A sound basis for the generation of explanations in Expert Systems

A. Bruffaerts, E. Henin and A. Pirotte

Introduction

Expert systems solve problems that require human expertise in specialized domains. They work by exploiting knowledge ordinarily acquired from human experts and simulate some of the functions of a human expert in the application domain. Expert-system techniques currently work well only in carefully delimited domains, with strict limitations on context, relevance, and ‘common sense’. While the most enthusiastic advocates of the technology argue that this is only a provisional limitation \[1\], others are more sceptical and warn of connotations of the term expert not warranted by the achievements or the prospects of the technology \[2\].

Knowledge in expert systems resides in a knowledge base (KB from now on) in the form of more or less independent items of knowledge frequently called rules. In some systems, rules have a rather declarative interpretation close to the implication of logic (IF premise THEN conclusion), while, in others, the intended interpretation is more procedural (IF condition THEN action). Knowledge for solving a problem is invoked by a computer program called an inference engine. By examining data derived from a query and representing a problem being solved, the inference engine selects appropriate rules in the KB and executes them on the data for the problem (see fig. 1).

The performance of an expert system depends on the adequacy of its knowledge and on how effectively this knowledge is used, hence the term knowledge-based system, often used as a synonym.

Expert-system technology is often used in applications where a substantial part of the knowledge is heuristic. The experts’ understanding of the domain, although sufficient to address certain classes of problems adequately is changing and imperfect. Therefore, the knowledge represented in expert systems, like the experts’ knowledge, varies in its depth, in its stability with time, and in the degree of confidence that experts have in it. Also, application domains of interest for expert systems are usually specialized technical domains where the ordinary user’s grasp of the subject is far less detailed than the expertise of the specialist.

Because expert systems have these characteristics, simply stating the solution to a problem is usually insufficient to convince users of its adequacy. Further detail about the solution must be available, reasons should be given for it and it must be seen to be

---

Dr A. Bruffaerts, Dr E. Henin and Dr A. Pirotte are with Philips Research Laboratory, Brussels, Belgium. This work was partially supported by the Commission of the European Communities, under Project ESTEAM-316 of the ESPRIT Programme.

---

\[1\] E. A. Feigenbaum and P. McCorduck, The fifth generation: artificial intelligence and Japan’s computer challenge to the world, Addison-Wesley, Reading 1983.

justified. In human interactions, these activities are part of the work of the human expert and they correspond exactly to the ordinary notion of 'explanation': explaining is making something clear or giving reasons for it. If expert systems are to exhibit expert

### Varieties of explanations

The most obvious users of explanations are the non-expert users of an expert system. For these users it is necessary to justify the solutions, i.e. convince them that the solutions are correct and adequate. This conviction will follow from: (1) a basic understanding of the domain of application, (2) the acceptance of the underlying hypotheses from which the system derives a solution, and (3) the realization that the steps in the reasoning of the derivation are reasonable.

The amount of justification depends on the application and on the users. A conventional computer program (i.e. not an expert system) that inverts a matrix, for example, will be trusted if its users are confident that it codes a classical, well-known and well-analysed algorithm for doing the work. On the other hand, the adequacy of the solution provided by a program which advises users on how to invest their savings cannot be characterized by universally accepted criteria or hypotheses. This difficulty is not due to the program itself nor to the fact that the task is accomplished by a program. It is inherent in a domain where few established hypotheses are accepted by all the human experts and where some of the steps in the experts' reasoning depends more on their experience in the domain than on causal principles. This kind of reasoning, referred to as heuristic reasoning, is an essential part of human expertise. 'Explanation' thus implies: (1) precisely stating the hypotheses behind the reasoning leading to a solution so that users may decide whether they agree with them and (2) showing that the derivation of the conclusions from the hypotheses is reasonable.

If the user is to understand a solution, then it is preferable for him to be able to discuss it and ask for alternative solutions. The non-expert user may also be interested in answers to general questions, not directly linked to a particular problem being worked on, about the application domain, about its terminology, its principles and the methods of reasoning. In this educational role, a system or an expert teaches the basics of the application domain to the user.

