3 Expert system fundamentals[1]

3.1 Introduction

Expert systems are knowledge-based computer programs instructed to function like a human expert does in solving a particular problem or in giving advice. This does not mean, however, that they have brain functions at their disposal which are similar to that of humans. It is yet far from understood how our brain functions and expert systems are only one way by which it is tried to simulate human performance. The more formal definition that will therefore be followed, is given by Hayes-Roth *et al*. (1983), which says that an expert system is a program with a wide base of represented knowledge in a restricted domain, that uses inferential reasoning and, when necessary, user dialogue to perform tasks which a human expert could do.

The development of the expert system as we know it today, started in the late sixties. One of the first systems was developed at Stanford University, called DENDRAL (Feigenbaum *et al*. 1971). It could be used to identify the chemical structure of unknown substances. A few years later, another famous system (MYCIN) was built at Stanford (Shortliffe 1976). MYCIN was used for diagnosing patients suffering from a bacterial infection. Although the first systems were mainly built for the purpose of research, it was soon recognized that they could be of practical use as well. Especially companies and industries active on fields on which expertise is hard to get and thus expensive, considered them beneficial. Over the past two decades, various applications have shown the benefit of the expert system approach and nowadays they are successfully employed in all kinds of fields (*cf* Feigenbaum & McCorduck 1983; Jackson 1986; Vadera 1989).

The most interesting aspect of expert systems is that they offer a possibility to capture and organize human expertise and experience into a form that enables other people to employ it. This is not only interesting for laymen, but also for the expert who offers his or her knowledge. Most experts spend a large percentage of their time on problems they consider simple and, therefore, less interesting. For them, solving such problems is a routine. If an expert system could take over (part of) this routine, the expert gets the opportunity to concentrate on difficult and more interesting problems and to engage in new challenges that can expand his knowledge.

In this chapter, the fundamentals of expert systems will be described. It starts with their architecture, *i.e.* the components they consist of and their specific task (paragraph 3.2). Subsequently it will be discussed how knowledge and reasoning processes can be represented by means of expert systems (paragraph 3.3), of what elements the development process of an expert system application consists, what difficulties can be expected during a development trajectory (3.4), and what tools are available to build an application (paragraph 3.5).

3.2 Architecture

Expert system applications differ from other computer programs in their tasks and architecture. Applications which are built according to traditional programming methods consist of explicit and task-specific algorithms: they perform a task on the basis of a set of actions which are processed in a predefined order. Consequently, traditional programming methods can only be used for tasks that have an algorithmic nature. The expert system approach, on the other hand, has been designed to handle tasks which cannot be solved by straightforward and predefined procedures but by *heuristic methods* only. Heuristic methods are based on the concept of *trial-and-error*. They do not use formal problem solving procedures, but they simply test approaches of which it is uncertain whether they will lead to a solution. Expert systems employ actions that can be executed independently of each other and in a variable order. The application chooses the appropriate activities on the basis of the information that it receives from the external world. Consequently, the course of the program is automatically accommodated to the situations it is confronted with. This implies that an expert system application is more flexible than one which is built according to traditional programming methods. In figure 2 some of the tasks are shown for which this approach is known to be useful.

One of the main advantages of flexibility is speed. This can best be illustrated by the following, slightly overdrawn, example. Imagine a program that is employed to diagnose diseases. If this is a traditionally written program, it consists of a huge list of possible diseases which will be evaluated in a given order on the basis of the symptoms that are provided by the user. This evaluation may be quickened by using

Interpretation of data or a situation

Prediction of consequences of a given situation

Diagnosing diseases or malfunctions from symptoms

Planning of actions given certain circumstances

Monitoring behavior of vulnerable systems or instruments

Instruction of students

Classification of objects

Fig. 2. Examples of tasks suitable for an expert system approach (After Hayes-Roth 1983).

indexes, but it has to check all main groups of diseases in a given order until one will be encountered that fully fits the described symptoms. If the worst comes to the worst this may imply that a person suffering from a critical cardiac-arrest may first be questioned about all kinds of irrelevant infirmities before the system eventually concludes that the patient has died of an inadequately treated heart attack.
If this program would be an expert system, it may also consist of an enormous amount of knowledge which covers all possible diseases, but it is able to search more directly for the most appropriate one. Expert systems anticipate on the information that is received from the external world, *i.e.* the description of the patient's symptoms and his medical history, by eliminating possibilities and by avoiding irrelevant questions.

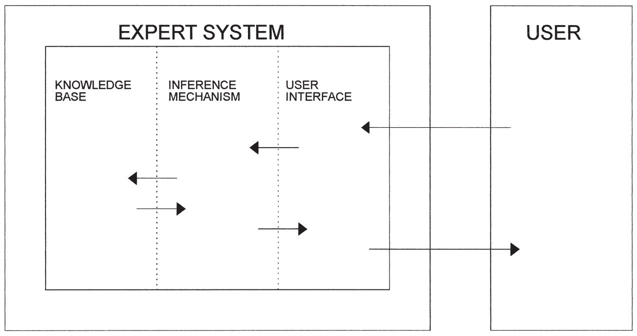

Fig. 3. Expert system architecture. The arrows indicate the system's internal and external lines of communication.

