1988

A knowledge acquisition assistant for the expert system shell Nexpert-Object

Florence Perot

Rochester Institute of Technology
School of Computer Science and Technology

A KNOWLEDGE ACQUISITION ASSISTANT FOR THE EXPERT SYSTEM SHELL NEXPERT-OBJECT™

By

FLORENCE C. PEROT

A thesis, submitted to
The Faculty of the School of Computer Science and Technology
In partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

8/19/88
Pr. John A. Biles (ICSG Department, R.I.T.) Date

Aug 19 1988
Dr. Alain T. Rappaport (President, Neuron Data) Date

19 Aug 1988
Dr. Peter G. Anderson (ICSG Chairman, R.I.T.) Date

ROCHESTER, NEW-YORK
August, 1988

ABSTRACT

This study addresses the problems of knowledge acquisition in expert system development examines programs whose goal is to solve part of these problems. Among them are knowledge acquisition tools, which provide the knowledge engineer with a set of Artificial Intelligence primitives, knowledge acquisition aids, which offer to the knowledge engineer a guidance in knowledge elicitation, and finally, automated systems, which try to replace the human interviewer with a machine interface. We propose an alternative technique to these approaches: an interactive syntactic analyzer of an emerging knowledge base written with the expert system shell called Nexpert *Object*™. This program intends to help the knowledge engineer during the editing of a knowledge base, both from a knowledge engineering and a knowledge representation point of view. The implementation is a Desk Accessory written in C, running on MacIntosh "concurrently" with Nexpert *Object*™.

Computing Reviews Subject Codes

Primary:	Learning	I 2.4
	Knowledge Representation Formalisms and Methods	I 2.5
Secondary:	Programming Language and Software	I 2.6

Keywords: Knowledge acquisition, Knowledge engineering, Knowledge representation, Expert system shells.

ACKNOWLEDGMENTS

I would like to thank Neuron Data for the loan of Nexpert, Guy Johnson for his help to find the necessary hardware, and Bernie Huber for his helpful advice, and also all the other persons who had helped me to do this project:

Alain Rappaport from Neuron Data,

Al Biles from the Graduate Computer Science Department,

Laurent Delamare and Albert Gouyet from Neuron Data,

Daryl Johnson from the Applied Computer Studies,

and

Shirley Bower from the Wallace Memorial Library.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

Most of the first expert systems, developed in the 1970's, required at least two years of development, tests, and refinement in order to provide a nearly achieved system that could be used in real applications [Blan 87]. A specialist, the knowledge engineer, was needed for that purpose. That person interviewed the expert(s) and compiled his/her (their) knowledge in an expert system shell, but he represented (and still represents) a bottleneck because of his scarcity and the specific skills needed for such tasks. Also, another issue occured even after a successful expert system implementation: maintenance. This problem implied a continuing modification of the knowledge base.

To face these new problems, a new generation of expert systems has entered the market place: those built with expert systems shells, which include features that allow a non-AI specialist (but nevertheless a domain expert) to enter his knowledge without (or nearly without) the help of an AI specialist. These products emphasize a user-friendly interface, using a window and graphics environment, and also have integration facilities with other existing programs; however, they usually offer only a simple knowledge representation such as basic rules.

Since real-world problems require more complex knowledge design and reasoning, more sophisticated expert system shells with powerful concepts (frames, objects, different levels of reasoning), have been developed. However, this has increased the gap between the ease of use and the complexity of knowledge representation; this gap is emphasized by the fact that experts are

deeply involved in domain problem solving and far much less concerned by AI techniques.

This has led to a new direction of the research in expert systems and the proposal of different solutions to the knowledge acquisition bottleneck. We will present in Part One a survey of what has been done so far in this research area, without intending to be exhaustive. This overview will mainly provide a perspective on the place of the knowledge assistant I propose to build among what has been developed. This will be followed in Part Two by a description of a specific knowledge acquisition assistant for an expert system shell called Nexpert *Object*[1].

Chapter 2 gives an overview of knowledge acquisition tools along the dimensions these programs address, that is, the dimension of knowledge engineering tasks.
Chapter 3 presents a set of knowledge acquisition aids representative of the actual research trend. The same dimensions used in Chapter 2 are used to guide this quick survey.
Chapter 4 intends to provide the basic concepts of knowledge representation used in Nexpert *Object*, and outlines some knowledge engineering issues as well as some knowledge representation issues encountered by Nexpert users.
Chapters 5 and 6 discuss the details of the implementation of the knowledge acquisition assistant for Nexpert *Object*.

---

[1] Nexpert-Object is a trademark of Neuron Data Inc.

Finally, Chapter 7 proposes the overall conclusions about this knowledge acquisition assistant and more generally about knowledge acquisition issues.

The reader can find in the appendices a complement of information about Nexpert *Object*, a detailed analysis of the Cornell application (the program test), as well as the structure of the program and a glossary.

PART ONE

BACKGROUND

(Solutions to the knowledge acquisition bottleneck in expert system development)

The process of acquiring knowledge from a domain expert is one of the most time consuming and complex tasks in expert system development. Not only the formalization of the domain knowledge, but also its translation into a suitable form for processing by expert system software, is a particularly acute concern during each stage of a design project. Several approaches have been taken to solve these problems: knowledge acquisition tools, knowledge acquisition aids, and automated knowledge acquisition systems. By knowledge acquisition tools, we mean software support for application of knowledge acquisition techniques, but without any guidance of their own. This is not the case with knowledge acquisition aids which provide process guidance of their own. Finally, the last approach is designing systems that automate part or all the knowledge acquisition process [Grub 87].

We will review these approaches along the dimension they address, that is, the dimension of knowledge engineering tasks. We can distinguish the following process stages: elicitation of a basic framework at the early stages of the expert system development; knowledge structuring and encoding; and finally, knowledge base refinement. We will not fully develop the automatic acquisition techniques, even though they are certainly interesting, since this thesis is concerned with knowledge acquisition assistant programs.

CHAPTER 2

KNOWLEDGE ACQUISITION TOOLS

Knowledge acquisition tools are, in fact, assistants to the knowledge engineer, providing real-time aid in collecting information from an expert at the different stages of expert system development. Human information processing demands are relatively high during interviews with experts, and sometimes the knowledge engineer risks being overwhelmed by the wealth of information available, particularly when the content is unfamiliar to him/her [Olso 87]. In this sense, knowledge acquisition tools can be used to lighten the information load. It is also possible, as we will see later in this section, to provide the system with inference capabilities that allow it to examine and analyze the emerging knowledge base.

2.1 Tools for Eliciting a Basic Framework

Early work in knowledge engineering concentrated on developing expert system shells and representations. Can they be considered as complete cognitive models? This is an open question.

Recent emphasis has been on methodologies for structured knowledge acquisition, which especially emphazise building a conceptual or knowledge-level structure of the domain. Here, experts can describe their domain structure in some accessible representation free from the implementation

issues of the representation in the final expert system. These architectures provide problem-solving primitives.

Two different approaches have been taken by the designers of knowledge acquisition tools: they both refer to two classes of methods for revealing what experts know [Olso 87]. The first approach uses direct methods, which ask the expert to report on the general model he/she can articulate: this approach is essentially influence diagram analysis. In contrast, the alternative approach refers to the indirect methods, which do not rely on the expert's abilities to articulate the information that is used; they collect other behaviors, such as recall or scaling, from which the knowledge engineer can infer what the expert must have known in order to respond the way he/she did. This includes multi-dimensional scaling, hierarchical clustering, and repertory grid analysis. In the following section, we will explore some applications using these approaches.

An Application Using Repertory Grids (indirect method)

John Boose from Boeing Computer Services has developed a system called Expertise Transfer System (ETS) using clinical psychotherapeutic interviewing methods. These techniques were originally developed by George Kelly, who was interested in helping people categorize experiences and classify their environment.

First, ETS elicits conclusion items (or elements to follow Kelly's terminology) from the expert. These are traits from which the system will determine, with the expert's help, differences and similarities. If the expert cannot verbalize the set of conclusion items initially, ETS enters an incremental interview mode of operation, using on a program called DYAD,

where elements are elicited one at time, based on differences between and similarities to other elements [Benn 83].

After the expert has listed the items to be considered, ETS asks him to compare successive groups of three elements, and name an important trait and its opposite that distinguishes two members of this triad from the third one. This provides a list of items to be classified, and a list of classification parameters, derived by the expert.

Next, the system asks the expert to rate each element against each construct (i.e. each trait and its opposite), thereby forming a rating grid (refer to Figure 1). Once this grid is established, several analytic methods are invoked to structure the knowledge. A non-parametric factor analysis is performed to cluster similar constructs into constellations. Then, an entailment graph of implication relationships is built using the methodology of ENTAIL. The constellation analysis shows constructs that are nearly functionally equivalent, given the elements chosen by the expert. A more elaborate version named AQUINAS also uses hierarchical clustering [Boos 87].

If the expert disagrees with several of the relationships, he may edit the corresponding ratings until he is satisfied with their values. If this does not correct the problem shown by the entailment graph, ETS asks for any elements that would be exceptions to the entailment. The last solution relies on the refinement of the construct involved in the implication relation by invoking a simple laddering method that asks "why?" and "how?" questions concerning the constructs.

After the entailment graph has been constructed, ETS generates heuristic production rules with a belief strength. All rules might be reviewed and modified by the expert.

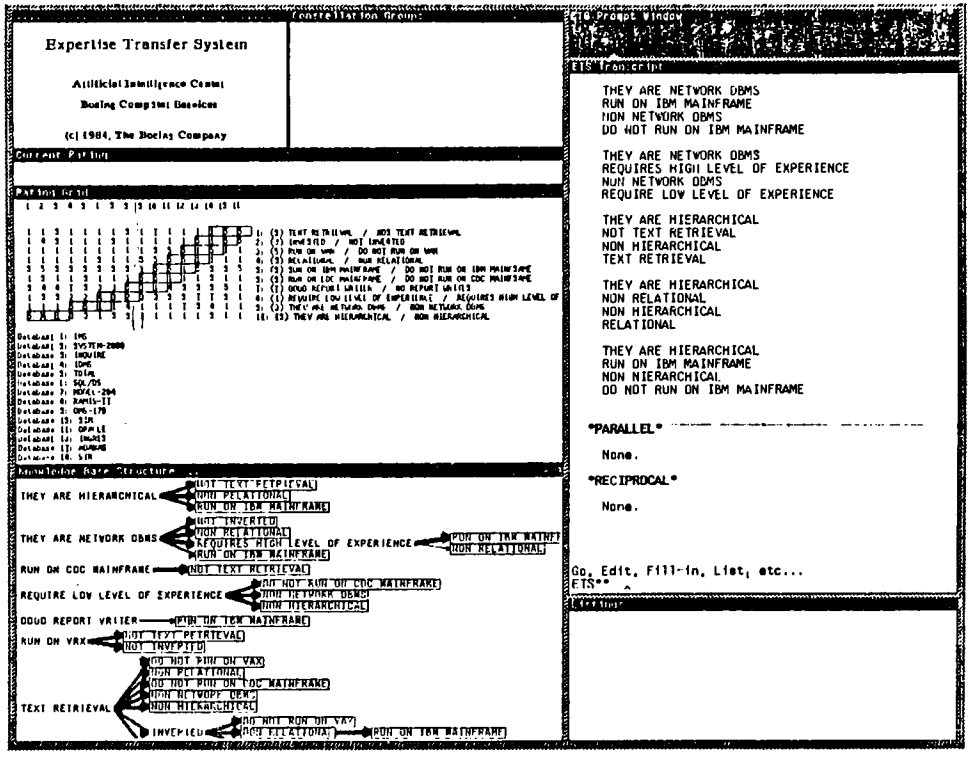


Figure 8. Screen Snapshot of ETS Showing Rating Grid and Entailment Graph.

This system allows the knowledge engineering team to settle on a basic vocabulary, important problem traits, and an implication hierarchy for these traits. They do not need to begin from scratch when beginning discussions with the expert. A question persists: the use of the generated rules in the whole process. Nevertheless, ETS provides a useful front-end processor to elicit initial traits and heuristics for analytic problems; however, it does not offer any guide to the user. The final decision relies on the expert for the validation of the results of the analysis and the sufficiency of elicited constructs.

An Application Using Influence Diagrams (direct method)

INFluence diagram FORmer (INFORM) is essentially a knowledge interface for building knowledge-based systems in the Influence Diagram-based Expert System (IDES). The advantages of influence diagrams rest on their graphical representation of the decision problem structure, but they also maintain the computational utility of the decision tree. They consist of three layers: the relational layer, the functional layer, and the numerical layer (refer to Figure 2). This hierarchy seems to fit well into the way people tend to build models from simple to complex, and from conceptual to numerical. The key ideas used in INFORM's approach are the following: start modeling at the most general level of precision or specificity, and increase specificity only for the best improvements in model performance [Mich 86].

The user (who can be the expert) goes through a problem and session structuring module, and then a succession of editing and analysis phases. At any time, the user may get help about the syntax, options, or intent of the

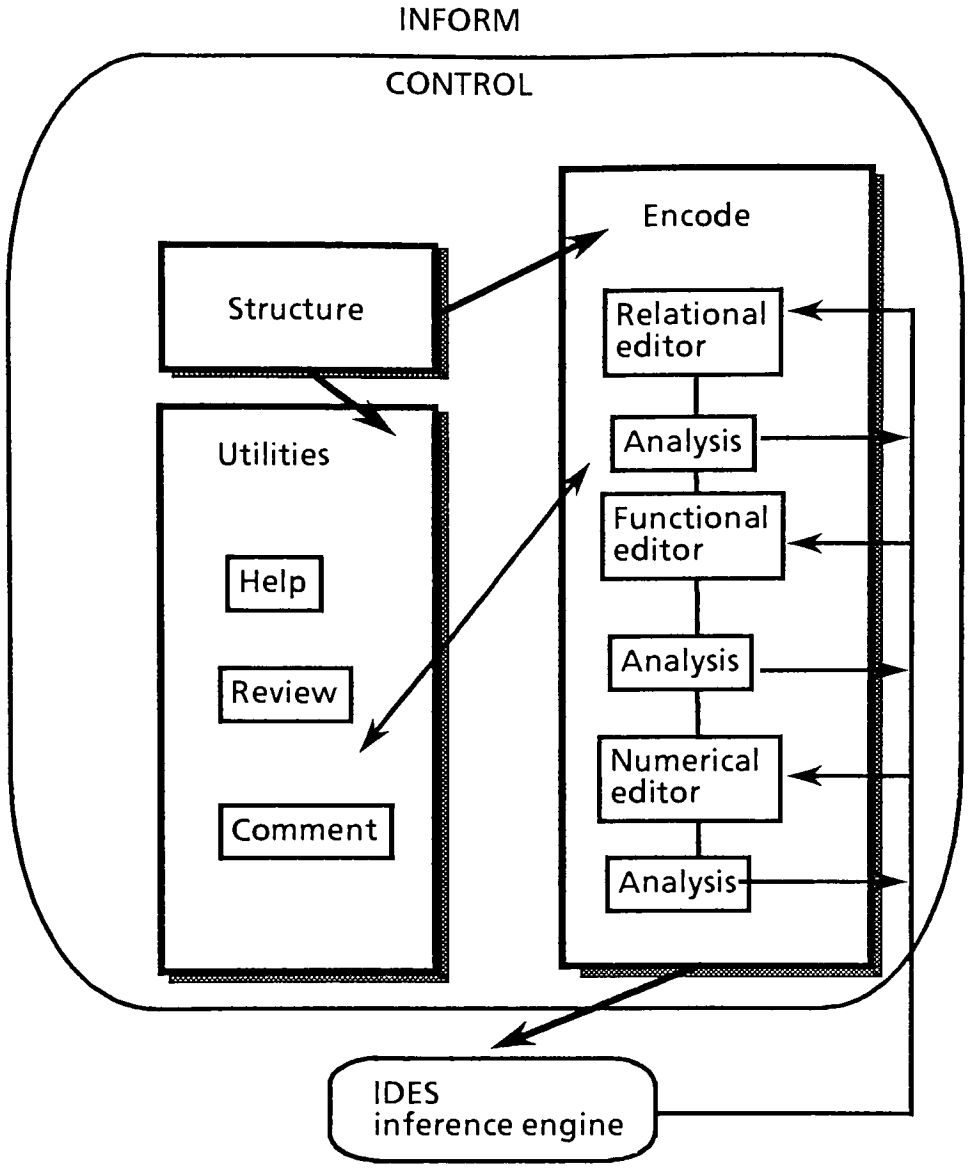


Figure 2. INFORM's architecture

current phase, comments about some aspect of the model or the modeling process, or he may review some graphic or textual aspect of the model.