The system is intended for the end-users, of course, but the explanations produced by an expert system will also be of value to 'knowledge engineers' and experts in the application domain. Knowledge engineers interact with the expert system, first to create the initial version of the system, and later to control its operation and extend its capabilities. They have to monitor and debug the system at several levels. Explanations can provide confidence and make errors more apparent than just by tracing the execution of the program. Human experts are the source of the knowledge represented in the system. Explanations

---

**Fig. 1.** Simple (pure) problem-solving by an expert system. The problem-solver PS in the expert system is addressed with a query Q. An inference engine IE selects and executes rules from the knowledge base KB to derive an answer A.
can give them a better mastery of the knowledge represented in the system and will help them to clear up misunderstandings. They can use the system to test the validity and compatibility of new items of knowledge or of strategies they are thinking of adding to the KB.

The main differences in the needs of the various kinds of users concern: (1) the level of detail at which explanations should be formulated and (2) the discrepancy between the model of the domain that the user has in mind and the model encoded by the expert system. These differences affect both the content and the structure of adequate explanations.

One difference concerns the degree of familiarity with the internal workings of the expert system. Most end-users will want to address the system and to receive its output in some kind of natural language, so as to minimize their interaction with the computational mechanisms of the system. The natural language can be, for example, a simple version of English adapted to the specific application domain. On the other hand, when knowledge engineers are monitoring the system they will want something more like an operational trace of its operation. They will want to know, say, which item of knowledge was activated at a given point in the formation of a solution.

Another difference concerns the level of understanding about the domain of expertise. Experts must be able to ask for concise explanations focused on particular items of knowledge, while a first-time user may need a detailed justification of the simplest answers, explicitly linked to the basic elements of the application domain. But even so, end-users would probably want the level of detail in the explanations to evolve as they become more expert themselves. Thus, a general requirement for a good explanation system is that it should deliver explanations that can adapt to the needs of different classes of users.

Research on the production of explanations addresses two questions: (1) what is the knowledge necessary for good explanations and (2) which tools and strategies will be adequate for representing that knowledge and for using it when generating explanations.

**Early attempts at generating explanations**

Explanations produced by early expert systems consist essentially of a trace of the steps leading to a solution \[^1\][^4]\(^n\): \((fig. 2)\). These systems answer questions about the derivation of a solution by displaying the complete trace of rule applications in the process of solving the problem (\(how\)-explanations). Whenever they request additional information from the user, the systems can explain why the new information is necessary and how it will be used, by displaying a trace of their current reasoning (\(why\)-explanations). These systems are also able to explain why they did not draw a conclusion that the user had in mind (\(why\)-not-explanations).

However, tracing the reasoning of the system merely describes its behaviour without fully justifying it. Besides accounting for a solution, adequate explanations should also be able to give the reasons for choosing the steps that led to it, by justifying these steps and their validity by references to general principles of the application domain.

Sometimes, the reasoning followed in solving a problem does not use strong causal arguments but relies instead on empirical associations or heuristics (rules of thumb) that the human expert has learned through experience. The KB then contains a representation of the heuristic methods that are more efficient because they short-circuit explicit causal reasoning. Such knowledge is sometimes called 'compiled' knowledge.

However, the reasoning reflected in a convincing explanation may need to be more detailed than that necessary for problem-solving alone. Some of the 'missing' steps must then be made explicit — provided

---

the corresponding knowledge is available, of course. With the passage of time, researchers have come to realize that more is required than purely performance-oriented knowledge, and that some knowledge must also be represented in the $KB$ for giving explanations.

This trend towards the incorporation of extra explanatory knowledge in the $KB$ runs in parallel with the evolution of expert-system technology in such a way that there is a clear distinction between different types of knowledge to make the $KB$ easier to use and modify. Various types of knowledge come into the picture:

- strategic knowledge or knowledge about domain principles which explicitly represents problem-solving methods,
- structural knowledge or terminology relating to the application domain, and
- support or descriptive knowledge which encodes the causal model of the application domain [5] [6] (see Fig. 3).

**Dialogue manager and problem-solver**

We believe that, in order to generate versatile and adequate explanations, an expert system should consist of at least two cooperating components, which we call the problem-solver and the dialogue manager.