Expert systems owe their flexibility to their architecture: they are composed of three elements that operate independently of each other. These elements are a *knowledge base*, an *inference mechanism*, and a *user interface* (fig. 3). The first element comprises the knowledge that an application requires. In a way, it can be compared with a database, because they are both storage facilities. The main difference, however, is that a knowledge base contains knowledge instead of raw data or information. Within the context of artificial intelligence research there has been much discussion on what 'knowledge' exactly is and it appears that it can have several forms, like defaults, facts, rules of thumb, strategies etc. In broad outline, knowledge can be separated into a static (or descriptive) and a dynamic (or procedural) part, representing respectively the facts and the conclusions that can be drawn from them. In the context of this study, knowledge is defined as facts and the relations between these facts.
The second element of an expert system is an inference mechanism. Whereas the knowledge base consists of domain dependent facts and relations, the inference mechanism consists of domain independent procedures. It can be seen as the central nervous system: it controls the reasoning process, *i.e.* the problem solving strategy. It selects the knowledge that is needed to solve the problem or to carry out the task. In other words, the inference mechanism makes sure that the appropriate knowledge is applied at the appropriate moment. An expert system employs its knowledge either to interpret new information or to collect information that may answer a question. They are *data oriented* or *goal oriented*, respectively. Both approaches use a specialized reasoning strategy. A data-oriented system has no predefined goal: it reacts to information that the system receives from the external world.[2] The system will try to interpret this information by consulting its knowledge base for conclusions that can be drawn from it. This is called *forward reasoning*. A goal-oriented system does the opposite, it 'reasons' in a *backward direction* in order to confirm a predefined goal. It will try to retrieve information from the external world that is required to confirm that goal. This can be done by questioning the user or by consulting an external data source such as a database. Since data-oriented systems can be used to interpret data or to react to (changes of) incoming information, they are most suitable for applications with analytical and educational purposes, especially for those that require an immediate reaction of a 'master'. A goal-oriented system can best be applied to situations in which a user either wants to have a hypothesis validated. In paragraph 3.3 will be further elaborated on reasoning strategies.
The third component of an expert system is a user interface. It handles the communication between an application and its users. Any application needs communication with the outside world in order to gather information that can help to solve the problem — or to perform a task — and to return its conclusions. Since the quality of the information is of decisive influence on the adequacy of the reasoning process and thus for the conclusions that the application can draw, it is very important that the dialogue between the system and its

user does not cause misunderstandings (see also chapters 5 and 7). Therefore, the interface must be adapted to the level of the user and provided with explanatory facilities. The same counts for the transmission of the system's conclusions. A system can only present its suggestions and advices to its user through the user interface. In order to convince the person on the other side of the screen or to enable him or her to make the right decisions, the application must offer clear messages and additional information on how it reached its conclusion.

Irrespective of the fact whether an application is data oriented or goal oriented, the dialogue between system and user can be user initiated, computer initiated or a mixture of both. The first form is often used by systems designed to support users with a high level of experience on the domain. These users only ask the system for advice in case of difficult problems and they determine the system's role. A computer-initiated dialogue is characteristic of systems designed to give direction to users without any domain experience. Depending on the degree of experience of the users, the dialogue is sometimes also alternating initiated by the user and the computer.

It is this special architecture of expert systems that realizes the required flexibility. Due to the fact that the knowledge base and the reasoning mechanism are independent elements, the latter can consult the knowledge base whenever it is required and it can select only those facts and relations that are relevant for that particular situation. Moreover, the reasoning mechanism can either apply the facts and relations from the knowledge base for the purpose of drawing new conclusions from the known information or for the validation of a hypothesis.

An additional advantage of the division of expert systems into three elements concerns the aspect of maintenance. The algorithmic architecture of traditional programs makes maintenance a hazardous enterprise because all procedural actions relate to each other. If one single aspect of the program is changed, the entire program must be adapted or rewritten. Since the components of expert systems are independent, they can be updated or expanded without this having effect on each other. For instance, the inference mechanism can be changed from data oriented into goal oriented, but it will still be able to use the same knowledge from the knowledge base. Reversely, if the knowledge base is expanded with new facts or new relations the inference mechanism does not have to be adjusted as well: it will still be able to consult the knowledge. Furthermore, the user interface does not influence the reasoning process nor the contents of the knowledge base. If the lay-out of the application is improved, neither its knowledge is affected by this nor its reasoning strategies.

3.3 Knowledge representation and reasoning methods

3.3.1 INTRODUCTION

In the previous paragraph, it was shown that an expert system not only consists of a knowledge base, but also of a reasoning mechanism which enables the knowledge to be used. How knowledge can be represented in a knowledge base and how the inference mechanism accomplishes reasoning is shown in this paragraph. Since the early days of expert system development knowledge and reasoning representation has evoked much research (*e.g.* Charniak & McDermott 1985), because it is a complex matter and it turned out that different tasks and their associated knowledge require different representation formalisms. Nowadays many formalisms are available, but in this paragraph we will only introduce the most common ones: predicate logic, decision rules, semantic networks and frames.

3.3.2 PREDICATE LOGIC

One way of representing knowledge is by means of mathematical logic: it is a method to prove a theorem and thus to deduce facts. Pure logic, however, is not easy to computerize and it is not a suitable medium for the representation of procedural knowledge. Hence for computational purposes it is not used in its original format, but in a format that is adapted to the specific demands of a knowledge representation formalism. An example of such a specific logic-based method is PROLOG. This is a programming language which uses a combination of predicate logic, for representing facts (descriptive knowledge) and conventional non-logical functions, for including procedural aspects (Schotel 1987).

Predicate logic is a formal means for describing facts, which are *propositions* that are either true or false. Propositions consist of a predicate and one or more arguments: predicate (object x, object y). Reasoning with this method is based on the concept that a fact is either true or false and that from known facts new facts are logically deducible. Imagine the following archaeological 'knowledge':

the edge of an artefact can be blunt
the edge of an artefact can be sharp
the edge of an artefact can be used for scraping
the edge of an artefact can be used for cutting
the edge of an artefact can be used for piercing
If an edge is blunt, it cannot be sharp, *vice versa*
If an edge is blunt, it can be used in a scraping motion
If an edge is sharp, it can be used in a cutting motion
If an edge is sharp, it can be used in a piercing motion

When representing this by means of a programming language that is based on predicate logic, it could look like:

motion (scraping, blunt edge)
motion (cutting, sharp edge)
motion (piercing, sharp edge)

These three lines would constitute the entire knowledge base. Since expert systems work with a *closed-world assumption*, facts simply do not exist if they have not been made explicit. Therefore, it is not necessary to include exclusions such as 'a blunt edge cannot be used for cutting'. This simple and small knowledge base can already be consulted by a user. A question could for instance be: can we use a blunt edge for a cutting motion? This would look like:[3]

motion (cutting, blunt edge)?