In the relational editor, the user specifies the combinations of nodes (i.e. possible events or properties for some object) and arcs (i.e. influence of node outcomes on other node outcomes) of the influence diagram (refer to

Figure 3). On exiting the editor, this information is parsed and the system

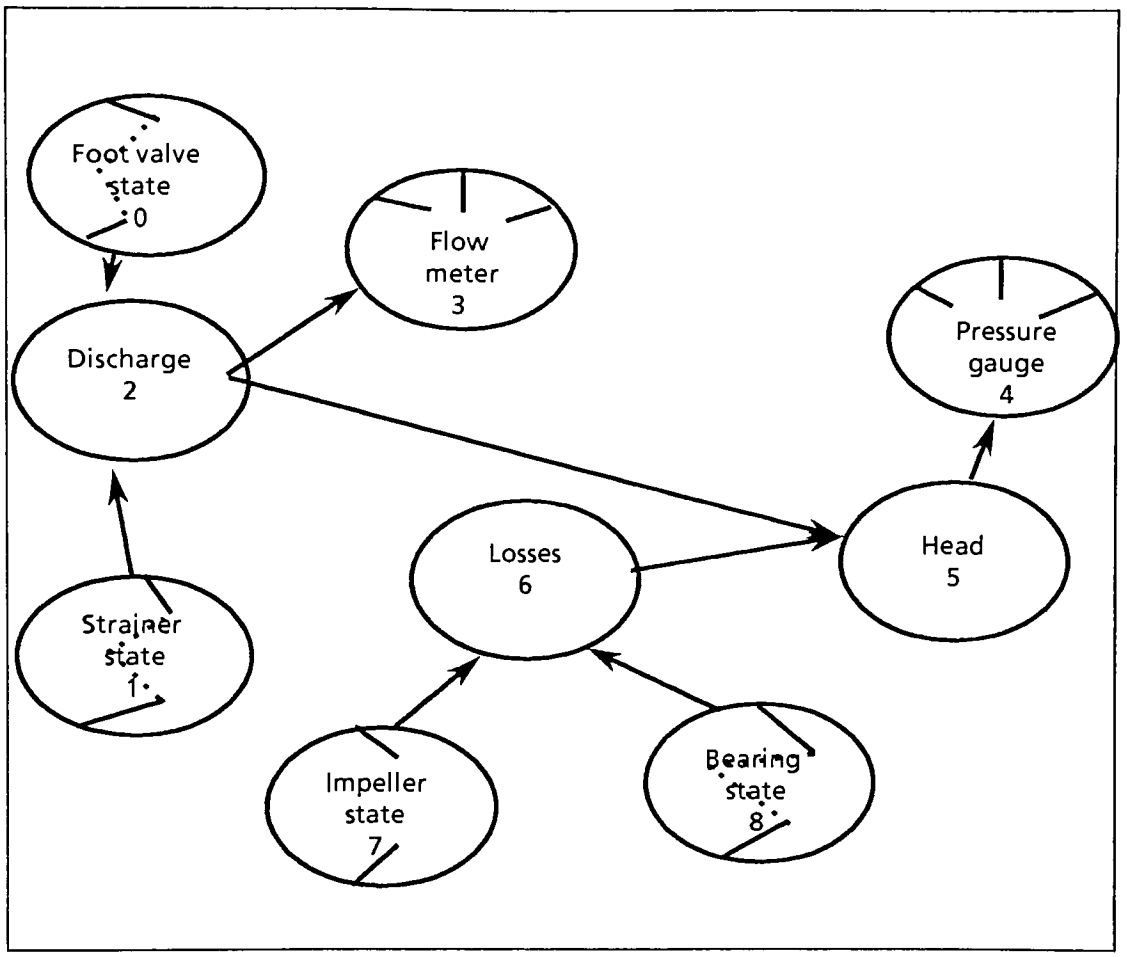


Figure 3. Sample influence diagram: a diagnostican's model for a simple centrifugal pump.
At the relational level, we specify that the pump's discharge (high, low, or nil) is influenced by the foot-valve state (open or closed) and stainer state (clear, partially clogged or clogged). At the relational level, we specify that the arc from foot-valve to discharge is logical (and not probabilistic): if the foot-valve is closed, the discharge is certainly nil.

assures the completeness of each node. It also offers a simple analysis (cycles and bushyness checking) and advice about what to do next. In the functional editor, the user is prompted for the description of functional form, states, and the type of IDES rules (decision rules, quantitative form, or uncertainty representation). Here again, the system parses all the information for

completeness and offers an estimation of modeling effort as well as advice about the number of states in the nodes, for example. Finally in the numerical editor, the system assesses agregate uncertainty information; it also offers an analysis of sensibility and performance.

INFORM thus offers a specific knowledge representation, the influence diagram, and also offers a way for the knowledge engineer to define a context and a model for problems whose solution is strongly driven by model structure. In this sense, it corresponds to a set of tools for initiating a model.

2.2 Tools for Structuring and Encoding Knowledge

Gathering knowledge from an expert is not the only difficult task assigned to the knowledge engineer. He also has to translate his knowledge into a form that can be executed by a computer program. This task is particularly acute because expert knowledge is expressed in a different form from that required by the machine, even with the development of new knowledge formalisms, this is known as the representational mismatch. In this sense, building good general "bridges" between expert and implementations and making these bridges easy to "traverse" become key issues in reducing the knowledge acquisition bottleneck [Grub 87].

These principles have led to the design of tools for structuring and encoding knowledge at the creation or at the updating of a knowledge base. Two approaches have been followed. The first approach consists in providing the user (expert or knowledge engineer) with a knowledge interface or knowledge editors; this includes form-filling, graphic and natural language. The alternative approach is to provide the user with problem-solving primitives specific to a task (task-specific architectures) or general problem-

solving primitives (general-purpose inference engines). We will give some examples in the rest of this section to illustrate this classification.

Knowledge Interface and Knowledge Editors

The MU (Manages Uncertainty) architecture presents interesting features in the view we have defined previously. It is divided into three parts: a knowledge acquisition interface that hides the technicalities of encoding knowledge; a virtual machine providing task-specific reasoning mechanisms; and an AI tool box consisting of the expert system shell KEE. The knowledge acquisition interface is a set of tools that together present a "user illusion" of a language of application-specific instantiations of constructs provided by the architecture. MU's primary function is to make it easy for experts to formulate their expertise in the available language, one restricted to task-level terms [Cohe 87].

MU is a generalization of the knowledge system MUM, which Manages Uncertainty in Medicine, currently in the domains of chest and abdominal pain. Its architecture is shown in the following table:

Tool	Objects in user's view	Software support
Knowledge acquisition interface	*Application-specific terms* Diseases, tests, treatments, questions, intermediate diagnoses, criticality of diseases, cost of tests, efficacity of treatment	*(Meta-)knowledge base utilities* Language-specific editors, form-filling interfaces, inferential analyzer, graphical display for objects and relations
Virtual machine (shell)	*Task-level constructs* Clusters and combining functions, control parameters, control rules, preference rankings	*Task-specific reasoning mechanisms* Value propagation functions, interface to the inference net,rule-based planner, decision-making support
AI tool box (KEE™)	I*mplementation primitives* Rules, pattern matching language, frames and slots Lisp objects and functions Windows and graphic objects	*AI programming techniques* Rule interpreter, knowledge base bookkeeping, inheritance mechanisms Assumption maintenance Demon and message passing support, window system

Table 1. A hierarchy of knowledge engineering tools to support the MU architecture

A better example is OPAL (Oncology Plan Acquisition Language), the knowledge acquisition interface for the ONCOCIN expert system, which uses form-filling to acquire the majority of the expert knowledge used in specifying treatment plans for cancer therapy. The authors call this technique a "knowledge editor" [Mori 87]. It is based on a conceptual model (the structural model of the domain itself). It is this domain model for cancer therapy that gives the program the ability to solicit and display a model for cancer treatment plans, using graphical "form" and other visual

representations that correspond to the way expert physicians seem to think about oncology treatments. Knowledge specified graphically using OPAL is first stored internally in an intermediate representation and then automatically converted to a format used by ONCOCIN to provide clinical consultations [Mori 87].

The entity-relationship model is implicit in the OPAL interface. For example, the form in Figure 4 allows the user to specify the names of the individual drugs that make up a particular chemotherapy. The program presumes that chemotherapies and drugs are entities in the domain and that they are related compositionally. When the user interacts with OPAL, the names of the drugs are entered into the blanks of the graphical form by selecting appropriate choices from a predefined menu or by typing in from the keyboard. Whenever a blank requesting a drug name is filled in, knowledge stored in the definition of the graphical form causes OPAL to create a new drug object in the intermediate representation and link it to the object for the related chemotherapy. Thus, the user's path through the different forms determines the particular entities to which new knowledge is related at each step.

Another feature of the knowledge editor is the graphical environment. In OPAL, a graphical environment permits the user to create special icons that represent the various procedural elements of oncology treatment plans. By positioning these icons appropriately on the computer display and drawing connections between them, physicians can create diagrams that mimic the flowcharts typically found in printed protocol descriptions (refer to Figure 5). Concepts associated with traditional programming languages such as

Specification of Dose information

Chemotherapy: VAM

Subcycle:

Drug: Methotrexate

Drug module: Normal

Dose	Route	Dose Interval and/or Number of Doses	Starting on which days of (sub)cycles?
30 MG/M2	IVPUSH	1 dose	1

Round each dose to Nearest	Maximum Single dose	Maximum cumulative dosage	Acceptable dose modification range
5 MG			

Figure 4. Attribute of the drug entity in OPAL.
The blanks allow physicians to enter knowledge about the administration of a drug in the context of a particular chemotherapy.

sequential control, conditionality, iteration, exception handling and concurrency all can be specified using a graphical syntax.

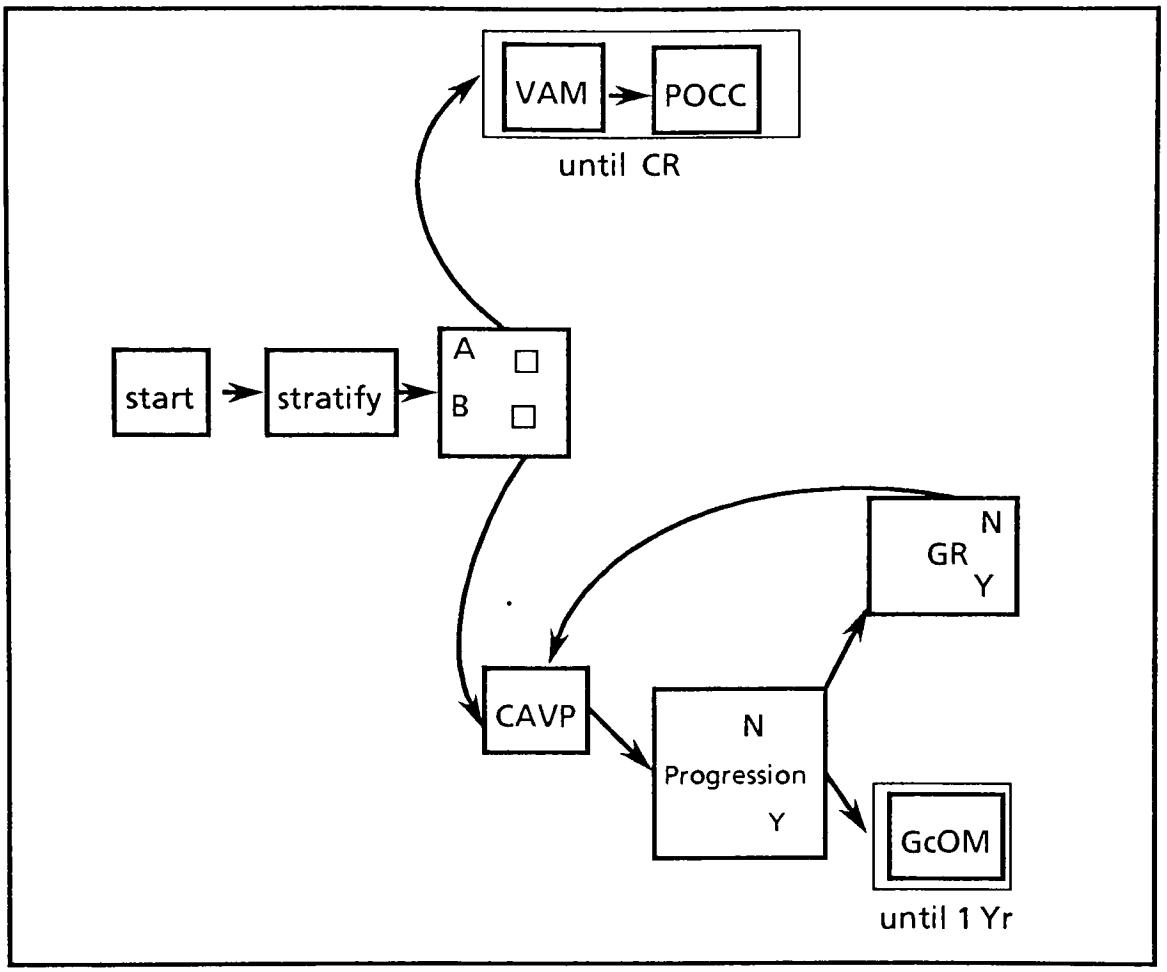


Figure 5. Prodecure representation in OPAL

A final feature of a knowledge acquisition interface can be a natural language processor like INKA, the *INglish* Knowledge Acquisition Interface. INKA is a natural language interface to facilitate knowledge acquisition during expert system development for electronic instrument trouble-shooting. Its only purpose is to elicit knowledge [Pazz 86].

The user (usually a technician) enters a sentence using a subset of English, called "Generalized Language for Instrument Behavior" (GLIB). INKA translates this input sentence into the internal form of diagnostic rules. The translation of the user sentence uses functional schemata, attached to the

phrase structure rules of GLIB and proceeds by instantiating the meta-variables of the schemata of the parse tree, created by INterface enGLISH (INGLISH); this is done to produce a functional structure. Finally, from this structure the final rule form, a Prolog clause, is built up.

Problem-Solving Primitives

Most of the second generation expert systems possess problem-solving strategies such as meta-rules, but these meta-rules cannot qualify as problem-solving primitives, which are based on a higher level of problem conceptualization: the knowledge-level analysis. This leads mainly to task-specific architectures that usually are domain-dependent, such as report evaluation systems or problem-type-dependent using decision-tree exploration [Moze 87].

The design evaluation problem offers an example of a task-oriented problem-solving method. This task can be subdivided into two phases: gathering information about designs and evaluating designs. Most design projects require a finished report containing information about general design objectives, tradeoff decisions, analysis and results, as well as detailed parts specifications. Based on this report, the design evaluation can be made. Consequently, a possible problem-solving primitive for this kind of problem could be report building.

This has been implemented in KNACK (Knowledge Acquisition) in the specific context of electro-mechanical system designs. KNACK is a knowledge acquisition tool that elicits knowledge from experts about a skeletal report and report fragments, and then customizes the report fragments. KNACK thus provides support to the expert in defining this

knowledge. The expert uses keywords to indicate chapter, section and subsection headings. From this the system determines the skeletal report. The expert then can type the text for a report fragment under two forms: texts or blank-filling. This report is then used by the expert-system to evaluate the design project. The main advantages of this primitive lie in the report, which is the expert's knowledge representation [Kitt 87]. A disadvantage of this application comes from its lack of generality.

A more general-purpose approach is the one used in KATE™. Kate is an expert system building tool that is based on an inductive generator of decision trees and on a frame package. The key idea of such a system is based on the tendency of experts to explain concepts by giving examples (refer to Figure 6). From these example sets, Kate automatically generates a diagnostic expert system [Mich 83]. The system can even handle noise in data and deal with a large number of training examples. General domain knowledge can be entered under the object formalism to orient the generalization. After this learning phase, Kate is usable in expert system mode to process new cases. It is able to transform the acquired knowledge into non-nested production rules or objects. Kate can thus be used as a self contained product or as a knowledge acquisition front end for another expert system shell [Inte 88].

2.3 Tools for refining a knowledge base

With the development of expert systems in industry, the problem of knowledge base refinement becomes crucial, but as far as expert systems are concerned, only logical rule (or frame)-based checking is done. This consists of

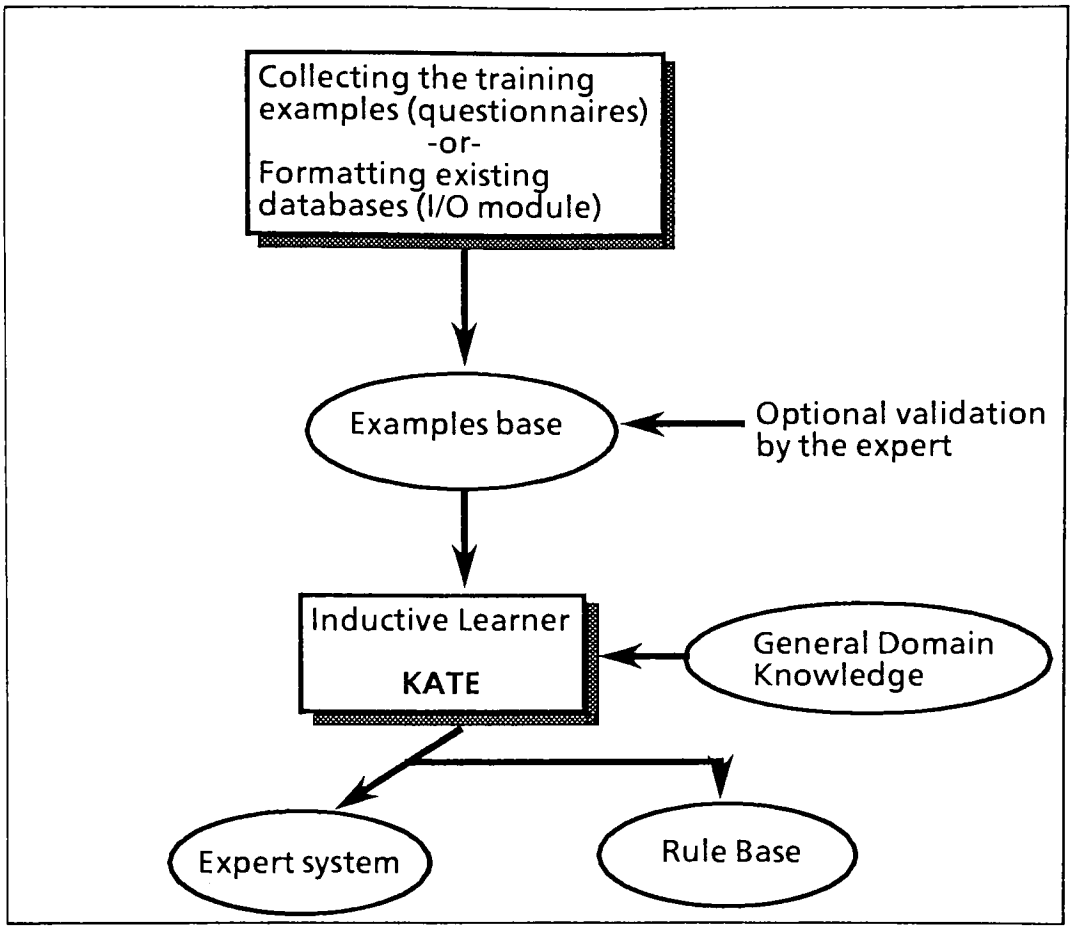


Figure 6. A methodolgy for knowledge acquisition using machine learning (from Intellisoft)

logical checking for consistency (conflicts, redundancy, and subsumption) and logical ckecking for completeness (missing rules).

The methods used in knowledge base refinement depend on the unit of knowledge checking: one rule at a time, one frame at a time, all the rules together, or all the rules and the facts together. In this section we will review some applications involving these methods with more or less powerful checking and using different techniques such as Petri nets and decision-tables.

One-rule-at-a-time Checking

This method refers to the early expert system checking systems. The most famous one, EMYCIN, checks essentially for syntactic errors after each editing of a rule. By this we mean whether terms are spelled correctly or not and whether parameter values are legal. However, EMYCIN also performs a limited semantic check: each new or changed rule is compared with any existing rules that lead to the same parameters to make sure it does not directly contradict or subsume any of them. In either case, the interaction is reported to the expert, who may then examine or edit any of the offending rules [Buch 84].

