo prolog.

The domains section is also as part of most turbo prolog programs. It depends on the type of each object.

The goal section of the program defines the internal goal. If turbo prolog can match the goal with a fact or rule in the data base, it succeeds otherwise it fails. Variables are used in a turbo prolog clause or goal to specify an unknown quantity. A variable name must begin with a capital letter and may be from 1 to 250 characters long. Except for the first character in the name, one may use upper case or lower case letters, digits or the underline character. If a variable has a value at a particular time in a program, it is said to be bound. If a variable does not have a value at a particular time, it is said to be free. For a goal to succeed, all variables in the goal must become bound.

The turbo prolog program can consist of upto eight sections. Most programs only use a few of these sections, but the complete list gives the extensive flexibility : global predicates for modular programming, dynamic databases, internal goals and compiling directives.

Although not all sections will be used in all programs, those that are used must be in the following order for the compile operation.

1. Compiler directives
2. Domains
3. Global domains
4. Data base
5. Predicates
6. Global predicates
7. Goal
8. Clauses.

- **Unification**

The process by which prolog tries to match a term against the facts or the heads of the other rules in an effort to prove a goal is called unification. Unification is a pattern matching process.

Predicates unify with each other if

1. They have the same relation name.
2. They have the same number of arguments.
3. All argument pairs unify with each other.

- **Prolog execution rules**

The rules for prolog execution are given below :

1) Prolog executes using a matching process.

2) When the original goal is specified, prolog tries to find a fact or the head (conclusion) of a rule that matches the goal.

3) If a fact is found, the goal succeeds immediately.

4) If a rule is found, prolog then tries to prove the head of the rule by using the antecedent (the body of premises) as a new compound goal and proving each premise of the antecedent. If any premise of the antecedent fails, prolog backtracks and tries to solve the preceding premises with other bindings. If prolog is not successful, it tries to find another fact or rule that matches the original goal. If all premises succeed, the original goal succeeds.

5) Turbo prolog continues to execute until all possible solutions for the goal are tested.

There are no global variables; that is, variables that maintain a value throughout the entire programs execution. All variables are local to the clause of which they are a part. Even if the same variable name is used in another clause, it is not the same variable. If a particular clause fails, prolog backtracks and tries to solve the same goal another way using clauses.

- **Built-in Predicates**

Turbo-prolog offers dozens of built-in predicates to support not only input and output operations, but graphics, file operations, string handling and type conversion. These

predicates can be used in rules or facts, just like any other predicate. They do not need to be defined in the predicate section, as they are integral to turbo prolog.

The turbo prolog predicates are classified into eight groups :

1) Reading predicates :
   For reading data from the keyboard or file to a variable.

2) writing predicates :
   For writing data to the screen, printer or a file.

3) Control predicates :
   For controlling program execution, forcing or preventing backtracking.

4) File system predicates :
   For managing disk files from a turbo-prolog program.

5) Screen-handling predicates :
   For graphic control of the display and sound control.

6) String-handling predicates :
   For various operations on string data.

7) Type-conversion predicates :
   For converting data types from one form to another.

8) System-level predicates :
   For access to MS-DOS functions from within a turbo prolog program.

A few examples of the built-in predicates are given below :

- **The read predicate**

Turbo-prolog provides several built-in input predicates. One of these is a read predicate. There are four types of read predicate : readin, readchar, readint and readreal.

The readln predicate permits a user to read any string or symbol into a variable. Let us consider the following example :

```
Symptom(Child, fever) :-
        write ("Did the ", Child, "have vomiting last night (yes / no)?"),
        readln (Reply),
        Reply = "yes".
```

When this is invoked, it displays the question and then pauses for an answer. The rule will succeed if the user enters yes and fail if the user enters anything else. In the same way, **readchar** predicate permits to read any character, **readint** to read any integer and **readreal** to read any real value into a variable.

- **The write predicate**

The write predicate displays the value of the variables or objects. The variable must be bound before the write predicate is invoked. For example, The clause

```
test :-
        write ("Sorry, I don't find "),
        write ("no more new findings").
```

displays

```
Sorry, I don't find no more new findings.
```

The output may be written in two lines using the control predicate nl. The predicate nl indicates a new line. Any output to the display can be directed to a printer or a file. To redirect output, the writedevice built-in predicate is used.

The following code directs the output of the write predicate to the printer and then redirects further output back to the screen.

```
        writedevice (printer),
        write ("This will print on the printer"),
        writedevice (screen).
```

Likewise Turbo-prolog provides the readdevice predicate for redirecting input.