If we pose such a question, the system will try to find the answer in its knowledge base. In this case, it simply checks whether the combination of the objects 'cutting' with 'blunt edge' occurs in the knowledge base. The answer would be negative. The question 'can we use a blunt edge for a scraping motion?' would, however, be answered positively. The dialogue between the user and the system could look like:

user: motion (scraping, blunt edge)?
system: yes

Other questions for which this knowledge base could be used are:

what kind of edge do you need for a scraping motion?
user: motion (scraping, X)?
system: X = blunt edge

or:

show all motions that can be performed with a sharp edge
user: motion (X, sharp edge)?
system: X = cutting
X = piercing

These examples did not really require complex reasoning processes, in fact, it did not require reasoning at all. The application merely had to consult its factual knowledge. Most applications have, of course, much more complicated knowledge bases. They may contain all kinds of functions through which complex relations between specific facts can be represented or inferencing strategies can be expressed and through which complex reasoning processes can be simulated. Fortunately, reasoning can also be achieved through simple logical inferences. This is illustrated by the following example:

a burial is an archaeological feature
a plough mark is an archaeological feature
a mole hole is a biological feature
all archaeological features are indicative of human activity

In a knowledge base this knowledge could be expressed by means of three facts and one relation:

facts: archaeological feature (burial)
archaeological feature (plough mark)
biological feature (mole hole)
relation: human activity (X) := archaeological feature (X)

The last line expresses the relation between the predicates 'archaeological feature' and 'human activity'. It means that the predicate 'human activity' receives (:=) its object (X) from the predicate 'archaeological feature'. The X's are variables of which the values can be exchanged between predicates. In order to obtain a value for X, the inference mechanism can consult the knowledge base. The predicate 'archaeological feature' from the second part of the relation, can get the values from all predicates in the knowledge base that have the same name. In this case, the predicate 'archaeological feature' that is part of the relation can get the values from the predicates 'archaeological feature' that are expressed as facts. These values are 'burial' and 'plough mark'. In other words, by means of the above relation it can be deduced that a burial as well as a plough mark points to human activity. Thus, a question a user could ask this knowledge base is:

user: human activity (burial)?

Since this is not described as a fact, the system has to deduce the answer from the given relation. By means of the rules of logic this can be easily achieved.[4] The answer is positive because the value 'burial' is one of the values which the predicate 'archaeological feature', that is part of the relation, can collect from the lists of facts. Other questions could be:

user: archaeological feature (mole hole)?
system: no

user: human activity (X)?
system: X = burial
X = plough mark

user: human activity (mole hole)?
system: no

One of the main advantages of predicate logic is that it offers a user the possibility to consult the knowledge base in various ways. Even the above simple knowledge bases could answer several questions. This implies that once a knowledge base has been composed, it may serve several purposes. One of the major drawbacks of this representation method, however, is that it already yields large and complex knowledge bases when only small tasks or problems are represented. The problem with such large and complex

knowledge bases is that they are rather inefficient in terms of storage, processing speed and maintenance. For instance, a consultation of such a system means that it checks all facts and relations in order to make as many deductions as possible. As a consequence, all deductions that can be made, will be made, even if they are of no concern for the problem.
Another disadvantage of predicate logic is that it is not very user-friendly because it is quite difficult to learn. Moreover, it is hard to represent all aspects of human knowledge by means of predicate logic. The reasoning of a human expert rarely follows exactly the rules of logic. On the contrary, experts predominantly think in terms of heuristics, defaults, strategies, rules of thumb, etc., which are difficult or sometimes impossible to model into a logic framework. Furthermore, it is not easy to handle exceptions to the rules or uncertainties with this method, because it is such a straightforward formalism for which a fact is either true or false.
These drawbacks imply that this formalism is less suitable to represent the empirical knowledge of domains like archaeology. In most cases it would be very difficult, or even impossible to describe archaeological knowledge in such a generalized way as in the above examples. This does not mean, however, that it is not a useful representation method. In fact, the principle of logical reasoning is employed in all other knowledge representation methods.

3.3.3 PRODUCTION RULES

The second representation formalism, that of *production* or *decision rules*, is more popular than predicate logic. They exploit the power of predicate logic, but do not have the same drawbacks. Production rules have specifically been designed by Shortliffe and Buchanan to encode heuristic knowledge in a simple manner (Davis *et al*. 1977). This formalism combines facts with procedural knowledge (relations) by means of logical deductions, but its syntax is easier to understand than that of predicate logic. It employs IF-THEN constructions that are reminiscent of human inference procedures.
With this method, facts are represented by means of *object-attribute-value triplets* and the relations between the facts by *rules*. An object consists of several attributes and each attribute has a value. At the beginning of a session the objects have usually no value. During the inference process they are gathered if they are needed. They can be received from the user or deduced from other facts. The rules are based on the principle that ‘IF condition X is true, THEN conclusion Z can be made’. The more formal format is:
IF attribute X OF object X has value X, THEN attribute Z OF object Z gets value Z. The knowledge from the first example in the previous paragraph could be represented by means of the following facts and rules:

objects	**attributes**	**values**
motion	cutting	TRUE/FALSE
	scraping	TRUE/FALSE
	piercing	TRUE/FALSE
edge angle	blunt	TRUE/FALSE
	sharp	TRUE/FALSE

rules
1 IF blunt OF edge angle IS TRUE
THEN scraping OF motion IS TRUE
2 IF sharp OF edge angle IS TRUE
THEN cutting OF motion IS TRUE
AND piercing OF motion IS TRUE