Frame Checking

The Knowledge Representation Editing and Modeling Environment (KREME) utilizes a frame package for knowledge representation. Its main feature is its incorporated tool, the knowledge integrator, for frame consistency maintenance. The basic idea is the following: the definitions of each new or changed chunk of knowledge is first checked to see how well it fits into the knowledge base, and, second, the other definitions are checked to see how they interact with the definitions of the new chunk of knowledge [Abre 87]. The knowledge integrator is based on an extension of the classification algorithm developed for the NIKL representation language. This frame classifier works in two stages: a completion stage and a classification stage.

During the completion stage, the system refers to the basic inheritance mechanism used by KREME frames. The knowledge integrator installs all inheritated features of a frame in its internal description. Given an object, its set of defined parents and its set of features, the algorithm determines the

full, logically entailed set of features for this object. The completion testing is partitioned by feature type (i.e. slot, disjoint-class, etc...).

The second stage of the frame consistency checking consists of inserting the new frame at the right place and eventually reclassifying the frames. This is done by finding all of the most specific subsumers of the concept being defined or redefined: the algorithm is mainly a special-purpose tree-walking algorithm.

The advantage of such a system is that the user can see immediately the effects of a reclassification via the dynamically updated graph of the subsumption.

Knowledge Base checking

In knowledge base checking the system groups the rules by context in order to evaluate their logical consistency and completeness. Different techniques are used to define these contexts: they can be generated automatically by the system, as in the case of ESC (Expert System Checker) or SACCO, or they can be generated from the information provided by the expert, as in CHECK or ONCOCIN. Also, different knowledge base modelings are involved in checking methods: tables, decision tables or Petri nets. We will review quickly these applications to show the kind of consistency and completeness checking that can be done over a knowledge base.

ONCOCIN, as we have already seen, offers logical consistency and completeness checking using rules' contexts. ONCOCIN requires the expert to designate which parameters are used as context. To check the knowledge base, the system partitions the rules into disjoint sets, each of which concludes about the same parameter in the same rule's context. The resulting sets can

be checked independently. To check a set of rules, the program finds all parameters used in the conditions of these rules. It then makes a table, displaying all possible combinations of condition parameter values and the corresponding values that will lead to the action parameter. Finally, it checks the tables for conflict, redundancy, subsumption, or missing rules through combinatorial enumeration. The tables are then displayed with a summary of any potential errors that were found. It is the expert's responsibility to review, edit and eventually add new rules [Moze 87].

CHECK addresses some additional problems in knowledge base checking by addressing unreachable and dead-end clauses as well as circular chains. It assumes rules are naturally separated by "subject categories" (that is, a group of related rules kept together for documentation). Rules are checked against all others in the same subject category and all others that have the same goal. One advantage of CHECK is that it is independent of the rule-based knowledge systems it checks [Crag 87].

In contrast to these programs, Expert System Checker (ESC) determines context automatically without any explicit context definition and evaluates the knowledge base through decision table building. ESC begins by constructing for the entire knowledge base a master decision table with multiple logic choices; it then automatically partitions the logic into disjoint sets for the purpose of checking completeness and consistency. This partitioning is based on both the context (i.e. logical isolation from other rules) and a hierarchical rule structure within a given context. Ambiguity and redundancy are reported to the expert or the knowledge engineer along with a listing of the logical conditions where such occur. The use of decision table speeds up these processes [Crag 87].

Another solution, involving Petri nets, has been proposed in order to avoid querying the expert for a context definition. The knowledge base is structured by sets of simultaneously reachable rules or "concepts" depending on premises and possible values of parameters as shown in Figure 7, and Petri net analysis software is used to detect inconsistency and unreachable rules.

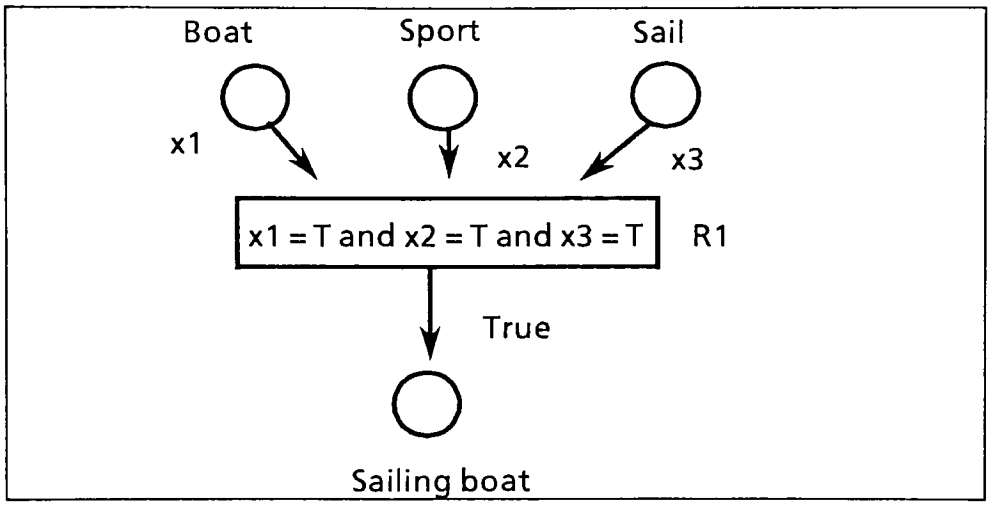


Figure 7. Example of Petri net modeling using the rule R1:
Boat, Sport, Sail -> Sailing Boat

Dynamic Checking

Dynamic checking, in contrast to the static checking we have seen previously, involves both rules and facts in attempts to prove the adequacy of the rules to match the real world the facts are supposed to represent. In this sense dynamic checking qualifies as semantic checking [Quinl 82].

Ayel and Laurent have implemented this view in a program called SACCO using a logic formalism. Their techniques are based on the concept of

completeness with regard to the initial values, which have been obtained by querying the knowledge engineer. The first step of this method consists of determining the maximal consistent fact base. Then, the rules are applied to each fact base determined previously. Each of the resulting fact bases is compared with the corresponding initial fact base; this allows the system to check for consistency with the expert's help. This approach thus permits the detection of discrepancies in the rules and also the completion of constraints provided by the expert.

We have seen how a program can help the knowledge engineer during the different stages of the knowledge acquisition tasks (framework elicitation, knowledge structuring, knowledge encoding, and knowledge base refinement). However, these tools only provide the knowledge engineer with new techniques, and the decision of the time and type of techniques to be used still relies on the knowledge engineer, whatever his/her skills in knowledge engineering.

CHAPTER 3
KNOWLEDGE ACQUISITION AIDS

Several approaches have been taken to attempt to automate aspects of the process of uncovering expert knowledge with the goal of improving the efficiency of knowledge engineering. Thus far, the trend is to replace the human interviewer with a machine interface capable of presenting questions to experts and recording their answers. Sometimes, capacities for analyzing knowledge representations are added, but the lack of generality of such applications and their inability to deal with unusual circumstances have led to an alternative approach: an augmentation of the knowledge engineer's skills by providing real-time analysis of the knowledge collected from an expert. Such programs may also offer guidance for the interviewer's responses and actions.

In the previous chapter, we have seen how knowledge acquisition tasks provide a basis for describing and classifying knowledge acquisition tools. The same dimensions can be used to provide an overview of the different methods employed in analyzing knowledge: knowledge eliciting assistants, knowledge structuring and encoding assistants and knowledge refinement assistants. Also, a new kind of program, the workbench system, is emerging. These systems offer a panel of tools and also offer guidance in the choice of the right tool. In this section, we will introduce some applications covering the main concepts of these knowledge acquisition aids.

3.1 Aids for eliciting a basic framework

During the initial phases of expert system construction, experts often have difficulties articulating the most appropriate aspects of their problem-solving behavior. This complicates the already complex task of the knowledge engineer by introducing two key issues: organizating the expert's interview and understanding the problem's knowledge level. Based on these issues, we can regroup into two classes the applications considered as assistants for eliciting a basic framework: applications that are attempts to organize a knowledge acquisition interview on the basis of a particular problem-solving task, and applications that are attempts to conduct a knowledge acquisition interview on the basis of psychological methods. We will review successively an example from each group.

Knowledge Eliciting Based on Problem-Solving Primitives

ROGET is a knowledge-based system that assists a domain expert in the task of defining an expert system's skeletal design (problem-solving task and abstract categories or goals) [Benn 83].

ROGET starts by interacting with the expert to define the problem-solving task. It requires the expert to characterize briefly what sort of problem-solving task the expert system will perform. ROGET can also provide the expert with examples of previous expert systems. Whatever method is chosen by the expert, ROGET will attempt to classify the problem-solving task in terms of one or more tasks it can recognize. In fact, ROGET contains an enumeration of several diagnostic problem-solving tasks that an expert system might perform: determine-problems, determine-causes, recommend-actions, determine-additional-tests, predict-observations, and

evaluate-evidence. This enumeration resulted from a survey of the current expert system literature.

After the task has been identified, ROGET enters the phase of conceptual structure acquisition. Knowing the problem-solving task, ROGET suggests, interactively with the expert, a set of standard advice categories (such as determine-problems, or determine-causes) to include in the conceptual structure of the new expert system. The system is then able to construct a skeletal configuration of advice category instances, each instance representing a particular goal in the suggestion set. Each domain-specific instance organizes groups of pertinent facts that capture the definitions and distinguishing characteristics of that category.

Finally, the system pursues the dialogue with the expert to elaborate additional types of evidence and inference that might support the subtasks identified in the initial structure (see Figure 8 for an example). Examples of such supporting categories are directly-observed-sign, reported-symptoms, predisposing-factor, laboratory-tests, and important-events. Again, for each of these new categories, additional information is requested of the user to complete the conceptual structure.

Categories in ROGET provide an organization for managing the acquisition dialogue, while the examples are its complement and constitute ROGET's primary method of assisting the expert.

Knowledge Eliciting Based on Interviews

RuleCons is an interactive aid designed to assist knowledge engineers with the initial stages of expert system building. This system incorporates the structure of repertory grid interviewing (as did ETS; see section 2.1) into a

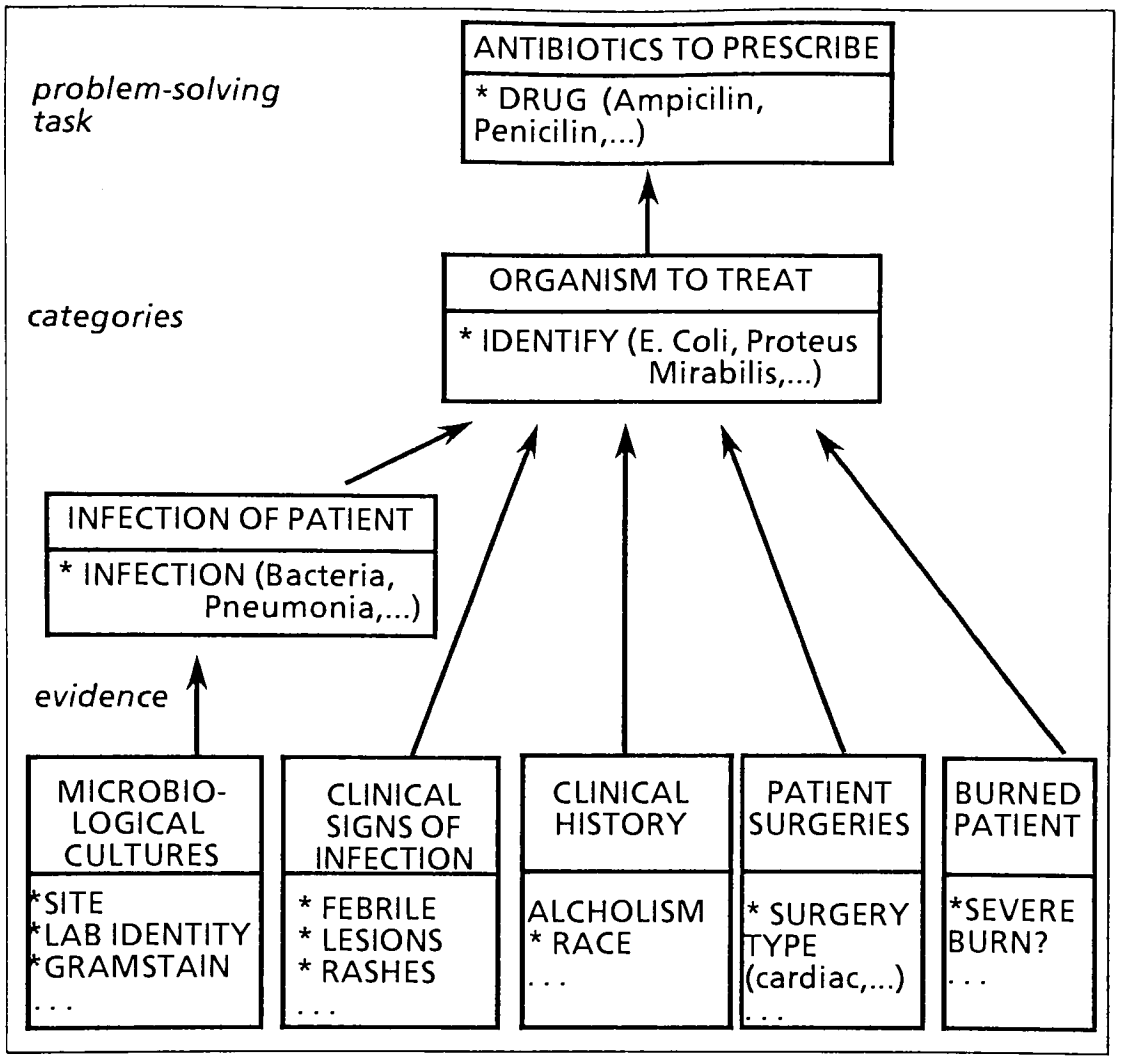


Figure 8. The Conceptual Structure of MYCIN7S suggested by ROGET

system capable of inferring processes. The main feature is its inference engine, driven by a knowledge base of interviewing rules that analyzes the obtained knowledge at each step of the interview. The knowledge base provides rules that guide the initial stages of case elicitation and act to gather additional information in a manner that maximizes the usefulness of the knowledge collected [Muse 87].

RuleCons consists of five basic modules: a repertory grid module, a meta-knowledge base of interviewing rules, an optional historical knowledge

base experts opinions in the problem domain, an inference engine, and finally a production rule constructor (see Figure 9).

The knowledge engineer interacts with RuleCons while conducting the knowledge acquisition interview. At each step of the interview, the program presents a prioritized list of optional interviewer actions in the Prompt Window (see Figure 9). This advice is generated from the meta-knowledge base rules, which may utilize information from the grid administration. A variety of interview action recommendations may be triggered, including elicitation of a new case meeting particular specifications, introduction of a case from the historical knowledge base containing grids previously administered on the same or related topics, elicitation of a new descriptor meeting particular specification, introduction of a descriptor from the historical knowledge base, administration of the implication grid, and examination of preliminary rules.

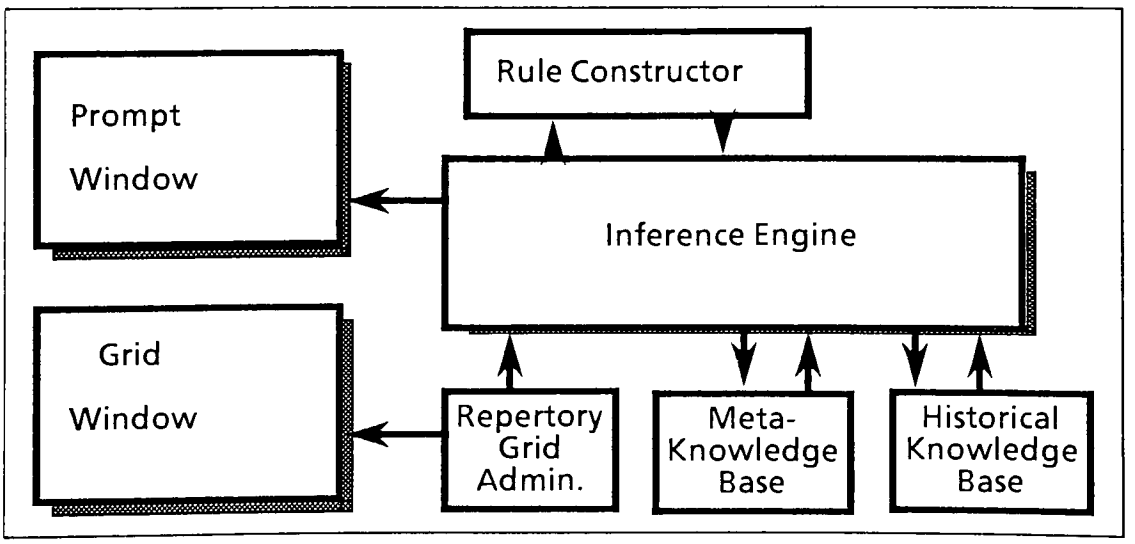


Figure 9 Architecture of RuleCons

The knowledge engineer enters verbal responses and ratings from the expert as they occur. Each input triggers an update of a graphic display of the grid matrix in the Grid Window and initiates a scan of the new data by the inference engine; this can lead to a modification of the list of recommendations made by RuleCons. The knowledge engineer is, however, free to accept or reject any recommended action.

With such an assistant, the knowledge engineer certainly gains refined skills in conducting an interview, but the main focus of this system is a constructive interaction with domain experts in contrast to the previous system, which emphasizes the understanding of the domain knowledge-level.

3.2 Aids for structuring and encoding knowledge

During the difficult activity of modeling knowledge from the human expert, the knowledge engineer needs some guidance that is seldom available. With this issue in mind, more complete aids have been designed to structure and encode expert systems. They present a major common feature, a task-specific architecture addressing only one kind of problem-solving method. We will show what kind of aid a knowledge engineer can obtain from such systems in two particular cases: in the case of a constructive or synthesis problem with the application SALT, and in the case of a deductive or analysis problem with the application MORE (see Figure 10).