- **The fail predicate**

In prolog, forcing a rule to fail under certain conditions is a type of control and is essential to good programming. Failure can be forced in any rule by using the built-in fail predicate. The fail forces backtracking in an attempt to unify with another clause. Whenever this predicate is invoked, the goal being proved immediately fails, and backtracking is initiated. The predicate has no arguments, so failing at the fail predicate is not dependent on variable binding; the predicate always fails.

Example :

```
go:-
        test,
        write ("you will never get here").
test :-
        fail.
```

If the goal is specified as go, prolog will unify with the head of the first rule and then try to prove it premises. The test premise will unify with the head of the second rule, whose premise is fail, and the goal will fail. Prolog will backtrack to the first rule and the go goal will fail again. The write predicate will never be executed. The same program may be tried again, reversing the two premises and changing the test string slightly :

```
go :-
        write ("you will get here"),
        test.
test :-
        fail.
```

Again, the goal will fail, but the text string will be displayed this time.

- **The not predicate**

If we want to express explicitly in the database that a particular fact is not true, the built-in not predicate has to be used. The not predicate cannot be used to express a fact or appear in the head of a rule. It can only be used in a premise, as in

```
replace (left_speaker) :-
not (is (left_speaker, functional)).
```

In this case, if

```
is (left_speaker, functional).
```

is in the data base, the rule will fail.

- **The cut predicate**

The cut is one of the most important, and also one of the most complex, features of prolog. The primary purpose of the cut is to prevent or block backtracking based on a specified condition. The cut predicate is specified as an exclamation point (!). It has no arguments. The cut predicate always succeeds, and once it succeeds, it acts as a fence, effectively blocking any backtracking beyond the cut. If any premise beyond the cut fails, prolog can only backtrack as far as the cut to try another path. If the rule itself fails and the cut is the last premise, no other rules with the same head can be tried. Prolog must accept failure or success of the predicate based on that particular clause.

There are two types of cuts. In Turbo prolog, these are called the **green** and the **red** cuts. The green type of cut is used to force binding to be retained, once the right clause is reached. Green cuts are used to express determinism. A program is nondeterministic if it is capable of generating multiple solutions on backtracking. The red type of cut is used to omit explicit conditions.

The use of any type of cut in a prolog program is controversial. It implies a type of procedural control, which is in sharp contrast to the declarative style of prolog programming. If used with caution, however, cuts improve the clarity and efficiency of most programs. Of the two types of cut, the green cut is the more acceptable type. One can often use the not predicate instead of the red cut.

Example :

Let us assume a rule of the form :

```
go :-
        premise1,
        premise2, ! ,
        premise3,
        premise4.
```

Prolog will try to prove the go by backtracking between premise1 and premise2 as necessary until both are true. Once this occurs, the cut is reached. The cut predicate always evaluates as true, so testing begins on premise3. Once the cut is crossed, prolog cannot backtrack across the cut. If premise3 fails, the rule fails. If premise4 fails, prolog backtracks to premise3 to try another path for this premise, but prolog goes no further. If either premise3 or premise4 does not evaluate as true, the rule fails without going back to premises before the cut and attempting to prove them in a different way, for example, with different bindings. All variables in premise1 and premise2 are bound when the cut is crossed, and prolog is committed to all choices before the cut.

- **The makewindow predicate**

A window may be created at any time in a program by using the makewindow predicates. The general form of the makewindow predicate is

makewindow (WindowNo, ScrAttr, FrameAttr, Header, Row, Col, Height, Width)

All except the header arguments are integers. The header is a string or a symbol. The arguments are explained as follows :

WindowNo	Assigns a number to the window.
ScrAttr	Defines the video attributes of the window display.
FrameAttr	Defines the video attributes of the border.
Header	Defines a header for the window.
Row, Col	Defines the starting point (upper left corner) for the window.
Height, Width	Defines the size of the window.

Example :

```
makewindow (1, 31, 31 "A FUZZY KNOWLEDGE BASED NEONATAL RESUSCITATION
            MANAGEMENT SYSTEM ", 0, 0, 25, 80)
```

Several windows can be created with separate number. The shift between windows can be performed using shiftwindow predicate :

```
shiftwindow (WindowNo)
```

When a window is shifted, the window will be clear, with the cursor at the upper left. The cursor (Row, Col) predicate positions the cursor at (Row, Col) in the currently active window.