The basic process of reasoning with rules is to match facts with the rules that relate to them. This matching can be done in a *backward* or a *forward* direction. The first is called the *backward chaining approach* and the latter the *forward chaining approach*. As was explained in paragraph 3.2, the inferencing process of the backward reasoning approach starts with a predefined goal. In practice, an expert system that is based on backward chaining usually provides the user with a number of goals from which he or she can choose. These goals refer to the hypotheses that the application is able to validate.
If, in our example, the user wants to verify whether his artefact is a cutting tool, the goal of the system becomes ‘cutting OF motion IS TRUE’. Subsequently, the inference engine starts searching for a rule that may confirm this goal, *i.e.* that can conclude that ‘cutting OF motion IS TRUE’. In this case, this is rule number 2. This rule, however, can only draw this conclusion if its own condition ‘sharp OF edge angle IS TRUE’ is acknowledged. Consequently, the inference mechanism now has to search for another rule that can confirm this condition. Alternatively, it may ask the user to provide the additional information. If the knowledge base does not contain rules which may confirm the condition or if the user denies that the edge angle is sharp, then rule number 2 fails and the system concludes that the user’s hypothesis is falsified. If, on the contrary, the user confirms that the edge is sharp, the system can infer that the conclusion of rule number 2 can be drawn and that the user’s hypothesis is correct.
With respect to the forward reasoning approach the deduction process starts when it is triggered due to a changing value of one of the facts from the knowledge base. Such a change can be caused by information that is provided by a user, a database, another computer, or another device like a measuring-instrument. If this happens, the application immediately starts checking which rules are activated, *i.e.* which conclusions can be drawn from this fact, and what consequences this change has on other facts. In our example, the process may start when the user indicates that he has found

an artefact with a sharp edge. As soon as the inference engine receives this message, it starts searching for rules that relate to this fact. Rule number 2 will be found, because it has 'IF sharp OF edge IS TRUE' in its condition. Subsequently, two new facts can be inferred, *i.e.* 'cutting OF motion IS TRUE' and 'piercing OF motion IS TRUE'. Since there will be no more rules triggered by these two facts, the inferencing process will stop and the user receives the conclusions which can be deduced from his observation.
One of the major advantages of a knowledge representation method that is based on production rules, is the ability to employ the so-called *Boolean operators* AND, OR, NOT etc. These operators enable rules to describe both positive and negative relations between objects and the attributes of objects. For instance in rule 2 of the above example, the Boolean operator AND was employed to deduce a multiple conclusion. Usually, these operators can simultaneously be part of the condition of a rule and of its conclusion. This makes it a suitable method to represent especially non-logic or non-linear heuristic knowledge.
Another advantage of decision rules is that its (IF-*condition*, THEN-*conclusion*) syntax represents a more natural, human way of reasoning. Consequently, they are relatively easy to understand and work with. Furthermore, they can represent knowledge of different levels. They can make inferences with facts but they can also contain knowledge about knowledge (meta knowledge) and procedures (see also chapter 5).
Nevertheless, a rule-based system has some disadvantages as well. Like with the pure logic method, the knowledge is still is not very well-structured. Complex systems may contain hundreds of rules and, from a developers point of view, such systems are still difficult to develop, handle and maintain. The more rules you create, the less predictable and orderly the system becomes. Due to the fact that rules influence each other it may become impossible to keep track of all the effects of simple changes (Jackson 1986). Furthermore, rule-based systems are still not as efficient as may be wished. For example, each time an application receives or deduces a new value (a fact), it will check all rules in order to find out which one of them can be activated by that fact. With complex systems it may take a relatively long processing time before a conclusion is drawn. This can be problematic if time is a critical factor, like with process control. That is why Payne & McArthur suggest that "*...when the rules in a system become harder to understand than the equivalent programming code, it is clearly time to explore solutions other than rules.*" (Payne & McArthur 1990: 60).
Furthermore, it has been repeatedly argued that the expressiveness of rules is not optimal for all kinds of expert system applications (see also chapter 2.4.1). Rules may represent complex relations sufficiently explicit and they may, therefore, not explain the underlying decisions in reasoning process adequately enough (*cf.* Clancy 1983).
Another limitation is that with goal-oriented applications the goals must be explicitly incorporated in one of the rules of the knowledge base. This means that this restricts the exploitation of the knowledge: it is only employed for the purpose of confirming these predefined goals, while it could be used for many more purposes. This is in contrast with, for instance, predicate logic. The example in paragraph 3.3.2 showed that predicate logic enables a user to consult a knowledge base in various ways.
Despite these limitations, the representation formalism of production rules is most frequently chosen by system designers: it yields transparent and understandable knowledge bases. Exactly for these reasons it has also been deployed for the construction of WAVES (see chapter 5.5.5).

3.3.4 SEMANTIC NETS

Another way of representing knowledge is by means of a *semantic net*. This method is predominantly used for knowledge that consists of a hierarchical nature. Semantic nets consist of *objects (nodes)* and the *hierarchical relations (links)* between the objects. The relations between the objects have the form of 'IS-A' or 'HAS-A'. Imagine the following hierarchical knowledge:

prehistoric man occupied hunting camps
characteristic for a hunting camp are its flint artefact assemblage
a scraper may belong to this assemblage
an arrowhead may belong to such an assemblage
a scraper has a blunt edge, while an arrowhead has a sharp edge
a blunt edge has an angle of more than 60 degrees
a sharp edge has an angle of less than 60 degrees

This could be represented as:

```
site IS-A hunting camp
        HAS-A flint artefact
                IS-A scraper
                        HAS-A blunt edge
                                HAS-AN angle > 60 degrees
                IS-AN arrowhead
                        HAS-A sharp edge
                                HAS-AN angle < 60 degrees
```