An Assistant for Structuring and Encoding Knowledge About a Constructive Problem

SALT is a knowledge acquisition aid for generating expert systems that use a propose-and-revise problem-solving method. This commitment lends a considerable power to its guidance. Its method is based on an analysis of the

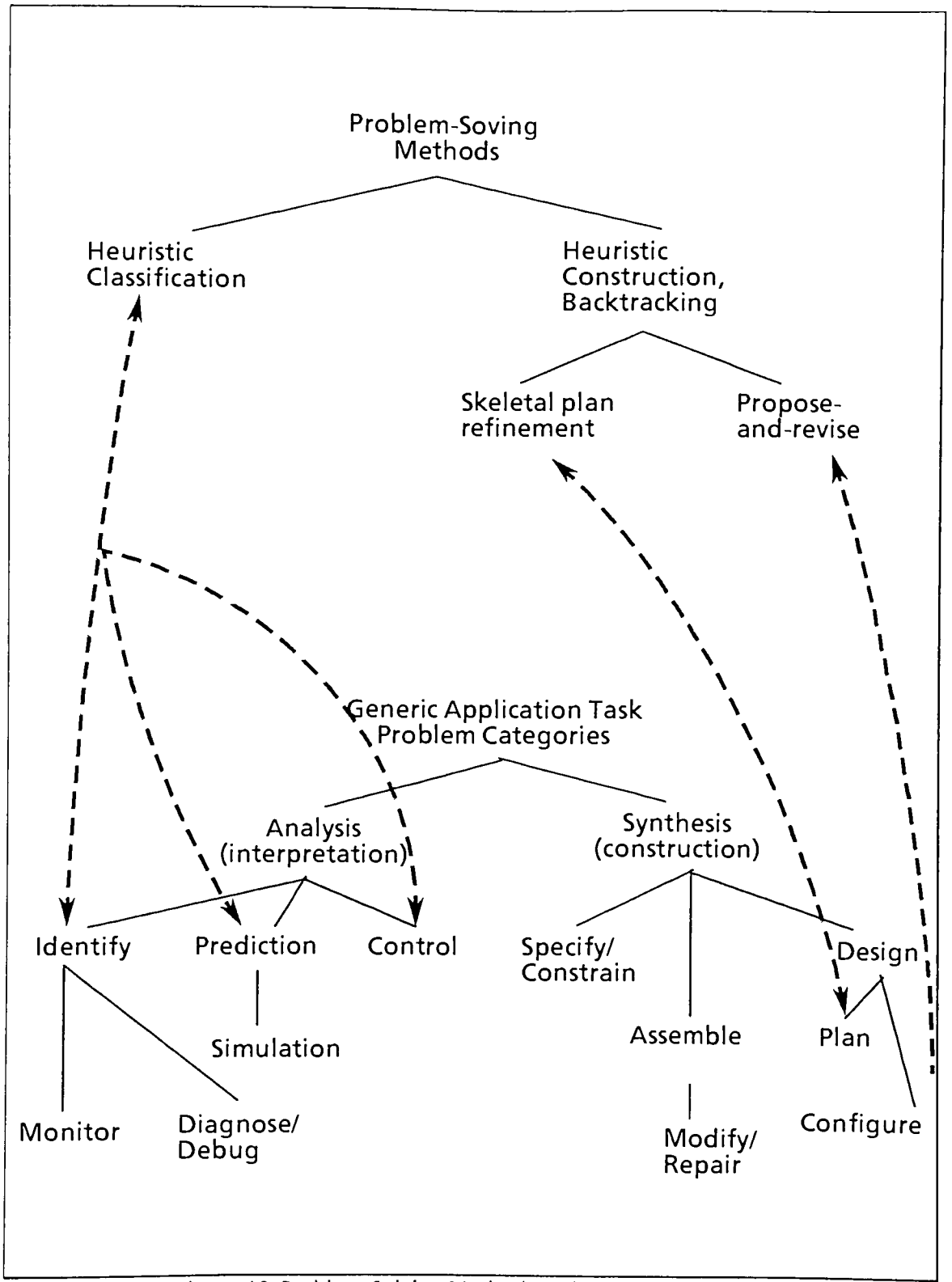


Figure 10. Problem-Solving Methods and Application Tasks

entered knowledge according to its understanding of the constructive problem-solving strategy [Marc 87].

At the start of an interview with an expert, SALT proposes a menu for indicating the type of knowledge to be entered or viewed: procedures for proposing constraints on individual pieces of the design, identification of constraints on individual pieces of the design, and suggestions for ways of revising the design if the constraints are not met. In the first two cases, a set of prompts is then presented to the expert to elicit necessary information to define a procedure or a constraint.

Once the required information has been entered, SALT stores this knowledge within a dependency network and tries to construct an expert system that applies procedures in the least commitment order. If any impossibility of applying three of the steps in a forward chain, the expert is notified. The system also aids the expert in modifying the design using the propose-and-revise strategy; for example, the expert is aided in breaking down loops. In case of constraint violation, the system aids the expert in determining what parts of the proposed design to revise using a "knowledge base backtracking" method. In other words, SALT uses its domain expertise in evaluating disruption and costs to decide what values to change in order to remedy a constraint violation.

An Assistant for Structuring and Encoding Knowledge about an Analysis Problem

Like SALT, MORE relies upon a model-theoretic approach (see Figure 10) to knowledge acquisition issues, but it addresses the specific problems of interviewing domain experts for information of diagnostic significance and providing a shell for building diagnostic systems [Kahn 87].

By its general understanding of the factors that contribute to the evidence of symptoms, the system seeks out information that can give leverage in a diagnostic task. It can query the expert for determining the nature of conditions that could make the occurence of a particular fault more likely, but it also can look for information to build a more sophisticated probe: for example, to determine if a symptom may result from one of several causes by distinguishable underlying causal paths. Such a selection of strategies to structure information is guided by an analysis of the information already acquired and structured in a causal-consequent and prior-posterior model.

MORE maps such domain knowledge into diagnostic rules to which the expert associates confidence factors, but the system supports qualitative expectations on the relative significance of rules, and a warning is displayed when these expectations are violated.

3.3 Aids for refining a knowledge base

We have seen in Section 2.3 some tools for helping the knowledge engineer to refine a knowledge base. Whatever the method used, the knowledge engineer was provided only with substantial checking for logical consistency and completeness and the semantic interpretation of missing or incorrect knowledge was still the responsibility of the knowledge engineer without any assistance from the program. With a knowledge acquisition assistant, a new form of knowledge base refinement has been introduced, involving a semantic check for completeness by own-error analysis. However, such an approach requires at least an understanding of the knowledge-level, or the knowledge representation of the expert system. We will present an example of each of these methods, TEIRESIAS based on rule-model, and MOLE based on a model theory.

An Assistant for Refining Knowledge Base Using Meta-Knowledge

TEIRESIAS is an interactive system for transfer of expertise in a knowledge base, whose basic control structure and representation are assumed to be established and debugged. The fundamental approach to the problem is also supposed to be acceptable [Davi 77].

TEIRESIAS maintains a detailed record of the actions of the performance program during any consultation, over which the expert is queried for his opinion. If the expert disagrees with the performance program's results, he repairs the bugs by tracking down the source of the problem and teaching the system the new rule in a high level dialogue conducted in a restricted subset of natural language. For each result unexpected by the expert, the system asks for correcteness of the rule leading to this result and for correcteness of the premise truth. If any discrepancy is noticed by the expert, TEIRESIAS continues by looking further back in the chain, otherwise the system suggests a missing rule as a diagnostic. When the eventual new rule is typed by the expert, the system checks to see how this new piece of information fits the model of its own knowledge. If it discovers that its expectations have been only partially met, it indicates to the expert what expectations were left unfilled and proposes a "second guessing."

The main idea of TEIRESIAS is based on the focus of attention by context, which allows the system to built up a set of expectations concerning the knowledge it is likely to be acquired from the expert. This model, continually revised as a by-product of the interactivity with the expert, relies on an abstraction description of subsets of rules. These sets, organized into a number of tree structures are built from empirical rule-based generalization and consist of three parts: a list of rule examples, a description of typical

premises and actions, and pointers to models describing more general and more specific subsets of rules.

This model corresponding to what the author calls "meta-knowledge" aids not only in understanding the text typed in by the expert, but also to see how well each interpretation of the new rule fulfills the set of predictions about the likely content of each new rule and the system's models of its knowledge. Another use is to make clear the overall approach of the system to a given topic by simply "reading the rule model to the user." It also gives TEIRESIAS a model of what it does not know.

An Assistant for Refining Knowledge Base Using a Model Theory

MOLE is an knowledge acquisition assistant to build a heuristic problem-solver, but its main feature is its capacity to detect and remedy deficiencies of the initial knowledge base, guided by its understanding of how to diagnose what knowledge the problem-solving method might be missing [Eshe 87].

MOLE proposes to first gather information for constructing the initial knowledge base by asking the expert list the events (i.e. hypotheses and evidence) that are relevant to the domain problem and to draw associations between pairs of events. Four additional pieces of information are also required by the system: the type of event (i.e. whether observed or inferred), the type of evidence (covering knowledge in the case of hypotheses explaining or covering symptoms, and anticipatory evidence in the case of expected characteristics of a justifying hypothesis), the direction of an association, and finally the numeric support value attached to an association that is generated directly by the system. MOLE checks to make sure that hypotheses can be differentiated using anticipatory knowledge or circumstantial knowledge, for

Missing Page

Missing Page

other can be defined by the knowledge engineer (corresponding then to meta-knowledge).

It is then possible to define knowledge about domain knowledge, in order to control the focus of attention of the system. As we have seen in the previous sub-section, the developer can enter knowledge by granuality as the expert gives him pieces of reasoning. But, these knowledge islands are part of a global reasoning and Nexpert offers the possibility to connect them by oriented context links: during a session, the system will explore each knowledge islands and jumps "by idea associations" to another chunk of knowledge following the predefined links. These jumps can eventually be conditional (see Figure 13).

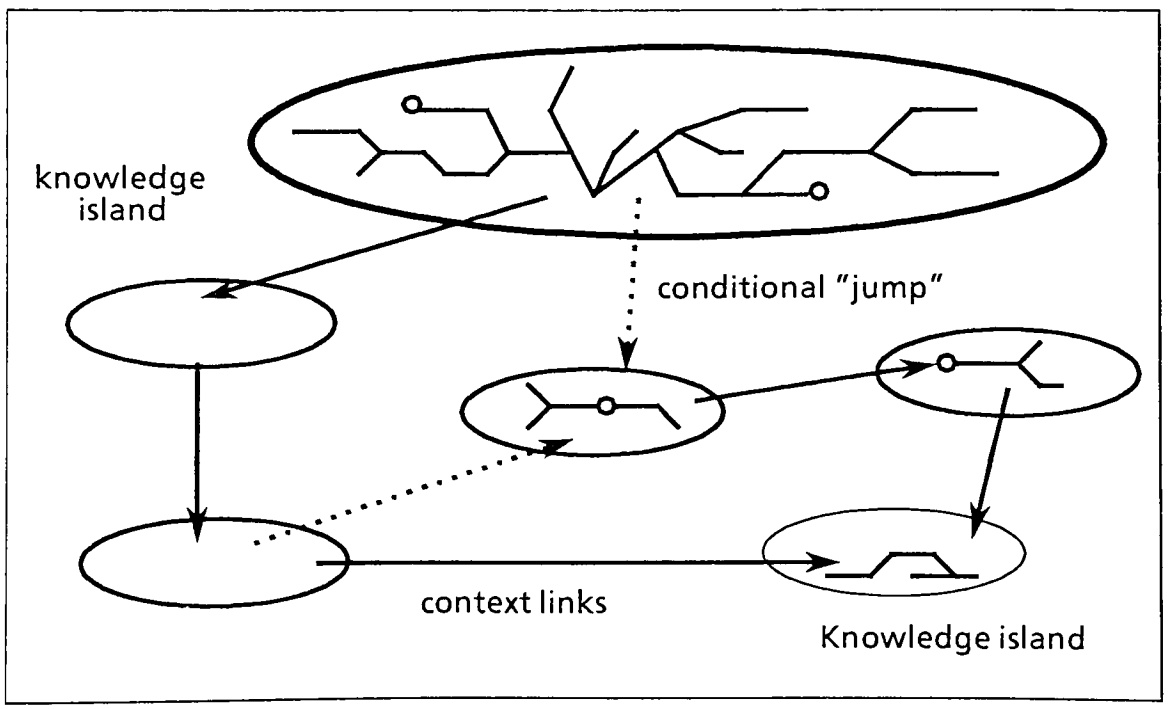


Figure 13. Managing focus of attention in Nexpert *Object*
(from Neuron Data)

Another way to represent an important chunk of knowledge is to develop several small knowledge bases and load the required knowledge base(s) when the given problem requires it (even in the middle of a session).

Nexpert proposes also some local built-in control strategies of the engine: event-driven reasoning can be simulated using the strategy option, propagation of action effects; reasoning revision is possible using a specific operator which resets all used paths leading to a given conclusion; a default reasoning is possible using a Nexpert predefined value NOTKNOWN and its consequences on the truth of a premise containing a datum so instantiated; finally, the system is able to generate automatically new goals by shifting its attention to other rules containing certain true conditions already known.

4.2 Examples of representation matches

Based on industrial applications built with Nexpert, I will introduce some examples of Nexpert knowledge representations. Hopefully, this will clarify Nexpert concepts developed in the previous section.

Model-based Diagnostic Application

An example of a model-based diagnostic problem, consisting of fault diagnostic from frequential spectra received from sensors, has been developed using Nexpert *Object*. The main structure of the reasoning was to pass all the data available through a "mask" to detect a specific fault suggested by Nexpert. The suggestion of a potential fault was based on the knowledge of the studied machine structure. Table 2 shows the corresponding matches between Nexpert concepts and the different elements of the problem [Pero 87].

Nexpert Concepts	Corresponding knowledge-problem meaning
Object Structure	*World model*
Class	generic elements of the machine
Objects	physical components
Properties	physical characteristics
Subobjects	state of the component at a given time
Hierarchical structure	functional/causal structure
Methods/Inheritance	default values
Dynamic objects	studied case(s)
Rule structure	*Dynamics of knowledge*
Rules	Reasoning control
Meta-knowledge	*Control knowledge*
Knowledge base loading	not used
Knowledge island	not used
Inference Strategies	*Control knowledge*
Forward/backward chaining	diagnostic strategy
Link by hypothesis	steps in the reasoning
Automatic goal generation	functional dependency of (mis)-functioning elements
Event-driven	causal dependency of mis-functioning elements
Non-monotonicity	revision of erroneous hypothesis
Pattern-matching	to detect anomalies in data
Procedural programs	*Control knowledge*
databases	purification of data from noisy data
Explanation files	*Explanations*
Show file	justification of a conclusion

Table 2. Nexpert concepts and knowledge chunk in a model -based diagnostic

The object structure was mainly used to represent the different components of the machines as well as their interconnections. However, a special attention should be put on the use of subobjects; they simulate the different states of the machine characteristics through the time period instead of representing sub-components. This conceptual structure allowed the developer to use pattern-matching to identify an anomaly and to correct the fault assumption if necessary.

Planning Application

An example of a constructive problem has been built with Nexpert: the system proposes a daily possible configuration of a plant distributing water to cities (see Table 3).

In fact, Nexpert enters a cycle of generation and test process: it generates solutions for each period of the day. Here, again the object structure was used with a different semantics from the one used in the example of the analysis problems [Folk 87, Pero 87]. They represent abstract entities and parts of the final solution on which the expert system will conclude.

4.3 Issues encountered by Nexpert users

Confronted with the powerful capabilities of Nexpert, users encountered three types of problems: knowledge development coordination problems, expert elicitation problems, and representational mismatches. These issues can be more or less addressed by an assistant program based on a knowledge syntactic analyzer.

Nexpert Concepts	Corresponding problem-knowledge meanings
Object Structure	*World model*
Class	generic solutions
Objects	potential solutions generated
Properties	physical characteristics
Subobjects	different states of retained possible solutions
Hierarchical structure	time dependency of the solutions
Methods/Inheritance	default values
Dynamic objects	new solution(s)
Rule structure	*Dynamics of knowledge*
Rules	cyclic reasoning generation
Meta-knowledge	*Control knowledge*
Knowledge base loading	each of the two steps inside of the reasoning cycle
Knowledge island	early stages of the problem-solving
Inference Strategies	*Control knowledge*
Forward/backward chaining	generate and test reasoning
Link by hypothesis	steps in the reasoning
Automatic goal generation	to jump from one cycle to another
Event-driven	not used
Non-monotonic	to switch to the following period
Pattern-matching	constraint test on generated solutions
Procedural programs	*Control knowledge*
simulation	to generate characteristics of generated solutions
Explanation files	*Explanations*
Show file	presentation of results

Table 3. Nexpert concepts and knowledge chunk in a planning problem

Knowledge Development Coordination

This kind of problem comes from the youth of the expert system technology applied to industrial applications. Manufacturers are used to big software developments and have some troubles to identify the real role of an expert system [Pero 87].

One example of role confusion is mainly due to Nexpert hierarchical object structure, which looks like a data base organization. Nexpert users then tend to use pattern-matchings and object hierarchy to get information about entities, whereas a classical relational data base will be more efficient and appropriate. Nevertheless, the expert system can still have the role of a supervisor, organizing the search and eventually adding an analyzing dimension to the software (see Figure 14). This confusion can be also explained by the presence on the market place of expert system shells integrating their own data base, graphics, spreadsheet package.

This shows an example of bad division of tasks between an open-architecture expert system and more classic programs: computation and graphics must be done by appropriate software communicating or driven by Nexpert's reasoning whenever possible. This separation of tasks has one big advantage: to let the knowledge engineer free to concentrate on the reasoning elicitation.