The problem-solver ($PS$ from now on) works with symbols, like any computer program. These symbols are intended to represent domain knowledge in the $KB$, but do not have any significance in themselves. The intended interpretation of this symbolic knowledge resides in the concepts and principles of the application domain as these are perceived by experts and knowledge engineers. In more technical terms, the $PS$ and its $KB$ encode a symbolic system whose standard model is the real world as perceived by experts and knowledge engineers.

The interface to the $PS$ is a symbolic query language. User problems must be transformed into symbolic queries before the $PS$ can solve them. Similarly, an answer from the $PS$ is a string of symbols, derived from the query and the symbolic contents of the $KB$, which must be translated back into a suitable model of the application domain. So that they can put problems to the $PS$ in the interface language and receive the answers in the same language, users must be familiar with the standard interpretations for symbols. This is usually the case for domain experts and knowledge engineers. Non-expert users, on the other hand, require more help with interpretation.

The dialogue manager ($DM$ from now on) acts as a mediator in such a step. Its input function consists in translating user requests into corresponding queries that can be accepted by the $PS$ (see Fig. 4). Conversely, in its output function, it translates the symbolic answer produced by the $PS$ into the user model of the application domain. This user model can be different from the expert's standard model: this is illustrated by the misunderstandings that occur in real life between a human expert and a non-expert.

The syntax for communicating with the user can be rudimentary, like computer menus, or it can take a more complicated form, like a manageable subset of a natural language. For example, the output function of the $DM$ may be no more than generating natural-language paraphrases of $PS$ answers from pre-designed ('canned') natural language templates [3]. But comprehensive support for natural language interaction not only requires that the $DM$ shall understand the literal meaning of user utterances; it also requires that it shall recognize user intentions and relate them to system capabilities and to system expectations about the way in which users formulate problems. This calls for advanced techniques and knowledge structures for dialogue management [7], such as planning and monitoring the progress of dialogues or understanding what the user means (e.g. user utterances that imply requests without actually saying so). To eliminate mismatch between the experts' model and the users' intuition, a good $DM$ must also update data about the users themselves, reflecting their current familiarity with the domain, and the $DM$ must
choose, for each user and in each situation, the most appropriate way of referring to domain concepts [8]. The more naive the non-expert user, the larger the discrepancy between the experts' model and the user's model is likely to be, and the more important the role of the DM. An alternative to a complex DM is a human mediator between the system and the non-expert users. For a financial counselling system, for instance, this might be a member of the bank staff.

Research on the generation of explanations is intended to improve the operation of some component or other of the architecture. Work on the DM side is concerned with improving the presentation of explanations to users. This covers a vast area with several difficult and essentially unsolved problems, like the true understanding of natural language. Work on the PS side is concerned with the identification of the knowledge that should be represented, the organization of that knowledge in the KB, and the design of strategies for generating satisfactory explanations from the KB. When both components were tackled simultaneously, there was often confusion.

Our work is focused on the PS part of the architecture. Our objective is to define a formalism with tools to represent several types of knowledge that will give meaningful answers to queries and explanations.

Generation of explanations by the problem-solver

A logical framework for explanations

Logic as specification tool. To provide a sound framework for explanations, reasoning based on the contents of the KB must be possible, and it must also be possible to specify the relationship between an answer produced by the PS and the knowledge that was used to produce it. In practice, this reasoning is only feasible if the KB and its relationships with a PS answer are described in a precise declarative manner, i.e. without references to the execution strategy of the inference engine of the PS.

One of the most widely used and best understood frameworks for formulating precise declarative specifications is formal logic (in practice, the first-order predicate calculus and its extensions). First-order logic has long been studied as a formalism for modelling

References:
sound reasoning. It is therefore at least natural (and
we think it is essential) that the generation of explana-
tions should be grounded in such a framework, in
which the relationships between a request and its an-
swer can be precisely stated.

In principle, this can be done by associating every
KB, irrespective of its implementation formalism,
with a logical theory that is its declarative reading (see
Table I). The axioms of the theory express a declar-
ative interpretation of the rules of the KB. The evalua-
tion of a query by the action of the inference engine
on the KB is reflected, in the logical theory, as a for-
mal deduction from the axioms. The answer com-
puted for the query is then interpreted as a theorem of
the theory. The explanations or justifications of the
validity of the answer can be derived from a proof tree
produced by the formal deduction, that is, a struc-
tured record of the inference rules applied to constitu-
t a proof.