Like all other reasoning methods, the deduction principle is based on a direct comparison of the facts that the user provides and those the knowledge base contains. A user could consult this knowledge base by asking: what type of flint artefact has a sharp edge? The answer will be 'arrowhead HAS-A sharp edge'. Another question could be:
which flint artefacts occur on a hunting site?

```
user:    hunting site HAS-A flint artefact IS-A X?
system:  X IS-A scraper
         X IS-AN arrowhead
```

Semantic nets have another specific way of deducing facts, *i.e.* through *inheritance*. This means that the objects of a lower level automatically receive the characteristics of the upper levels. Regarding the above example, inheritance means that each type of edge angle belongs to a specific edge, to a type of artefact, and to a hunting site. Often, it is however more complicated than this. In fact, any node can be related to any node from the same or from any other level. In this way, knowledge can be passed between any related object. Consequently, the network and thus the directions of the knowledge exchange can be very complicated.
In the above example, the principle of inheritance allows a user to consult the knowledge base in different ways. One could, for instance, verify whether an excavated artefact can be classified as a scraper. This dialogue would look like:

```
user:    IS-A scraper?
system:  HAS-AN angle > 60 degrees?
user:    yes
system:  yes
```

Few systems that are based on structured objects reason by means of inheritance only. Most of them can use some form of forward or backward chaining as well. Examples of these are *when-changed* and *when-needed* methods. A when-changed method means that a command can be attached to a certain attribute. This implies that when the value of such an attribute changes, a particular action is taken, like with the forward chaining mechanism. A when-needed method means that whenever the value of a particular object is needed, an immediate action is taken to retrieve that value. This resembles the goal-oriented backward chaining mechanism.
A hierarchy-oriented formalism such as a semantic network, is very useful for representing hierarchical related knowledge and thus for tasks like object classification. It can structure such knowledge perfectly. Another advantage is its processing speed. Once you have a known fact, only one deduction path will be followed, *i.e.* that of the related nodes. As a consequence, only related facts are deduced without wasting time on irrelevant facts and conclusions.
Unfortunately, this method has also some disadvantages. It is less useful for representing rules-of-thumb or for knowledge that comprises many exceptions. In the above example, for instance, a wooden arrowhead would not be classified as an arrowhead. Exceptions must be handled as separate objects (nodes) with their own relations and a network with many exceptions soon becomes very complex and disordered and, thus, difficult to maintain.

3.3.5 FRAMES

The fourth representation method is based on the concept of *frames*. Frames have many things in common with semantic nets: they also describe facts in terms of objects and they reason through inheritance. One of the differences, however, is that their objects have their own associated attributes which have no links to other frames of the same level. In other words, each node of a frame can consist of several attributes, while in a semantic network each node represents one attribute. The frame based method implies that related knowledge is further grouped together. A semantic network can be used to describe the general structure of a situation or task, while frames describe stereotypical situations. This leads to another difference with semantic nets. Inheritance across the nodes of the same or other levels is impossible. It can only take place in an one-way direction, *i.e.* from the general (parents) levels to the more specific (children's) levels. The strength of frames is that knowledge can be deduced from an object as soon as it is known that this object belongs to a certain class. That object automatically has the same characteristics as its parents.
When the knowledge of the previous example would have to be described by means of frames, it could look like this:

```
Frame 1  SITE TYPE X
         characteristics
               excavated in:
               coordinates:
               environmental data:
               altitude:

Frame 2  EXCAVATED ITEMS
         part-of site type X
         characteristics
               artefact types:
               features:
               samples:

Frame 3  ARTEFACT TYPE SCRAPER
         part-of excavated items
         characteristics
               amount:
               edge shape:
               function:

Frame 4  ARTEFACT number xx
         part-of artefact type scraper
         characteristics
               edge angle:
               raw material:
               find condition:
               coordinates:
```

The inheritance in this example implies that if you ask information on artefact number xx, the system can automatically trace all knowledge that is related to that particular artefact, such as its function, its find context and the type of site it belongs to.
In comparison with the other representation forms, a frame-based knowledge structure is the most easiest to expand and maintain. New frames or relations between frames or characteristics of frames can be unlimitedly added or removed, without influencing the other knowledge and without increasing the complexity of the knowledge base. However, this form is only useful for tasks with perfectly structured knowledge, like classifications.

3.3.6 HYBRID REPRESENTATIONS

Each of the discussed representation methods has its own advantages and disadvantages but each suits a specific problem or application best. It highly depends on the characteristics of the knowledge and the demands of a system which method is used. Moreover, the use of one method does not exclude that of another. Often expert system shells (see paragraph 3.5.3) are hybrid systems that offer a combination of representation methods as well as of goal-oriented and data-oriented inference mechanisms. In that way, the advantages of various methods can be used to build an application that is optimally tailored to its task and the associated knowledge.

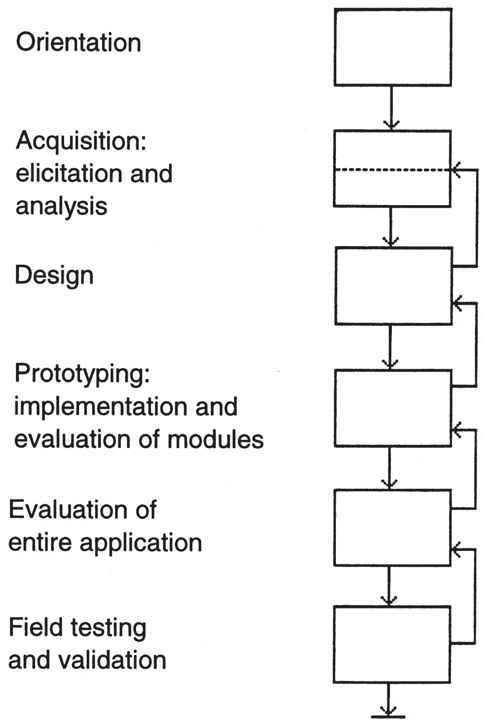
Orientation
Acquisition: elicitation and analysis
Design
Prototyping: implementation and evaluation of modules
Evaluation of entire application
Field testing and validation

Fig. 4. Stages in a top-down expert system development trajectory. Implementation occurs according to the *incremental programming approach*: each stage of the trajectory allows for feedback and comprises the possibility to adapt the result of the previous stages.