The other coordination problems encountered with Nexpert *Object* are relevant to the definition of expert system targets: it happens that the expert system goals are poorly-defined. Consequently, the project needs to be

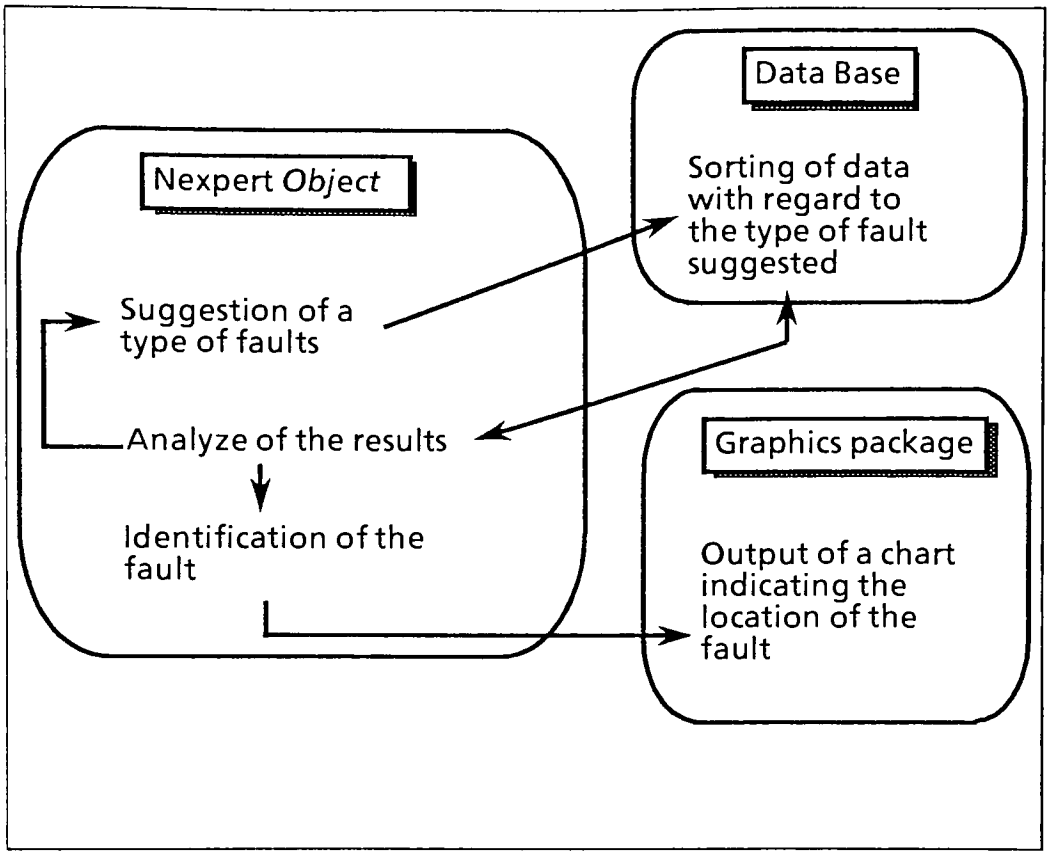


Figure 14. Nexpert: supervisor of computational tasks

reviewed regularly to keep a global consistency for the problem approach and the level of details addressed by the expert system. Since Nexpert offers to users different level of organization (knowledge base, knowledge islands, rules), this issue can be partially solved by using the adequate level of knowledge representation: this helps users to keep track of the level of details they are dealing with.

Expertise Elicitation

Problems encountered at this stage are relevant to knowledge engineering domain. They are mainly a consequence of the influence of the novice knowledge engineer's background. Among the different approaches,

we find exhaustive or algorithmic approach, logical approach, and systematic data-driven approach [Pero 87].

The exhaustive approach is a method used in management where decision trees or tables are current. The knowledge engineer tries to solve the problem by viewing all possible cases for each datum met during the reasoning elicitation. This method has a severe drawback in the case of a problem involving backward chaining or a reasoning involving a huge amount of data: the knowledge engineer can be overwhelmed before reaching the end of the problem. Even, if the expert uses this method to express himself, he usually does not use it when solving the same kind of problems on the fly and uses instead rules of thumb. Nevertheless, if the knowledge engineer enters table decision in Nexpert, it can have "unexpected" results. Because of the automatic goal generation, the system can review or jump to some rules (depending the way the decision table has been implemented) by "idea associations."

The logical approach corresponds to the Prolog-programming method and can be also considered more or less a Merise-like method. The developer begins by defining abstract problem entities. He pursues by emunerating all relations between these entities. He then concentrates on the choice of a data structure. The corresponding rules linking semantically these entities are also written. And finally, he checks for the global logical consistency. These steps, of course, are part of a cycle process. This method can be viewed as a natural approach to object-oriented programming system. But, here again, the number of data is a limiting factor and this approach can lead to some problems such as choosing an object structure not appropriate to the problem-solving method. Since Nexpert is a rule-oriented system, its principal inferring process relies on rules.

Finally, the data-driven method can be described as a forward chaining. For the same reasons as described previously, this can lead to problems. The main idea of knowledge engineering methodology is to rely on the method used by experts, but the knowledge engineer should be careful not to mix up the method used to solve the problem by the expert, and the mode of expression chosen by the expert: these two methods are often well embeded and it is the role of the knowledge engineer to isolate one method from the other method. In this, expert system shells can help the knowledge engineer by providing a way to show the expert what the knowledge engineer has understood.

Representation Mismatches

Different levels of representational mismatches can be encountered when the knowledge engineer enters chunks of knowledge in Nexpert formalisms. These mismatches mainly refer to the problem-solving structure, and the object structure [Pero 87].

Most novice users of Nexpert do not use the knowledge island concept, which provides the user with an interesting knowledge acquisition feature. As seen previously, knowledge islands correspond to different focus of attention an expert can have during solving a problem (see Figure 15). Instead of using this feature which has a natural origin in the mind modeling, they employ rules linked together by hypotheses. These hypotheses, in fact, tend to represent explicitly these changes of focus of attention, but complicate the task of knowledge base debugging. The other advantage of knowledge islands relies on their easiness to test part of the knowledge base, involving only one local independent chunk of knowledge. The modular representation of knowledge is not a method applicable only to software development.

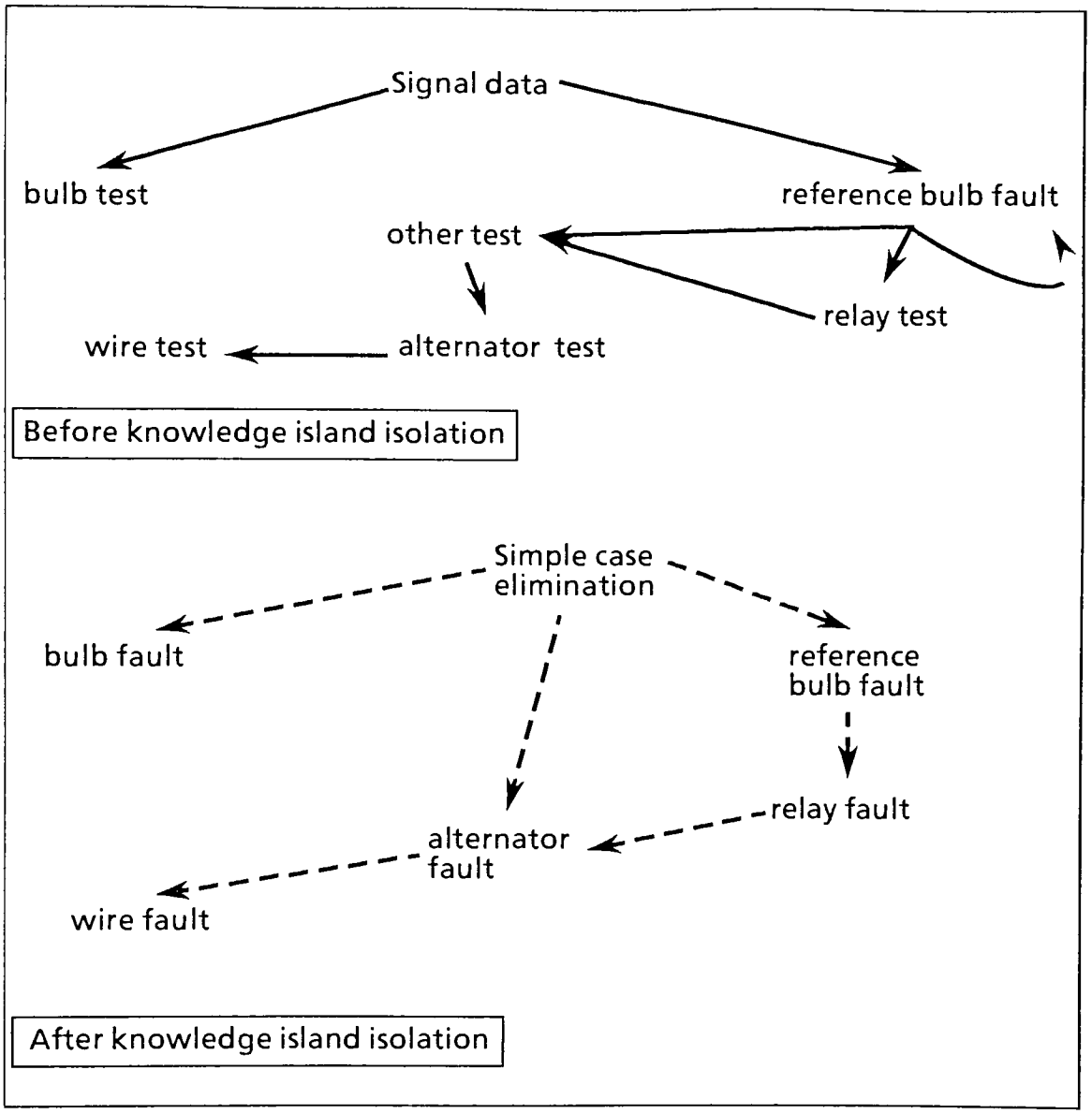


Figure15. With or without knowledge islands: bus door fault diagnostic

At a lower level of rule structure, we find the difficult dilemn "linear reasoning versus wide reasoning." By linear rule structure, we mean a series of hypotheses linked linearly together often in a tree (see Figure 16), in contrast with a wide structure which can be represented as network of hypotheses (see Figure 16).

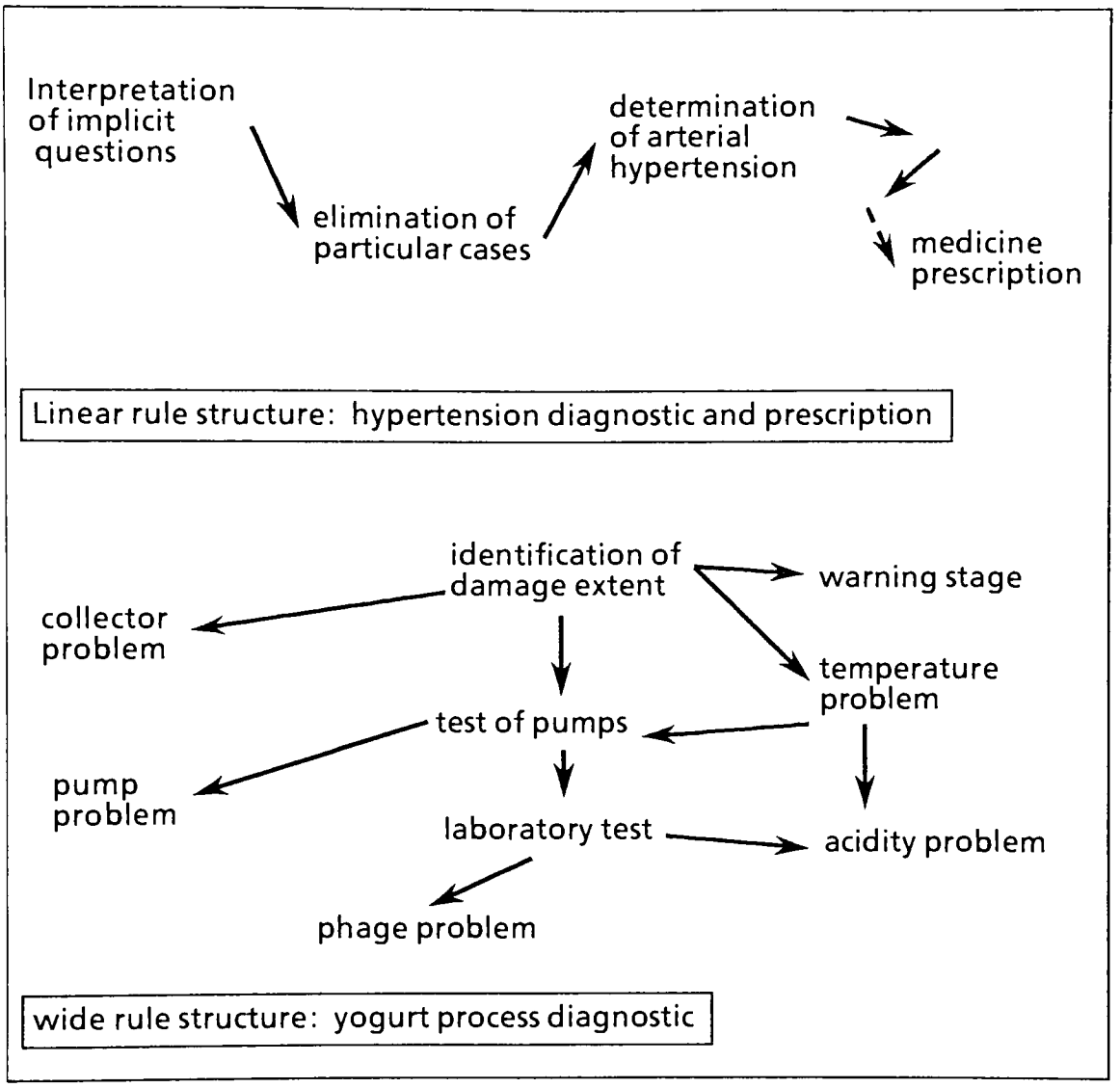


Figure 16. Linear rule structure vs wide rule structure

With object structure the main problem is to find the most appropriate structure to fit with the rule structure and the available facilities of Nexpert. For example, in a circuit diagnostic, we are dealing with sources, transmissions, elements, and receivers. The first object structure that comes to mind is the physical structure linking all these components together, but it is not the most appropriate structure. Also an object structure

representing the fault causal dependencies between the different components happens to be more easy to use jointly with diagnostic rule structure (see Figure 17).

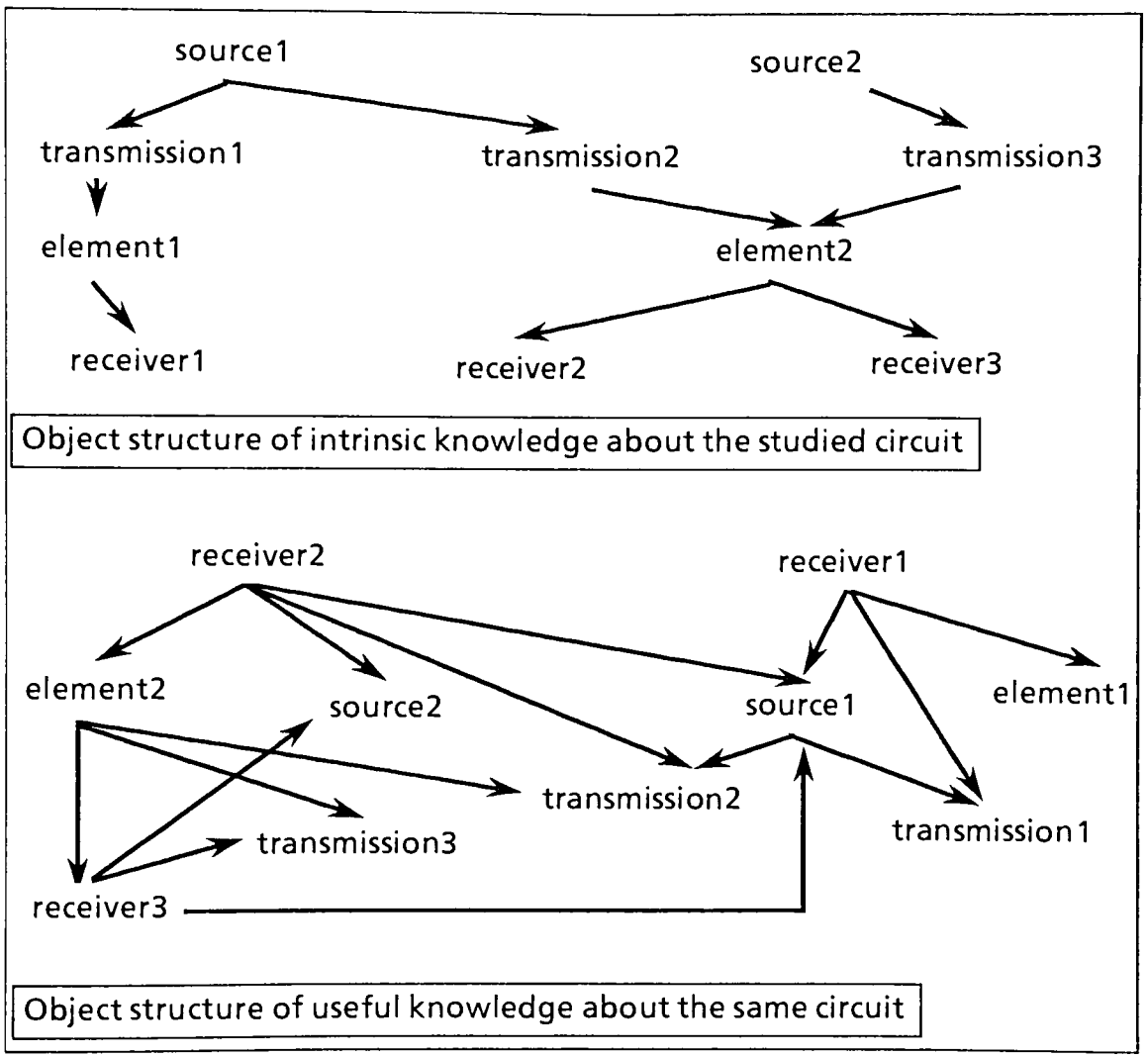


Figure17. Comparison of two object structures

CHAPTER 5

NEXPERT *OBJECT* AND A KNOWLEDGE ACQUISITION ASSISTANT

5.1 Goals of the program

The program written for this thesis has dual goals. It is concerned with both knowledge engineering methods and with Nexpert utilization techniques.

The knowledge engineering facet of the program dynamically offers advice about how to organize the construction of a knowledge base in the Nexpert environment. This refers mainly to my past experience of developing expert systems with this expert system shell, as well as to some general well-known strategies that are often forgotten in the fire of action. Among the advice that then can be given are the following key ideas: follow the expert's reasoning (even if it appears obvious, the knowledge engineer's background influences often negatively the knowledge encoding), decompose the problems into subproblems (weak methods are often useful in the early stages of knowledge base development), order the knowledge acquisition tasks (for example, it is time consuming to establish the session explanation in the early stage of the knowledge base development), and maintain readable and documented code.