Logic-based versus rule-based formalism as implementa-
tion tool. The interpretation of the execution of the
PS inference engine as a formal deduction process
requires that it should be possible to describe this in-
ference engine in terms of the laws of logic, which is
not always the case.

The computer language PROLOG [9], for instance,
does provide such compatibility, at least when re-
stricted to its logical features. The rules of a KB writ-
en in PROLOG have a direct declarative reading as log-
ical implications of the form IF premise THEN conclu-
sion. The premise part consists of a logical formula
containing logical connectives like AND, OR, NOT; the
conclusion is an atomic formula (i.e. an indivisible
formula). Examples of simple rules are given below
(small print). The PROLOG inference engine can be in-
terpreted as making deductions from the theory asso-
ciated with the KB [10].

When PROLOG is restricted to its logical features, rules have
a direct declarative reading. Examples of simple rules are:

\[ p(X) \iff t(X) \]
\[ p(X) \iff s(X) \]

The first rule, for example, is as follows: if the premise \( t(X) \) is true,
then so is the conclusion \( p(X) \) (with the usual notation for first-
order logic, this is the implication: \( \forall X (t(X) \Rightarrow p(X)) \), i.e. each value
of \( X \) that makes \( t \) true also makes \( p \) true.

In contrast, many expert systems are implemented
in formalisms derived from the early rule-based
systems. In such systems the interpretation of a
rule is more operational (IF conditions THEN action):
roughly, when all conditions in a rule are satisfied with
data derived from the query and the contents of the
KB, then the action part is executed by the inference
engine. The action usually contributes to enabling
conditions of other rules. The order in which rules are
executed by the inference engine, and therefore the
order in which the actions are performed, is often so
bizarre that it is difficult to characterize the KB log-
ically. For a KB of some complexity, the definition of

<table>
<thead>
<tr>
<th>Knowledge base</th>
<th>Logical theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
<td>Axiom</td>
</tr>
<tr>
<td>Query evaluation</td>
<td>Formal deduction</td>
</tr>
<tr>
<td>Answer</td>
<td>Theorem</td>
</tr>
<tr>
<td>Evaluation trace</td>
<td>Proof tree</td>
</tr>
</tbody>
</table>

rules becomes cluttered up with programming tricks
that enforce an appropriate sequence of rule execu-
tion with the inadequate strategy of the inference
engine. These programming tricks also obscure the
logical meaning of the KB [11] and sound reasoning
about its contents or sound justification of an answer
from the PS becomes practically impossible.

If the strictest attention is paid to detail, it is not
necessarily impossible to code a KB in a rule-based
system in a manner that reflects clear declarative
semantics. But if anything more than ad hoc explana-
tion is contemplated, we think it would be more con-
venient to use a logic-based framework.

Trace-based explanations

Even when it is feasible in practice to construct a
logical version of the KB, the answers from the PS,
and their relationships, we are still left with the issue
of just which inference system and logical theory to
choose for expressing them. In particular, the choice
of the inference system (i.e. the set of inference rules
available for performing formal deductions) influ-
ences the shape of the proof tree, which records the
inference rules applied to establish that a query is a
theorem of the theory associated with the KB.

Most procedures for logic-based problem-solving
(including PROLOG) are based on a version of the res-
olution principle. Resolution is a powerful, general,
and efficient inference rule. Therefore conventional
logic-based systems [4][12][13] derive their how-, why-
, and why-not-explanations from a trace of the evalua-
tion process that records all the resolution steps per-
formed by the inference engine during the evaluation
of the query.
However, there are problems with the direct use of resolution as a basis for the generation of explanations (we also refer to the evaluation tree or trace as the deduction trace with resolution to contrast it with the proof tree on which explanations are based).

First, resolution is not very suitable for the production of human-readable proofs. For producing clearer proof trees, a ‘natural deduction’ inference system is better: it consists of clearer inference rules, similar to classical syllogisms, and therefore closer to human reasoning, with each inference and each rule more specific than resolution, so that more rules are required.