3.4 Expert system development process

3.4.1 INTRODUCTION

In the previous paragraph, knowledge bases and reasoning processes were described in an operational state, but it usually requires a considerable effort to reach this stage of an expert system application. The main struggle is to retrieve the right knowledge, to analyze it and to mould it into a model that is effective and efficient for both the computer and its future user. Moreover, before any of the simulated knowledge can be actually consulted, it must be implemented into a computer and thoroughly tested. Consequently, the development of an expert system application that is meant to be used for practical purposes and by independent users, is a complex and time consuming venture. The success of the operation and the subsequent application depends on various aspects, such as the co-operation between the domain expert who is going to provide the required knowledge and the developer, the so-called *knowledge engineer*, the management of the project and the limiting conditions (*cf.* Kulikowski 1989).
In this paragraph the most important aspects of the development process will be discussed. Although there are no unambiguous strategies developed for this kind of projects, a development trajectory can generally be divided into five main components: orientation, acquisition, design, implementation and evaluation. I have treated them as consecutive stages that are prosecuted in a top-down fashion (fig. 4). This implies that the nature of a task and its knowledge is the starting point of the development process and that the characteristics of the application will be attuned to this.
In this chapter these stages are mainly described from a theoretical point of view. In chapter 5 they will be encountered in the practical context of the construction of WAVES.

3.4.2 ORIENTATION

In paragraph 3.1 (fig. 2), it was shown that expert systems can be employed for a large variety of tasks. This does not mean however that they can perform any task, or that all problems or tasks need to be handled by an expert system application. Often they can be perfectly-well carried out by a standard algorithm of a conventional program. The orientation phase of the development process, therefore, implies that it is studied whether the problem domain or task suits a knowledge-based approach.
One of the aspects which call for attention during the orientation phase is the nature of the task and its related knowledge.

Expert system applications are most successful when they are employed for tasks which can be clearly defined and which have a limited complexity. In respect to this complexity a useful directive may be that a human expert must be able to solve a particular task or problem within a couple of hours. Even more important, however, is that the required knowledge can be made explicit by means of formal representation methods. This may be problematic because it has been experienced that experts may find it difficult to describe their subjective knowledge explicitly and to explain the underlying reasoning processes that they apply (Kidd 1987: 3). Moreover, some expert knowledge can hardly be formalized because of its subjective character and because of the uncertain relations between the facts or the observed phenomena. Such difficulties may cause serious problems for an application building process and should be discovered and, if possible, bypassed before the development trajectory is started.

Since the success of an application building process highly depends on the character and possibilities of the knowledge domain, some authors (*e.g*. De Witte & Kwee 1987; Prerau 1989) have formulated directives for a *suitability evaluation* of the selected subject, such as:

- one should only select tasks for which sufficient knowledge is available, either in writing or in the form of an expert;
- the problem and its associated tasks must be well-defined and decomposable into smaller problems and tasks in order to keep the project manageable;
- the problem's solutions must be assessable;
- it must be possible to program the problem within the limitations of the available time and money and of the hardware and software facilities;
- the expert must be sufficiently available and co-operative to share his or her knowledge;
- the end-users must be able and willing to use an expert system;
- it must be acceptable for end-users that the system shall not be a duplicate of the real expert, but only a limited extraction;
- the problem cannot be handled sufficiently by conventional programming methods.

It must be stressed, however, that even if these conditions are met, the success of a development project is not guaranteed. Reversely, if some of the conditions cannot be met, this does not automatically mean that a project has no chance of succeeding.

3.4.3 KNOWLEDGE ACQUISITION

Once the decision is made to build a knowledge-based application, the first development phase begins, *i.e.* the acquisition of the required knowledge. In order to enable a program to simulate an expert's reasoning, it must be provided with expert knowledge. The acquisition phase consists of two activities: the elicitation of the knowledge and its subsequent analysis. The elicitation refers to the entire process of making an inventory of the knowledge, tasks and procedures that are involved in a task and of gathering it from the available sources. The analysis implies that the elicited knowledge is studied, organised and finally structured into a model.

In many cases it is far from easy to elicit knowledge (see chapter 5). Usually, the knowledge that is involved in expert system applications not only consists of formal facts and theories (gained by education), but also of *heuristic* elements, such as rules-of-thumb that are based on intuition and experience. The subjective nature of the heuristic knowledge makes it difficult to retrieve the expert's underlying reasoning processes and, therefore, to gather all the necessary information. Experts are not used to give a detailed or step by step description of their approach to a problem and may feel uncomfortable in doing so. Especially knowledge that is based on intuition may be hard to make explicit.

The engineer, on the other hand, may have difficulties to perceive the approach of an expert and to understand the underlying reasoning processes and must try to get acquainted with the jargon and the subject of the domain practitioner. There are mainly four methods to withdraw information from the expert. An engineer usually employs a combination of them. These methods are interviewing, observation, introspection, and participation. With an interview, the system developer initiates the elicitation trajectory and the expert is questioned. An observation implies that the knowledge engineer watches the activities of an expert at work. Introspection means that the expert describes and explains all his actions and thoughts while he carries out particular case studies. Participation implies that the engineer tries to perform some of the procedures or actions himself.

After the elicitation has finished, the next step is to analyze the information that was gathered and to divide it into small parts. This means that the relating items must be recovered, brought together and structured into distinctive tasks, *i.e.* the hypotheses are separated from the evidence and actions. Subsequently, each reasoning task is characterized and all actions are specified. Finally, all knowledge is captured in descriptive (factual) and procedural knowledge models. During this phase a frequent feedback with the expert is essential for the verification of all parts of the model. It may also be useful to consult other experts for an evaluation of the model. Furthermore, if it is possible to include the viewpoints of several experts, an application will be more complete and more easily accepted by the user (see also chapters 5 and 7).