In contrast, the Nexpert utilization facet of the program will help the user to avoid representational mismatches. Here we concentrate on the main Nexpert concepts, as we have identified in Section 4, that is, knowledge islands, rules, objects, and AI screens. These ideas will be developed in the following section.

Since this program presumes no predefined domain for the application, the only technique to analyze the method followed by the user relies on a

syntactic analysis of input (i.e. rules, objects and their attributes) and a rule or object structure analysis. This allows the detection of most common errors that can lead to dead-ends in expert system development.

However, some assumptions have been made to make this application realistic. It is first assumed that the knowledge about the problem can be formalized in the expert system formalism. Second, the knowledge base designer is supposed to know how to use the basic functions of Nexpert *Object*: it is not necessary for him to be an AI specialist or an experienced knowledge engineer. This program was built mainly for novice users of Nexpert *Object*. The third assumption concerns the size of the knowledge base, which must be small (no more than 200 rules): the program will be more efficient, and consequently will have a greater impact, at the early stages of the expert system development. Finally, the program, as we have said previously, is not supposed to have any knowledge about the domain. We do not claim to provide advice for all types of problems.

The reader will notice that the control of logical (or even semantic) consistency of the knowledge base is not taken into consideration; even though it could be very useful, event-driven reasoning and non-monotonic reasoning increase the complexity of such a task.

5.2 Principles and major functionalities

In such a syntactic analyzer of knowledge bases, three levels of analysis can be distinguished, and each level covers different issues of expert systems developed in the Nexpert environment. The first level of analysis is concerned with some aspects of project organization and corresponds to the knowledge engineering facet of the program. The two other analysis levels address issues

respectively in knowledge base architecture, that is, the choice of a knowledge representation schema, and in "low level" knowledge representation once a knowledge representation schema has been chosen by the knowledge engineer. These levels correspond to the Nexpert utilization facet of the program. The reader can refer to Appendix 3 for a complete list of warnings the program can provide to the knowledge engineer during the development of an expert system in the Nexpert environment.

The project analysis

A syntactic analyzer can provide only a limited amount of advice about project organization, which, however, may have an important impact on the current development of the expert system. The advice given at this level is not shell-dependent and could have been implemented with any other expert system shell.

The first set of advice is concerned with the life cycle of a project. By this we mean the organization of the different facets of a project, that is knowledge base design and tests, final user interface design, session explanation design and integration into the existing computer system (external program design), to give the most current facets of an expert system project. Even if an expert system project differs in several aspects from classical software projects, some characteristics taken from software engineering methodologies can be applied in most cases: these characteristics are relevant to the planning of the different facets of expert system development. It has been shown [Magu 88] that it is easier to concentrate on the knowledge base design and test in the early stages of the project development, while the final user interface as well as the session explanation

design are a major concern at the last stages of the project development (see Figure 18). Finally, the integration of the expert system into the existing

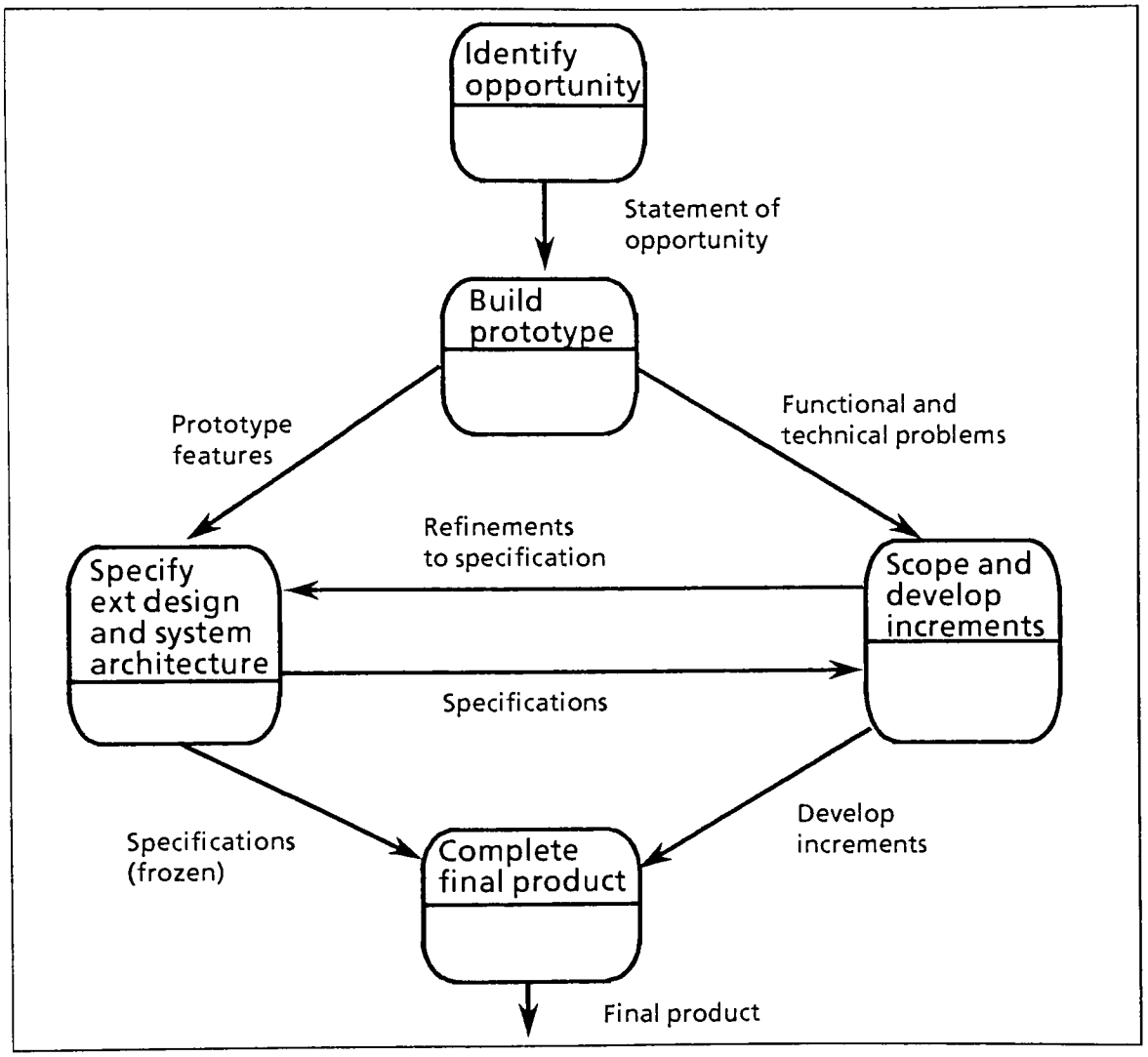


Figure 18. Planning for an expert system development (from Maguire, 1988)

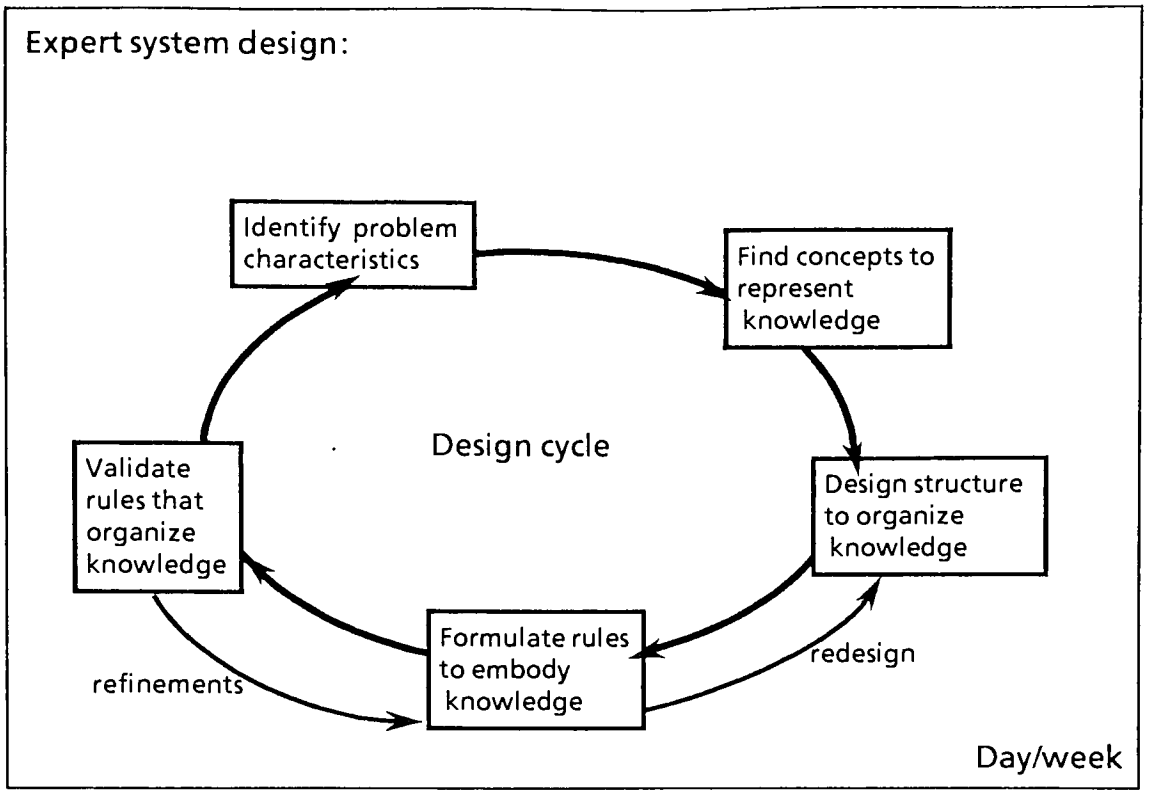
computer system must be the very last step of the project.

On a syntactic analysis basis it is then possible to detect if the knowledge designer tries to integrate interfaces and explanation features at a too early stage of the knowledge base project. In fact, in Nexpert *Object* such features correspond to actions used in rules or in methods attached to slots: among these actions are EXECUTE, to fire external programs or explanation

screens, and SHOW, to display explanation or conclusion screens. Consequently, the identification of early concerns about interfaces and/or external program integration is reduced to the detection of the use of those specific actions; this will fire the display of warnings on the screen to remind the knowledge engineer of the greatest concern of the moment.

As the reader can guess the critical step in an expert system development is the knowledge base design and test step. This phase is the goal of a second set of advice provided by the program. It is based on one main difference between classical software engineering methodologies and expert system development methodologies: importance of the cycle "design and test," which does not really exist at the day or week scale in a software engineering project (see Figure 19). Consequently, a syntax-based technique to remind the knowledge engineer, if necessary, of the importance of the test phase is the simple counting of knowledge chunks (basically the number of rules) recently created. If the number of recently created knowledge units reaches a predefined threshold, the program will display a message to remind the knowledge engineer of the importance of testing stored knowledge.

A last set of advice is related to maintenance issues. Two major concerns guide this part of analysis: project documentation and explanatory self-sufficient vocabulary of conceptual terms. Since the program techniques rely on the expert system input syntax, only a simple vocabulary checking can be done. However, we can notice that a very short name, let us say less than 3 characters, cannot carry a complete description of the concept it represents; in contrast, a too long name, more than 25 characters, looks like a sentence and can only bring confusion in the meaning of the concept it represents. The vocabulary checking is thus mainly based on the length of atom names. Another technique for vocabulary checking has been tried: comparison of



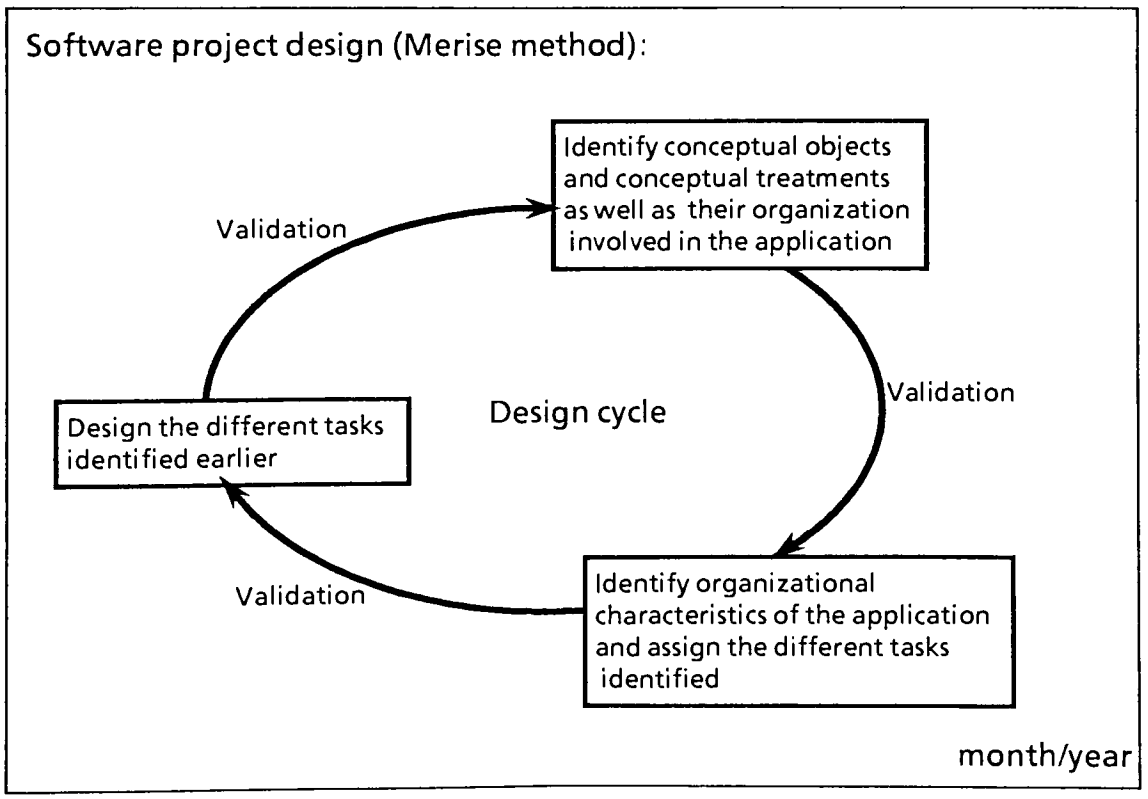


Figure 19. Comparison of design cycles between a software project and an expert system project

atom names with a list of meaningless words commonly used in programming such as "dummy." However, this technique involves the storage of a dictionary of meaningless words and for technical reasons (refer to Section 6) and has not been fully implemented.

Finally, the task concerning the detection of the most appropriate time to write documentation is far much more complex and is also application-dependent. The program can, then, just remind the knowledge engineer of this issue when he logs off. The knowledge engineer can use hypercard built-in product like Hypertrans [Veso 88] for knowledge documentation and as a work aid. A summary of the application, the expert's notes, interview notes, analysis of the expert's tasks, and a dictionary are among the information that needs to be saved during the development of the expert system.

Knowledge base architecture analysis

The knowledge base architecture analysis addresses classical issues in knowledge representation as well as concerns in knowledge representation schemes specific to the Nexpert environment.

A first set of advice tries to guide the knowledge engineer in the development of the high level architecture of the knowledge base. As seen in the Section 4, the knowledge engineer has the choice between two knowledge representation schemes that can be used at the same level of details: knowledge islands and knowledge bases. These two knowledge representation schemes are used to represent individual units of knowledge, that might be connected by a complex network. The major difference between them relies on dependencies: knowledge islands do not have any knowledge chunks in common (not even a datum); in contrast, knowledge bases might have several knowledge units in common. Consequently, it is more difficult to

control the automatic generation of goals via connecting data when using a knowledge base network. A criterion for detecting when knowledge bases are more appropriate than knowledge islands, then, is the size of knowledge islands automatically generated by Nexpert *Object*. If the average number of rules per knowledge island reaches a predefined threshold, the knowledge engineer is advised to subdivide his/her actual knowledge base into several smaller knowledge bases. This will also facilitate the testing and maintenance phases.

The connections between these individual units of knowledge are also analyzed in order to remind the knowledge engineer of the importance of the global architecture of the knowledge base. The program not only automatically warns the knowledge engineer when a new knowledge island is created, but it is also able to detect a lack of connections between existing knowledge islands. A last feature concerning knowledge islands is related to the network established between existing knowledge islands: a chain of knowledge islands shows a sequential reasoning and a lack of depth in the reasoning; new features could have been implemented. These last two functionalities are based on the same criterion: the average number of links per knowledge island. A lower threshold and an upper threshold for this criterion are the conditions for the program to display warnings about knowledge island architecture. A similar technique could have been implemented for the knowledge base network, since the connection between two knowledge bases simply corresponds to a LOAD action in a rule or in a method attached to a slot. However, a technical problem occurs since Nexpert *Object* does not keep track of a "map" of created knowledge bases like LightSpeedC does with a set of programs grouped into a project. Consequently, it is not possible for the program to keep track of the knowledge

bases loaded and unloaded, except with the following assumption: the knowledge engineer loads at least once all knowledge bases involved in the project so that the program can be aware of the set of knowledge bases involved in the project. This is obviously too restricting and has not been implemented.

A last set of advice concerns the hybrid aspect of Nexpert *Object*: these advice intend to guide the knowledge engineer in his/her choice between the different "low level" knowledge representation scheme available in Nexpert *Object* (rules versus objects). Unfortunately, only part of this problem can be solved by a syntaxic/structural analysis, that is, when objects can be used in rules. The program detects such a need when it finds at least 2 rules leading directly to the same conclusion with at least 3 conditions in common. To guide more completely the knowledge engineer in his/her choice, the program would have had a minimum domain knowledge, and its analysis would have then been based on the semantics of input.

"Low level" knowledge representation analysis

The analysis at this level is concerned with issues related to a specific "low level" knowledge representation schema (i.e. rules or objects). For each knowledge representation schema studied, the program provides the knowledge engineer with advice about the last entity created in this knowledge representation schema and also about its relations with other entities of the same knowledge representation schema.