Another problem is to be found in the explanation of failure, i.e. in the strategy for why-not-explanations. Fig. 5 shows the evaluation tree formed by the inference engine of PROLOG when executing the query \( p(a) \) on the simple \( KB \) consisting only of the rules mentioned above. All attempts to establish the query \( p(a) \) end in failure, as shown in fig. 5. So, in this conventional logic-based-system approach, the why-not-explanation reviews the branches of the tree in turn, which all led to dead ends (failure branches):

1. an attempt to establish \( p(a) \) failed because \( s(a) \) failed.
2. an attempt to establish \( p(a) \) failed because \( t(a) \) failed.

With the given \( KB \), there is no other possibility for verifying the query \( p(a) \), so this situation can indeed be interpreted as justifying the failure of the query. Thus, while how-explanations justify an answer to a successful query by producing the branch of the evaluation tree that led to the answer, why-not-explanations have to justify that all attempts to produce a solution failed. In this conventional approach, how-explanations are essentially equivalent to a proof, while why-not-explanations are not, since in essence they list logical sentences which, if true, would lead to the production of an answer. Any single failure branch accounts for only a part of the justification of the occurrence of a failure. Nevertheless, with a more powerful logical theory for modelling negation, the failure to verify a query (\( p(a) \) in our example) as a theorem of the theory associated with the \( KB \) would be equivalent to the fact that the negation of the query \( p(a) \) is a theorem in that theory. The proof tree of the negated query then contains all the information necessary to justify the failure of the original query.

We therefore designed an approach of this nature that generates actual proofs for both how- and why-not-explanations. Our inference system converts any finite evaluation tree of the PROLOG inference engine into an equivalent natural deduction proof in a theory defining the logical contents of the \( KB \). We also designed the corresponding interpreter to form a proof tree for any query. Successes and failures of query evaluation can then be treated more symmetrically, since both can be justified by a proof (tree) as theorems of the theory.

For our system, a proof tree giving a general account of the failure of the example query \( p(a) \) looks like the diagram in fig. 6. Each node records an inference rule that has been applied to prove that the negation of the query not \( p(a) \) is a theorem. An actual proof of the query then states: \( p(a) \) is FALSE because \((s(a) \lor t(a))\) is FALSE, that is, because both \( s(a) \) and \( t(a) \) are FALSE.

Decorating proof trees with extra-logical information

Proof trees may also be refined and ‘decorated’ with extra-logical information to provide more details:

- \( p(a) \)
- \( p(X) \lor s(X) \)
- \( t(a) \)

Fig. 5. Evaluation tree for \( p(a) \). Each 3-part node records an inference step (with resolution) by giving: (1) a goal (formula whose verification is attempted); (2) the inference rule applied (resolution with a \( KB \) rule); (3) the result of performing the inference step (one or more formulae to be verified). For example, from \( p(a) \) and the rule \( p(X) \lor s(X) \), resolution infers \( s(a) \). The figure shows two attempts to verify \( p(a) \) that failed successively (the branching in the tree is an or-branching), after noticing that \( s(a) \) (or \( t(a) \)).
tailed input to the DM. We have described the definition and the implementation of a formalism that marks the premises of a rule as either contextual, essential, or auxiliary, to distinguish their specific role. The formalism also handles sets and parametrized formulae while retaining the possibility of generating proof trees. Sets and parametrized formulae are convenient for formulating constraints, i.e. logical formulae that state properties of the solution and are constructed incrementally as problem-solving proceeds.

Fig. 6. A proof tree for \( \neg p(a) \). The inference steps are based on a natural deduction system. There is really only one proof (one branch) in the tree (the branching at the bottom is an \( \text{AND} \)-branching): to report success itself, the parent node waits for both \( \neg s(o) \) and \( \neg t(o) \) to report success.

With a uniform representation of problem-solving knowledge and explanatory knowledge, the formation of proof trees becomes a powerful and general mechanism for exploiting that knowledge. Indeed, the same technique can be used for tracing both the computation of a solution by the \( \text{PS} \) and the subsequent application of explanatory knowledge to that solution.