Since knowledge acquisition holds the key to success for any application, it is an important topic in the field of expert system research. Many scientists have developed methods, techniques or tactics that can support or facilitate this aspect of the development process (*e.g.* Kidd 1987; Roth & Woods 1989). This research even entails the automatic acquisition and synthesis of knowledge, which implies that particular types of knowledge are elicited by means of specially designed algorithms (*cf.* Hart 1987). Despite these more formal approaches, knowledge acquisition remains one of the most difficult phases in the expert system building trajectory.

3.4.4 DESIGN, IMPLEMENTATION AND EVALUATION

The knowledge models which result from the acquisition phase, provide the basis for the third phase in the development process, the design of the application. It means that a conceptual map is made of what the application is going to look like, how it will carry out its task, how the knowledge will be represented, what inference mechanism will be required, what explanatory facilities will be incorporated, what other aspects should be taken into consideration in the communication with the user, etc. It is important that a design is mature, because it is not only decisive for the functionality of the final application but also for its maintenance. Consequently, decisions must be made on the hardware and software that will be needed to effectuate the design.

In principle, if the design of an application has finished, the real programming, *i.e.* the implementation begins. This implies that the required software and hardware is selected and that the knowledge models are implemented by means of knowledge representation methods and inference mechanisms. The implementation process can be carried out according to three approaches, *i.e. linear programming*, *(rapid) prototyping* and *incremental programming*. Linear programming is a classical method. It means that an application is implemented and evaluated as a complete package. Often such applications are very well-developed technically. But their practical use may be disappointing for the end-users because this approach usually does not give them much opportunity to participate in the development process. Another risk is that the evaluation can reveal problems which affect the entire application and which may require severe and time-consuming adaptations.

It is important to involve the end-user in the development process because they determine whether the final system is accepted or not. It is rather obvious that a system which has the users' approval, will be more easily accepted than a new system which suddenly is imposed on them by some knowledge engineer. Prototyping is, therefore, considered to be an important part of the development process (*e.g.* Hayes-Roth *et al.* 1983; Bratko 1989; Kahn & Bauer 1989; Whipp & Lewis 1989; Van den Broek *et al.* 1990). It means that small tasks or modules are implemented and evaluated separately and that the user participates regularly in the development process. The user's feedback is very important for this approach.

In its most extreme form rapid prototyping implies that the application's final shape is predominantly established by means of a process of trial-and-error in which the user's wish is the developer's command. The underlying concept is that everything is allowed as long as it makes the user happy. Unfortunately, an application that is built in such a way lacks a firm and coherent structure. It may lead to an ill-considered and opaque application which is very difficult to maintain because the consequences of changes cannot be predicted (Van den Broek *et al.* 1990).

The third method, incremental programming is a combination of the previous two. With this method the system is divided into small parts or increments and each of these is implemented according to the linear method and subsequently evaluated by means of prototyping (fig. 4).

If necessary, the design of a module can be updated on the basis of each evaluation. Moreover, the consistency between the modules is constantly controlled. Hence, the user is involved in the development process, but the application's design remains the foundation for the implementation process. In this way, the system's maintenance is secured. Since the incremental method combines the advantages of the former two, it is the most efficient approach.

When the process of implementation has been completed, the resulting system has to be tested and evaluated as a whole. In particular, if an application will be used for practical purposes its quality needs to be assessed thoroughly. Unfortunately, there is no generally accepted validation method for each approach has its limitations (Hollnagel 1989: 377). In chapter 7 this aspect will be discussed in more detail. Finally, if the evaluation has been carried out the application can be made operational and released for practical use.

3.5 Implementation tools

3.5.1 INTRODUCTION

The actual implementation of an application is facilitated by using a tool that is dedicated to this, although sometimes conventional procedural languages are used as well (Waterman 1986; Alty 1989).[5] Various dedicated tools are available. The two most commonly used are *a programming language* and a *shell*. In the beginning, applications were built with general purpose expert system building languages, like LISP and PROLOG. Ever since, many different languages have been developed which are dedicated to building knowledge bases with particular representation formalisms. Examples of these are logic-based languages, rule-based languages, and frame-based languages. The choice of a language, therefore, depends on which of these formalisms will suit the application.

When more people became (commercially) interested in expert systems, new methods were developed to simplify the expert system building process and to improve the accessibility of this technology. In this respect, especially the success of the application MYCIN (see paragraph 3.1) has been important for a further development of the expert system technology. MYCIN has served as a blue-print for the *expert system shell*. This is an empty expert system application: a skeleton which provides representational, reasoning and communication facilities, but in which no knowledge is incorporated. In a way, a shell may be compared with a baby: it is equipped with all facilities and functions (brains and sense-organs) that are necessary to store knowledge and to communicate, but it does not know anything and cannot speak yet. Just like a baby can learn English as easy as Dutch or Swahili, a shell can be filled with knowledge from any domain.

Using a shell does not mean that it suddenly becomes easy to develop a knowledge-based system. It only enables domain specialists, like archaeologists, with some knowledge of computing to implant their own expertise or that of colleagues into a system. This is opposite to the traditional situation in which system development was solely reserved for computing specialists, who often had no affinity with the concerning domain. It does not mean, however, that shells make knowledge engineers superfluous. On the contrary, the development of really complicated applications must be entrusted to a person educated for such a job, and not to an archaeologist who happens to know something of expert systems, like myself. Since specialists in knowledge engineering master a great diversity of sometimes very complex knowledge representation techniques, their contribution will surely diminish the risk of ending up with a useless system.