Since Nexpert Object is mainly rule-oriented (even with its hybrid aspect), an emphasis has been put on rules. The analysis on rules themselves is based on several principles: among them are small rules for a better readability, and one rule per chunk of knowledge (for example, a situation or a

case to be identified). Consequently, the program bases its analysis of the rule on the number of conditions and/or actions. If a predefined threshold is reached, the program suggests the knowledge engineer to split the knowledge over at least 2 rules; this is done interactively with the user. In fact, if too many conditions have been detected, the program proposes a limit from which another rule will be built: by default, the program suggests to equally split the knowledge over 2 rules, and prompts the user one hand for confirmation (or for a new limit) and on another hand for the name of the intermediate rule, which will be chained to the simplified existing rule. The same behavior occurs when too many actions have been detected in the studied rule. A last feature of the analysis of a rule by itself is the detection of a "no condition," since a negative condition is always more difficult to test than a positive condition: it is mainly due to the fact that several paths can lead to a negative condition while only one path (in general) leads to a positive conditions.

After the analysis of the rule by itself, the program analyzes the impact of this new rule on the existing rule network. Among the characteristics of the rule structure which are analyzed by the program are the depth and the width of this rule structure, the possibility of clustering rules, and finally the possibility of isolating non-specific information, that is, a datum that occurs too many times in conditions. The analysis of the depth of the rule structure simply relies on the number of rules leading to the same hypothesis as the one of the studied rule, while the analysis of the width of the rule structure relies on the number of "levels" of the tree whose root is the studied rule. In the case where a linear or a deep rule structure has been found by the program, the program then displays a warning on the screen to make the knowledge engineer aware of the rule structure he/she is building. The detection of a potential clustering of rules uses the same kind of techniques: number of

rules with at least 3 conditions in common. A warning is then displayed on the screen to query the user about the possibility of clustering these rules into one unique rule. In contrast, the detection of a non-specific datum involves a deeper analysis of the rules: each condition of the studied rule is parsed to detect a datum whose average number of occurrences per rule is greater than a given threshold. In this case the program suggests the knowledge engineer to isolate this datum in an earlier set of rules in the rule network, since this datum corresponds to a non-specific criterion at this stage of the rule design.

The object analysis follows the same script. The last object created by the knowledge engineer is analyzed first in stand alone on a structural point of view and then replaced in its context among the already existing objects and classes. In stand alone, the program tries to detect if the object or the class is the appropriate knowledge representation of the concept to be entered: for this purpose, the program uses the number of properties, parents and children. A too big number of one of these characteristics is, for example, a sign of a technical design problem and the program then displays a warning on the screen. If the studied object has too many properties, the knowledge engineer might think about the real meaning of these properties: are they characteristics or more like components? In the latest case, the knowledge engineer could have chosen to use subobjects instead of properties. In the case of too many parents the knowledge engineer might cluster some of the studied object's parents into a class to define a new generic concept. In contrast, when too many children has been detected by the program, the knowledge engineer might think to add another level of objects.

After the analysis of the studied object by itself, the program analyses the impact of this object on the existing network on the basis of different criteria: the presence or not of a structure, and the depth of the object tree

whose root is the studied object itself. When the program tries to detect the presence of a meaningful object structure, that is, the presence of connections between objects, the use of classes to catch a generic concept, and the possibility of clustering existing objects, it refers to the average number of links per object and the average number of objects per class. Here again, when a limit has been violated, a warning is displayed on the screen, reminding the knowledge engineer of the role of the objects within rules and consequently, the necessity of an object structure. But, when the program tries to analyze the depth of the object structure terminating to the studied object, it refers to the number of "hierarchical" levels of this object structure. A too deep object structure fires the display of a warning about the right structure within rules.

In this chapter we have shown how a syntactic and structural analyzer of knowledge input can provide advice about knowledge engineering and knowledge representation issues connected to the expert system shell Nexpert *Object.*

CHAPTER 6

IMPLEMENTATION AND RESULTS

We will now present the techniques used to implement the ideas described in the previous chapter. We have chosen to run our program on a Macintosh™ (SE or II) since it was the only computer available at Rochester Institute of Rochester on which Nexpert *Object* could run, at the time we set up everything for this thesis. However, during the implementation of the program several other types of problems have been met: limited capabilities of the C compiler used (LightSpeedC™) and some restrictions of the actual commercialized version of Nexpert *Object*. This has led to the complete implementation of this knowledge acquisition assistant running along with a beta-version of Nexpert *Object*.

6.1 Architecture of the program

The functionalities described in the previous chapter imply that this program runs simultaneously with Nexpert *Object*. With the actual Macintosh version of Nexpert *Object* two techniques allow to have a second program running along with Nexpert *Object*: desk accessories running under the Finder™ or ordinary programs running under the Multifinder™ using the MPW™ library of Nexpert *Object* function calls. We have chosen the desk accessory architecture mainly because the MPW™ C compiler was not available at Rochester Institute of Rochester at the time we started the implementation, and also because the architecture of the program would have been totally different in the case of an ordinary program running under MultiFinder™.

Devil Mind as a desk accessory

A desk accessory is a "mini-application" (32 Kb code limitation) that can be run at the same time as a Macintosh™ application (in our case Nexpert *Object*) [Appl 86a]. The user opens a desk accessory by choosing it from the standard Apple menu (whose title is an apple symbol). Our desk accessory, Devil Mind, then displays a window on the desktop and that window becomes the active window (see Figure 20).

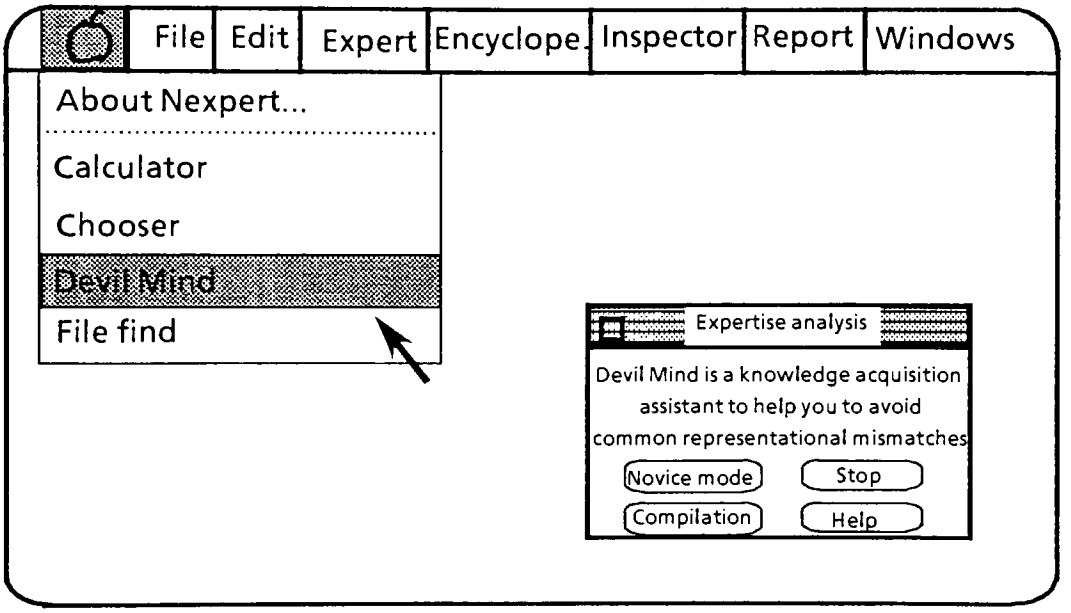


Figure 20. Chosen a desk accessory

After being opened, the desk accessory may be used as long as it is active: the user may ask for help by clicking on the button Help (see Figure 21); he/she may suspend the analysis for a while by clicking on the button Stop or reactivate the analysis by clicking on the same button, which is then labeled Continue; he/she may change by clicking on the button Novice from Novice mode to Expert mode where only the general analysis (project

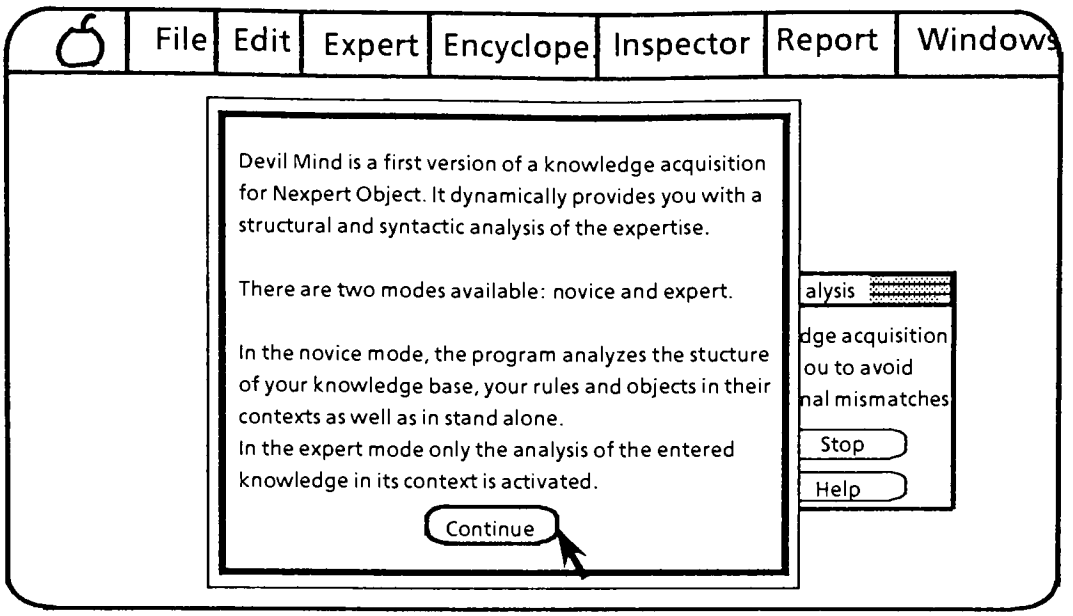


Figure 21. Help facility

and structure analyses) are available; he/she may as well switch back from Expert mode to Novice mode by clicking on the same button which is now labeled Expert; and finally, after each compilation of a rule, an object or a class, the user must click on the button Compilation to fire the analysis (levels 2 and 3, since the analysis level 1 is automatic) for technical reasons as we will see later in this chapter.

The user can activate other Nexpert *Object* windows (editor, network or control windows) by clicking outside the desk accessory window, and then whenever desired reactivate the desk accessory by clicking inside it. The user may quit the desk accessory by clicking the close box (small square at the upper left corner of the desk accessory window); this makes the desk accessory window disappear but the user is still in the Nexpert *Object* environment. The desk accessory is also automatically closed when leaving Nexpert *Object*.

The analysis (levels 2 and 3 only) is fired by clicking on the button Compilation, which implies the display of a modeless dialog window requiring the user to enter the name (or the number for a rule) of the atom created as well as its type (object, class or rule) and to then click on the button Ok (see Figure 22); the button Cancel allows the user to cancel the analysis

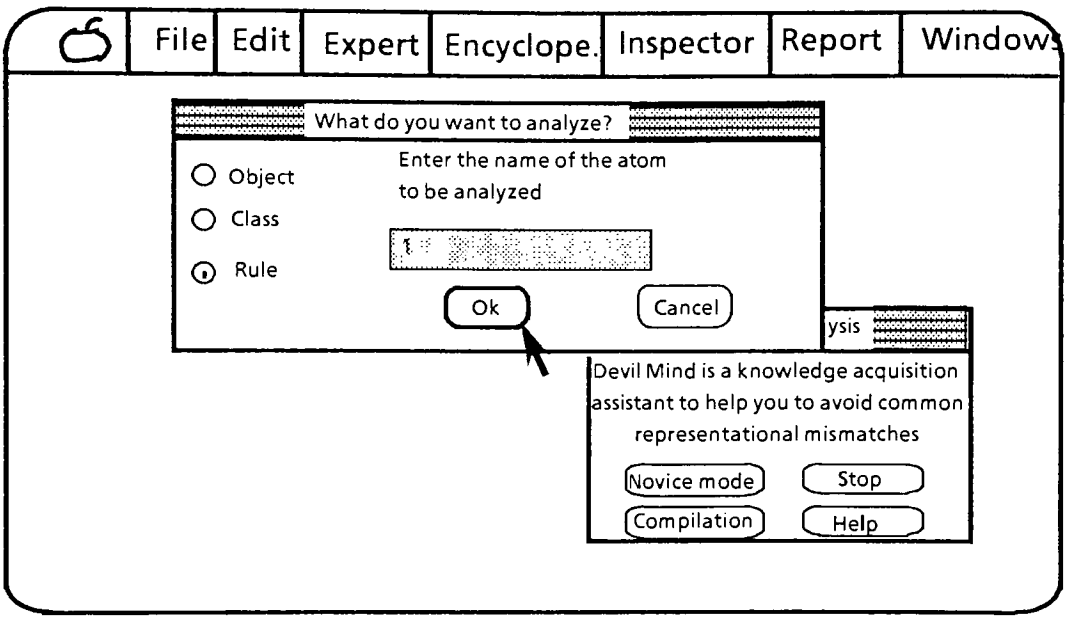


Figure 22. References of the atom to be analyzed.

request. The program when displaying this window forces the user to use only the features it provides.

Any warning resulting then from the expertise analysis is displayed in a modal dialog box, which grabs the stage and will disappear only when the user click on the button Ok (see Figure 23): this will enforce the knowledge engineer to pay attention to the warning, since it will not stay on the screen. The user may be prompted by the program for confirmation or more information as in the modification of a rule too big: in this case the user

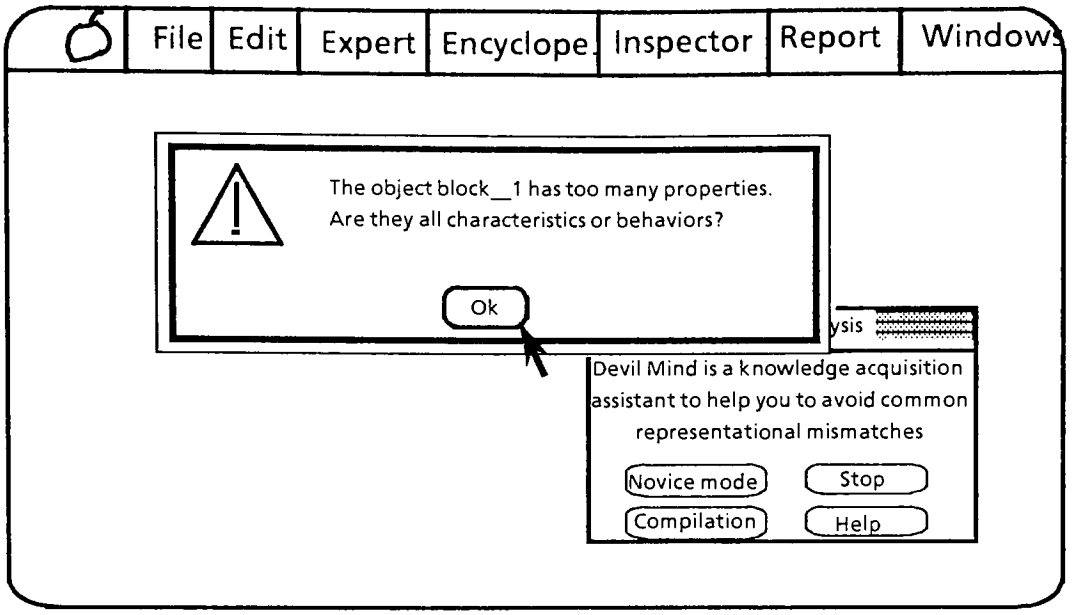


Figure 23. A typical warning window

just has to complete or correct the default information provided by the system and then to click on the button Ok (see Figure 24).

A last type of window (borrowed from Nexpert *Object* version 1.1) is displayed on the screen to warn the user of the creation of a new knowledge island (see Figure 25).

The desk accessory architecture

Since the Macintosh™ is event-driven, programs written for the Macintosh present a specific architecture [Swan 87]: it is all the more true with a desk accessory (a program built as a Macintosh™ driver) written in C with LightSpeedC™. As most event-driven programs a desk accessory has three fundamental parts plus a fourth part, which allows Nexpert *Object* to directly enter the program itself (see Figure 26):