More conceptual explanations

As argued above in the section on ‘Early attempts at generating explanations’, it is also interesting to cater for explanations not directly based on the \( \text{PS} \) proof tree of some problem-solving query. These require dedicated domain knowledge, which may relate to the terminology, the conceptual structure, or the causal model of the domain. For it to be available, such problem knowledge has to be encoded into the \( \text{KB} \), and it can then be directly queried, just like problem-solving knowledge. The border between knowledge for problem-solving and knowledge for explanation generation then becomes fuzzy and, moreover, it is not essential to try to distinguish between them. Generating more conceptual explanations becomes itself a type of problem to be solved, and quite naturally it requires dedicated knowledge.

Explanation knowledge is usually meta-knowledge, i.e. knowledge about knowledge. Therefore, entities that encode meta-knowledge must be allowed to refer to other entities of the same nature that encode knowledge. This calls for a formalism where all knowledge-encoding entities are treated alike.

We are developing a logic-based object-oriented formalism which allows uniform encoding of knowledge and meta-knowledge and the linking of entities from several conceptual levels in the \( \text{KB} \). The formalism is based on a powerful first-order logical theory in which individual objects, classes of objects, attributes of objects, and associations among objects are in turn all treated as objects, i.e. as elements of a ‘global universe of discourse’. This means that the normal structuring principles of the object paradigm (see box) are integrated into logic. This homogeneity of the object theory may be compared with that of set theory, where functions and relations are ordinary sets.

The formalism permits a variety of conceptual explanations to be generated, provided, of course, that the corresponding knowledge is present in the \( \text{KB} \). Because of the uniform representation of objects, all items of knowledge can be represented in the \( \text{KB} \) and queried directly through the \( \text{PS} \) interface.

The justification of a solution produced by the \( \text{PS} \) is provided as the proof tree obtained from another query to the \( \text{PS} \) requesting verification that the solution complies with the fundamental principles of the domain.

For example, if explanations are requested about a concept of the domain (normally represented as a class in the object formalism (see fig. 7), information is extracted from the network of objects, starting from the representation of the concept requested, exploring its attributes (the immediate properties of the concept), its associations with other concepts, its immediate superclasses (the closest generalizations of the concept), their attributes and their associations, and so on until a suitable level of detail is obtained. This exploration is conducted by appropriate queries to the \( \text{KB} \) through the \( \text{PS} \) interface.
Object-oriented approach to knowledge representation

Object orientation is more an approach than a specific set of constructs.
As an informal definition, an object-oriented formalism allows the representation of every important entity in an application domain as one object in the formalism.

A class is an abstraction of a collection of objects with similar properties.

In an object-oriented approach, a KB is structured around a hierarchy of classes. A class can be connected with classes of lower rank (subclasses) and with classes of higher rank (superclasses).
A class inherits properties (or attributes) of its superclasses while defining some properties of its own.

Objects are instances of at least one class and of all its superclasses.
General structuring mechanisms provide a powerful modelling methodology for:
• the relationships between an object and its class and superclasses (instantiation-classification)
• the relationships between a class and its superclasses in the hierarchy (specialization-generalization)
• the representation of associations among objects as other objects (decomposition-aggregation).

In another example, if a user wants to know the differences between two similar concepts, their closest common superclass will be inspected (the closest concept that is a common generalization), their common attributes (if any), their specific attributes, and so on.

Fig. 7. Two classes with one common superclass and some of their attributes. In the financial domain, savings accounts and fixed-term deposits are special cases of deposits. Deposits are represented as a superclass of the classes of savings accounts and fixed-term deposits. The attributes of savings accounts, for example, comprise the attributes specifically associated with the class (notice required, etc.) plus those of the superclass (debtor etc.).


Summary. In this article we have argued that a wide spectrum of explanations can be useful in an expert system. We have looked briefly at attempts to generate some of these explanations. A general architecture has been proposed for expert systems, clearly distinguishing between the functions of two cooperating agents, a dialogue manager and a problem-solver. We have argued that a logical framework is most convenient for a precise declarative version of the knowledge base and of its relationships to query answers from the problem-solver. Our own work is aimed at the development of a powerful problem-solver component along those lines. Formalisms and tools have been developed for generating a form of proof tree that offers a sound basis for the conventional trace-based how-, why-, why-not-explanations. A logic-based object-oriented knowledge-representation formalism is being designed to support a uniform encoding of knowledge and meta-knowledge for the generation of more conceptual explanations.