3.5.2 PROGRAMMING LANGUAGES VERSUS SHELLS

For some applications conventional programming languages can be used. However, for more complex problems for which no straightforward solutions can be programmed, the algorithm becomes very complex and, therefore, difficult to maintain. Since expert systems need maintenance regularly, conventional programs are often less suitable. The virtue of expert system programming languages is that they are highly specialized to a particular representation and inferencing method. This makes them quite sophisticated tools. Another advantage is that a language has almost no procedural restrictions, because the engineer develops the procedures him or herself. This means that he or she can build an application in such a way that it perfectly fits its task. Although it may need complex and trickery programming, it is said that "*...any facility missing in a language can be provided by programming.*" (Alty 1989: 198). Languages allow for custom-made applications.

Languages have their disadvantages as well, however. The fact that their representational and inferencing possibilities are predominantly dedicated to a particular type of task, means that the knowledge of the application that is being built, must fit this approach seamless. If this is not the case, it may be difficult to represent the various types of knowledge in an equally effective way. Another drawback is that it may take considerable time to learn a language. Especially system developers who have not been trained in the computing profession, may experience difficulties in exploring and employing their abilities. Furthermore, it takes a long time to actually implement an application, simply because all of its aspects must be explicitly programmed. Languages offer few standardized and ready-to-use procedural tasks.

That is why shells were developed. The first examplars were nothing more than abstractions of existing expert system applications, *i.e.* empty structures. They consisted of a representation method and an inference mechanism, which could be filled with knowledge. This enabled a developer to build an application without having to construct and program its entire architecture. These simple shells were soon succeeded by shells which offer combinations of different representation and inference methods, fancy user interfaces and built-in facilities to communicate with databases or other devices. Some shells even developed into sophisticated implementation environments, which also provide facilities to ease the actual programming, such as editors and automatic debuggers (Payne & McArthur 1990: 51). Shells became popular tools. Many commercial companies started to develop their own. Consequently, this has resulted in a large assortment of these tools.[6]

Compared with languages, shells are more easy to learn and work with. The developer no longer has to program complex inference mechanisms, user interfaces, etc. The built-in facilities make it fairly easy, even for an unexperienced system developer, to make an application that is able to simulate rather complicated reasoning processes.

Shells have disadvantages as well. The main objection against shells is that they may restrict a programmer's creativity to represent expert knowledge. Because of the fact that the facilities of a shell are prefabricated, its possibilities are bordered. Often they have been designed for particular tasks, like diagnosing or classification, which means that it is difficult to employ these special-purpose shells for tasks which deviate from these approaches (*cf.* Alty 1989). Using a shell implies that the engineer has to tune either the abilities of an application with the shell or vice versa. Generally, this is not as intervening as it may sound. Most shells are well-equipped and enable various combinations of knowledge representation methods and inference strategies to minimise this potential drawback. Moreover, if a particular shell cannot comply with the requirements one can simply choose another. Since in recent years all kinds of dedicated shells have been developed to accommodate specific kinds of tasks, nowadays many applications can be developed by means of a shell.

There are no clear guidelines as to which tool suits a problem best. In general, languages are more generally applicable, but they offer less facilities. Moreover, in both options the development costs for an operational expert system are more or less equal: shells require a higher initial expense, but languages require more implementation time. The hardware requirements are comparable as well. Hence, the choice of the implementation tool often depends on the character of the application's task (De Swaan Arons 1991) and on the personal experience or preference of the knowledge engineer.

notes

1 In general, the terms 'expert systems' and 'knowledge-based systems' are used indifferently. Some authors, however, make explicit distinctions. In some cases different terms are used in order to draw a distinction between systems that concern general knowledge and the real expert systems which employ highly specialized expertise (*cf* Van Praag *et al*. 1988: 45). In other cases it is meant to differentiate between the various problem solving methods on which the systems are based. For instance, searching through a large knowledge base in order to compare knowledge would be characteristic for a knowledge-based system, while only a true imitation of an expert's reasoning strategy would deserve the predicate 'expert system' (Nijssen 1992: 13). Furthermore, it is argued that the term 'expert system' may be misleading. According to Winograd and Flores "*There is a danger inherent in the label 'expert system'. When we talk of a human expert we connote someone whose depth of understanding serves not only to solve specific well-formulated problems, but also to put them into a larger context. We distinguish between experts and idiot savants. Calling a program an expert is misleading... The misrepresentation may be useful for those who are trying to get research funding or sell such programs, but it can lead to inappropriate expectations by those who attempt to use them.*" (Winograd & Flores 1986: 132). Since the term 'expert system' suggests that it simulates or approaches an expert's reasoning and problem solving abilities and may cause high expectations that cannot always be fulfilled, it would probably be better to use the term knowledge-based system. However, this is only logical from the point of view of a developer. Compared with the human expert of which he tries to simulate the reasoning processes, he may consider an application to reach a much lower level of performance. The user, on the other hand, is a layman: in his eyes the application may still act as an expert. The application that will be discussed in this study (WAVES) contains and processes only basic knowledge and handles and interprets data of a non-problematic character best, but students may experience the advices of the system comparable to that of a human expert. For this reason, and because it is a common term, I have not avoided to use the word 'expert system'. Moreover, I considered it irrelevant in the context of this thesis to make a careful weighing in favour of one of them.

2 The external world may be a person, a database or another device.

3 The format that is used in these examples does not represent the format that is used in an existing language or other representation method.

4 Besides by logical deductions, predicate logic can achieve reasoning through more complicated ways like backtracking and unification (*e.g*. Schotel 1987; Lucas & van der Gaag 1988), but it is beyond the scope of this thesis to discuss these methods as well.

5 Examples of procedural or high level languages are FORTRAN, COBOL, BASIC, PASCAL, C, etc.

6 Overviews and comparisons of different types of shells are, amongst others, given by Jackson 1986; Alty 1989; Filby *et al*. 1989; Payne & McArthur 1990.