- **Prolog databases**

A prolog program is a collection of facts and rules about a particular knowledge domain. The program really is a database and prolog is very powerful query language for this database, permitting one to select facts from the database through unification. The program, however, is a static database; that is, the database does not change over time. To solve the medical diagnostic problem, prolog permits us to add a dynamic database to the program. Built-in predicates permit to add facts to or remove facts from this dynamic database during program execution. To store information in a dynamic database, one or more database predicates have to be created. Facts can then be stored in these predicates during program execution using the built-in **asserta** or **assertz** predicates. To creat database predicates, a database section must to be added to the program. The database predicates are defined in this section. The section must follow the domains section and precede the predicates section.

Example :

```
Domains
        disease, symptom = symbol
        query  = string
        reply  = char
database
        xpositive (symptom)
        xnegative (symptom)
predicates
        hypothesis (disease)
        symptom (symptom)
        :
        :
```

Here, two database predicates are used in the database section : xpositive and xnegative. The database predicates are not defined in the predicates section. With

these database predicates one can store the facts, learned from the questions, in the databae (facts proven true in xpositive and facts proven false in xnegative) and query the database before asking the same question again.

- **The asserta and assertz predicates**

The asserta (fact) predicate stores a fact at the begining of the database. The assertz (fact) predicate stores a fact at the end of the database. Which of these will be used in the program depends upon where one wishes to put the fact in the database.

- **The retract predicate**

Once a fact is stored in the database, it can be removed only using the built-in retract (fact) predicate.

Example :

retract (xpositive (fever))

removes the fact fever from the database xpositive (fact). To save the current dynamic database, the built-in predicate save (filename) predicate may be used.

- **File opening predicates**

There are four file-opening predicates in turbo-prolog. These are :

1.	Openread (SymbolicFilename, Filename)	Opens file for reading. If the file is not there, the predicate fails.
2.	Openwrite (SymbolicFilename, Filename)	Opens the file only for writing. Any previous file with the same name is deleted.
3.	Openappend (SymbolicFilename, Filename)	Appends any new output to the end of file-name. If filename does not exist, the program terminates execution.
4.	Openmodify (SymbolicFilename, Filename)	Opens file for writing and reading using random access.

All the predicates discussed here have the same number and type of arguments. Any of them can be used to open a file. The first argument is the symbolic file name that will be used for the file in the program. The second argument, filename, is the name of the file on the disk. A file may be accessed randomly using the filepos built-in predicate. The general form of the filepos predicate is

filepos(SymbolicFilename, Position, Mode)

where Mode determines how position is measured as follows :

0 = from begining of the file
1 = from current position
2 = from end of file.

The end of the file during a read operation may be checked using

eof (SymbolicFilename).

The test will be succeed if current position is at the end of the file; otherwise, it will fail. To prevent loss of data, closefile predicate is used to close a file after using it. The general form of the closefile predicate is closefile (SymbolicFilename).

6.3.3. **Requirements vs. Capabilities**

Knowledge bases can be exported to a text file using PRL syntax in Level5 object. PRL (Production Rule Language) is Level5 object's application development language. One can see the underlying PRL structure of his / her application when one exports it to a text file. One can edit this file and also send it to Level5 object running on other hardware platforms and operating systems. This process lets one create applications where it is most convenient and deliver them to the platforms one wants. But, however, when export of an application to be transported to character-based platforms, such as VAX / VMS or MVS, some elements will import, but will have no effect when the application is run. For example, pictureboxes do not appear in character-based displays. So, with the PRL structure one can achieve the portability.

From the PRL source text, one can have the facility of quick and easy modification of existing knowledge base. So, the proper patching work is not difficult here. With the different editors one can easily add or delete an object, its attributes, methods and the consistency checking is monitored by the Level5 object itself.

Probably, one of the limitations of current tools is in their handling of inexactness of information. Level5 object manages only one form of inexactness i.e. uncertainty, in the form of certainty factors(-2..100). MYCIN style of approach has been used here. However, it is not capable of handling other forms of inexactness as identified for our problem domain e.g. fuzzy information, simultaneous occurence of uncertainty and fuzziness, and uncertain-fuzzy. But, however, implementing NMR may be more or less easy, although, it is very difficult to pinpoint the objects / attributes / rules / methods / demons affecting the absurd conclusion. But, once, they are identified it is easy to upgrade the information in knowledge base using different convenient editors.

Level5 object provides a complete access to the inferential reasoning process while running a knowledge base. Its analytical reasoning facilities, such as season monitor, historical traces, single-stepping and breakpoint enable developers (and when necessary, end users) to view all the current status of the inference engine; examine and change the state of any facts in a knowledge base; review the answers provided to Level5 queries; and follow the line of reasoning being pursued. By activating a debug window while running an application, developers can observe and trace the reasoning process. In single-step mode, Level5 pauses after each event the inference engine processes, allowing the developer to view the action before resuming the session. So, a comprehensive explanation facility is being provided by Level5.

Although Level5 objects manages objects and attributes from editor's panel which essentially freeze the knowledge base before running an application, but, however, it can manage instances of an object dynamically i.e. during running an application using MAKE and FORGET commands. The system can learn the situations of using MAKE and FORGET for the defined rules / methods / demons. The repetitive questionnaire of same kind during interrogation with a child and/or with parents / guardians can be avoided by suitably navigating the question-answer sequence. One can save recommendations of a typical session using an external database (typically dBASE III +). The required structuredness and modularity are being assured by the object-oriented design strategy of Level5 object.

Level5 supports the development of large applications through the use of knowledge-based subroutines that allow knowledge to be grouped modularly or chained to knowledge bases. Besides being easier to maintain, subroutines enable a host knowledge base to resume processing where it left off. Developers can navigate between modules and even redefine views of the knowledge domains.

Now considering Turbo prolog, we found the following attractive features :

- In-built inference mechanism which expedites initial prototype development of the system,
- Separation between knowledge base and inference mechanism,
- Static and dynamic knowledge bases,
- Declarative as well as some procedural knowledge can be intermixed,
- Supports modular and structured programming to achieve good modifiability,
- Table look-up scheme for fuzzification and the defuzzification processes can be easily implemented using TURBO-PROLOG's list-structure,
- Backward chaining favours the medical diagnosis problems, and
- Object-Oriented language[11].

6.4. **Conclusions** and **Discussion**

In this chapter we have tried to explain some of the potential inconveniences faced by an expert system designer in selecting an appropriate expert system implementation tool. In connection with the present work, the author has expressed his observations on two tools :

(i) Level5 Object, and
(ii) Turbo Prolog.

An attempt has been made to judge the features and capabilities as discussed in this chapter. Obviously, Level5 object has certain good features and capabilities to act as an implementation tool. But, however, it is fair to state that it has rudimentary capability of handling inexactness. In this respect, Turbo prolog may be considered suitable since it is a language where one can develop his / her own architecture.

References

1. J. L. Alty. Expert system building tools. In Topics in expert system design. G. Guida and C. Tasso. (ed.). Elsevier Science Publishers B. V.; (North Holland), 1989, 181-204.

2. J. Rothenberg. Expert system tool evaluation. In Topics in expert system design. G. Guida and C. Tasso. (ed.). Elsevier Science Publishers B. V.; (North-Holland), 1989, 205-229.

3. P. Harmon (ed). The cost effectiveness of the major U.S. Expert Systems Building Tools. Expert Systems Strategies; vol.3, no.11, 1987, 14-16.

4. I. Bratko. Fast prototyping of expert systems using Prolog. In Topics in expert System Design. G. Guide and C. Tasso. (ed.). Elsevier Science Publishers B. V.; (North Holland), 1989, 69-86.

5. D. Prerau. Selection of an appropriate domain for an Expert System. The AI Magazine; Summer 1985, 26-30.

6. A. B. Baskin and R. S. Michalski. An integrated approach to the construction of knowledge-based systems : Experience with advise and related programs. In Topics in expert system design. G. Guida and C. Tasso. (ed.). Elsevier Science Publishers B. V.; (North-Holland), 1989, 111-143.

7. W. Mettrey. A comparative evaluation of expert system tools. IEEE Computer; February, 1991, 19-31.

8. J. Durkin. Expert Systems : A view of the field. IEEE Expert; April 1996, 56-63.

9. A. Bundy (ed). Catalogue of artificial intelligence tools. Second Revised ed., Springer-Verlag; N. Y.,1986.

10. LEVEL5 OBJECT for Microsoft Windows Getting Started Guide (Release 3.0). Information Builders, Inc.; USA, 1993.

11. R. S. Pressman. Software Engineering : A practitioner's approach. 3rd. ed., Mc-Graw Hill Inc.; 1992, 531-532.

12. W. F. Clocksin and C. S. Mellish. Programming in Prolog. Springer Verlag; Berlin, 1981.