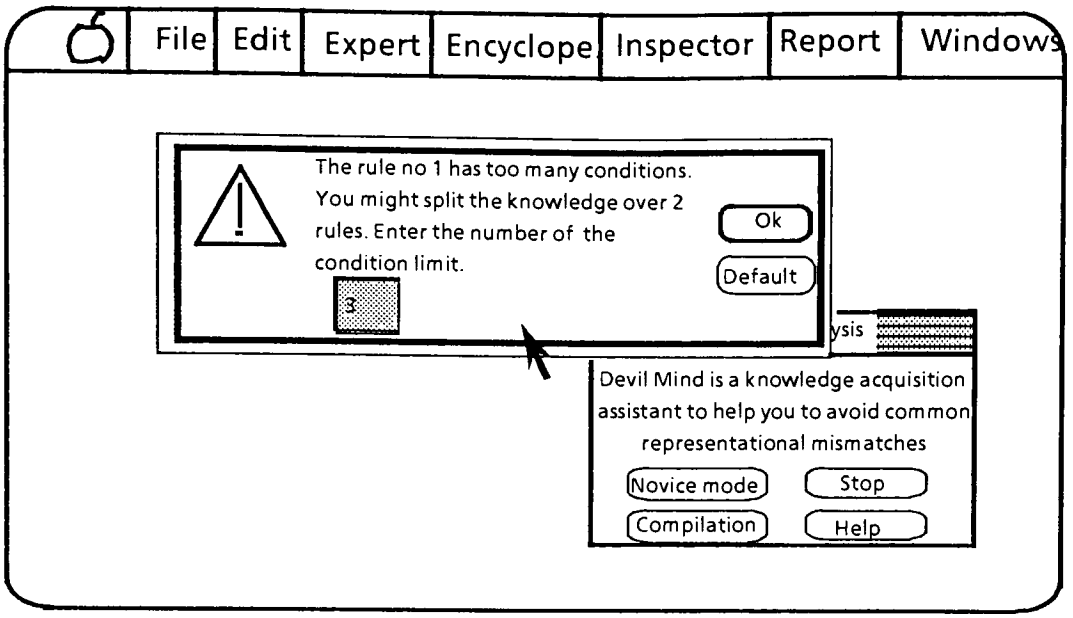


Figure 24. The user is queried for more information

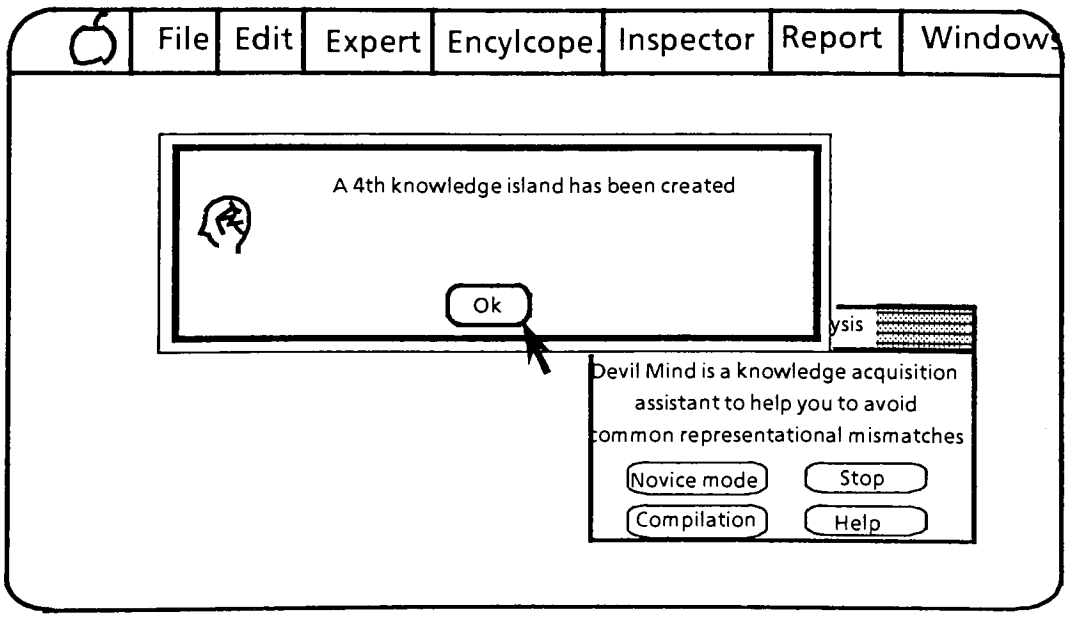


Figure 25. A new knowledge island has been created

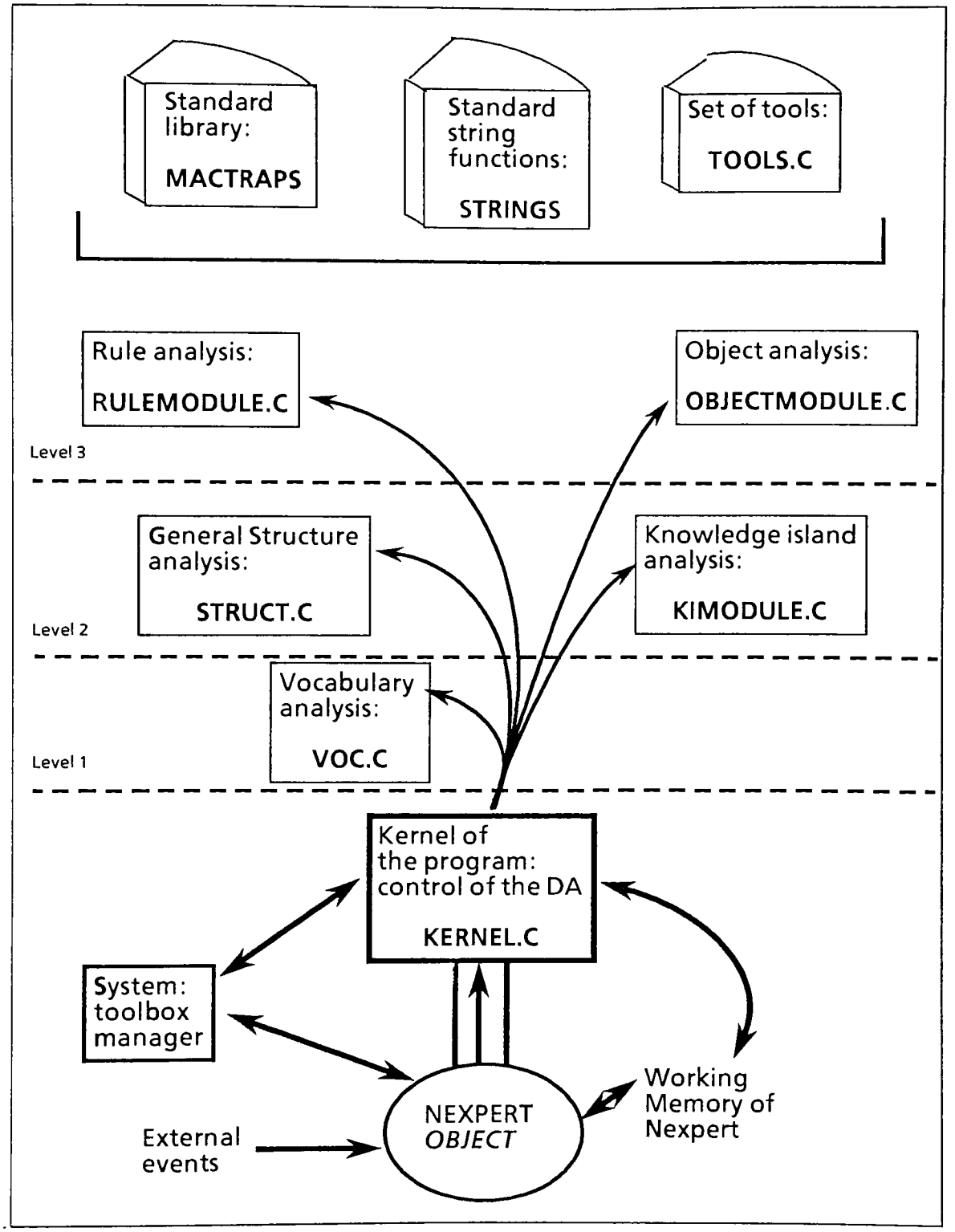


Figure 26. Global Architecture of the project

- Program entry point handler. Five system entry points allow Nexpert *Object* via the toolbox manager to send to the desk accessory its different events. The entry points used in this desk accessory are: open driver entry point, close driver entry point, and control driver entry point (prime driver and status driver entry points are not used).
- Event handlers, which response to the different system events (window open, activate or close events, mouse-down events, key-down events, auto-key events, and window update events).
- Program actions: the analysis itself (levels 2 and 3, mainly).
- Direct entry point for Nexpert *Object*: part of the analysis (level 1).

As a classical desk accessory it is dependent on the application is running along with: in our case, Nexpert *Object*. Nexpert *Object* sends to the desk accessory (via the toolbox manager) system events such as window, mouse, and keyboard events, that belong to the desk accessory. The program entry point handler (main() in kernel.c) directs the event flow between the functions DRVROpen(), DRVRClose() and DRVRControl() (the functions DRVRPrime() and DRVRStatus() are not used in this desk accessory).

The DRVROpen() has the responsibility to display an introduction screen (see Figure 27) followed by the desk accessory window (see Figure 20). Our desk accessory ignores multiple window opening. It is also in charge of establishing the direct communication with Nexpert *Object* (see the following section for the communication calls with Nexpert) and of the initialization of all needed variables (number of knowledge islands, of rules, flag of the beginning of a session, resource ID, window handle).

The DRVRClose() has the responsibility to close the desk accessory window and to remind the user of the importance of the documentation in the

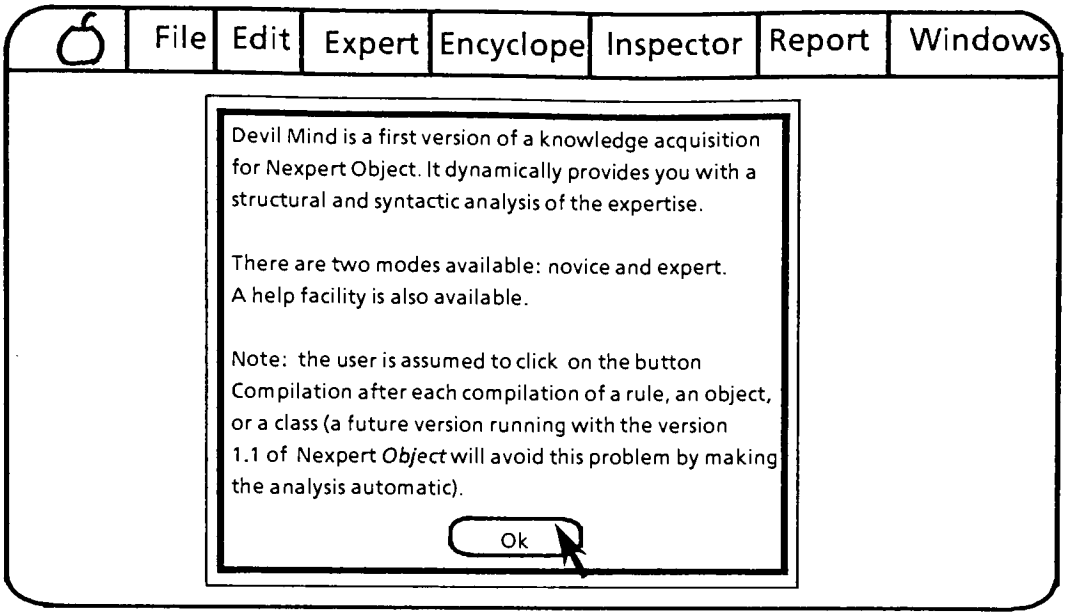


Figure 27. Introduction screen

development of expert systems (see Figure 28); it also terminates the direct communication with Nexpert *Object* and frees the memory allocated for the needed variables.

The DRVRControl() is the most important function since it responses to system events such as window refreshing, mouse actions associated to buttons such as switching to another mode, suspending a session, displaying the help screens or firing the analysis (with the functions doCtlEvent() and launchModule()). The reader can refer to the Appendix 3 for a more detailed description of the different functions involved in this application.

Also, Nexpert *Object* can enter directly the desk accessory code, once the direct communication has been established with the desk accessory. We will describe now the specific communication between Nexpert *Object* and our desk accessory.

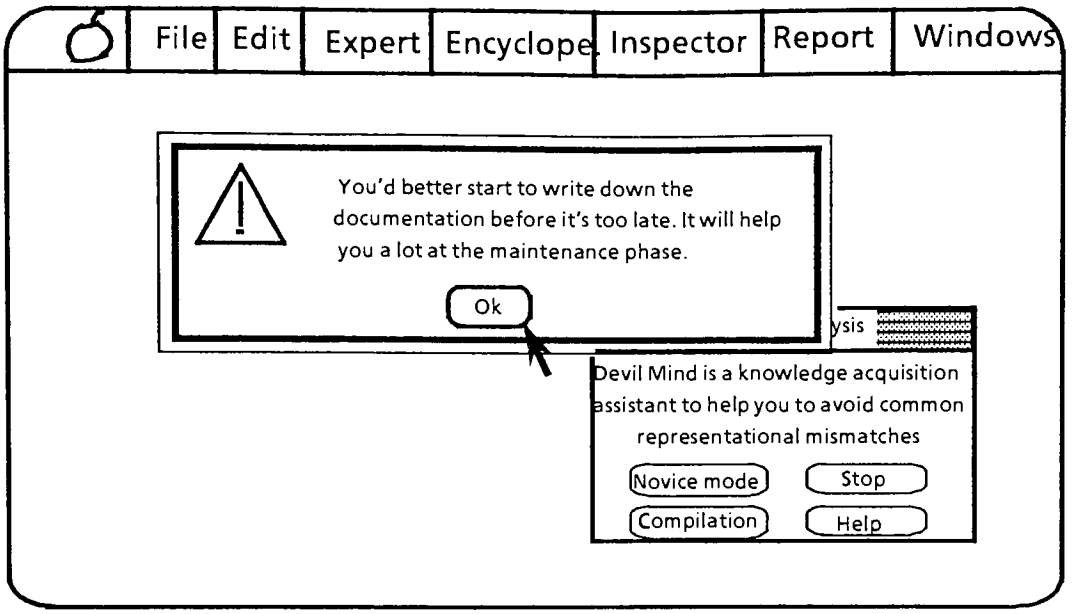


Figure 28. Documentation warning window before leavingthe program

6.2 Communication with Nexpert

Nexpert *Object* supports both call-in and call-out capabilities [Neur 88a]; these functionalities allow our desk accessory to investigate the working memory of Nexpert for information about objects, rules or object links, and to be notified of events specific to Nexpert such as the creation of a new atom. In fact, Nexpert *Object* is provided with a library of routines, which can be called from external programs such as desk accessories.

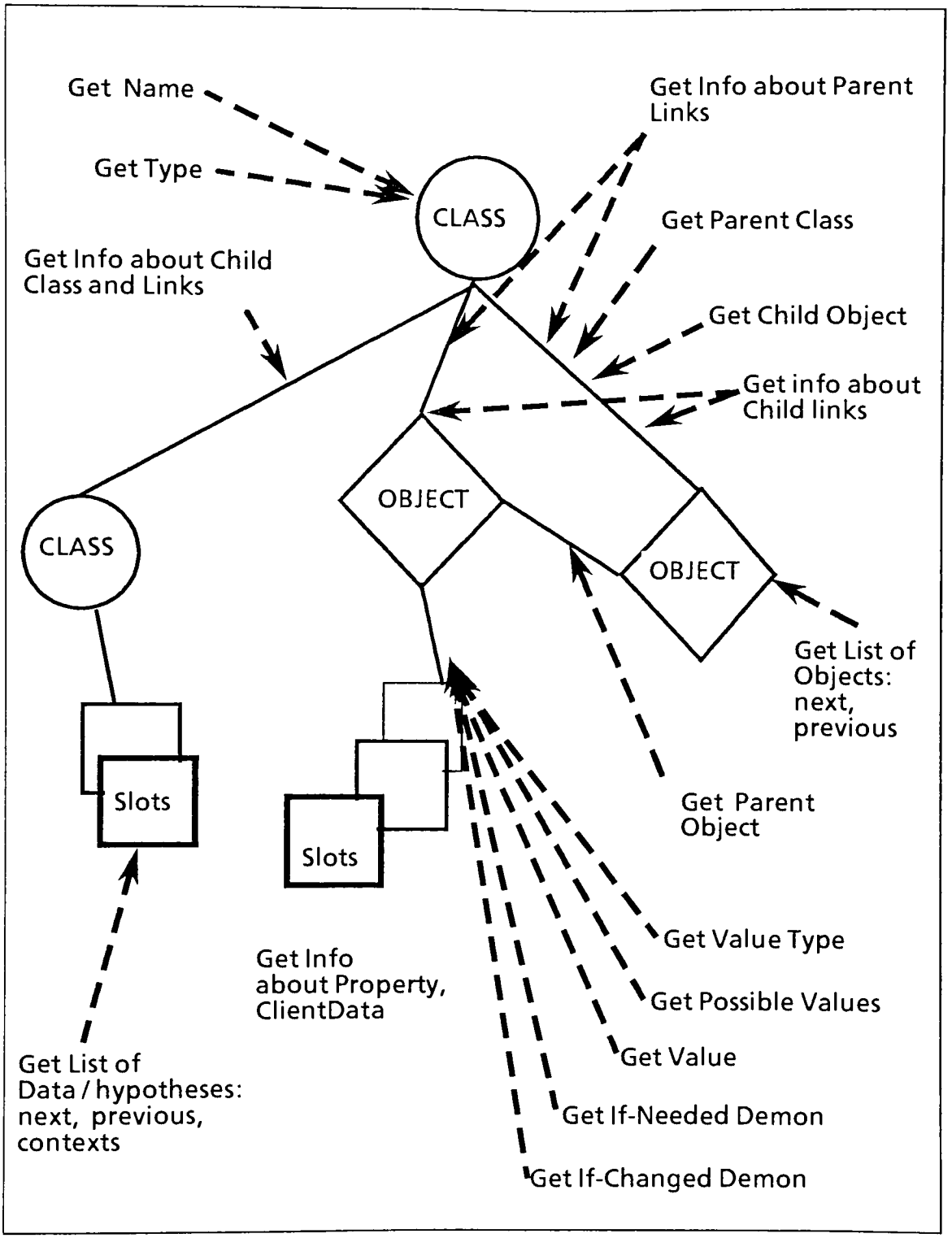


Figure 29. Information about the object structure that an extern program can obtain via Nexpert callable interface [Neur 88a]
(from Neuron Data)

Devil Mind and the Nexpert callable interface

Calling-in refers to the ability of a program such as a desk accessory to call functions of the AI kernel. Our desk accessory can then investigate the working memory of the expert system and read the structural information concerning objects or classes such as names and parent-child relationships, as well as the information concerning the rules themselves (that is, hypothesis, conditions, actions, and context links) (see igure 29). These functions also allow our desk accessory to modify the working memory of the expert system: create or modify a new rule, for example (see Table 4).

In contrast calling-out refers to the ability of Nexpert to call an external program (in our case, the function MyNotify() of kernel.c) which has set up the direct "communication path" with Nexpert using the NXP__SetHandler call (done when the desk accessory is opened in our case by the DRVROpen() function). This call installs inside the AI kernel all the functions which perform communication between the AI kernel and its interface. It installs in particular a Notify Handler, that is called each time a change occurs in the working memory (creation of objects or links, for example). This is used to detect the creation of a new atom (object, class or hypothesis) and to fire the vocabulary analysis (function CheckObj of Voc.c). In the beta-version 1.1 of Nexpert *Object*, this handler can also be used to detect the end of compilation of a chunk of knowledge (object, class or rule), so that the knowledge base architecture analysis and the "low level" knowledge representation analysis can be fired automatically without any intervention of the knowledge engineer.

Usage	Nexpert callable interface calls
To establish the communication with Nexpert	
setting client handler	NXP_SetHandler(, ,)
To get data from the emerging knowledge base	
Handler of a given Memory object	NXP_GetAtomId(, ,)
Get information about	NXP_GetAtomInfo(, Code, , , , ,) with Code equals to
the name of an atom	NXP_AINFO_NAME
a class (child class,	NXP_AINFO_CHILDCLASS
child object,	NXP_AINFO_CHILDOBJECT
parent class,	NXP_AINFO_PARENTCLASS
property)	NXP_AINFO_SLOT
an object (parent class,	
parent object,	NXP_AINFO_PARENTOBJECT
parent class,	NXP_AINFO_PARENTCLASS
child object,	NXP_AINFO_CHILDOBJECT
property)	NXP_AINFO_SLOT
a slot (value type)	NXP_AINFO_VALUETYPE
a rule(conditions,	NXP_AINFO_LHS (with NXP_CELL_COL1, NXP_CELL_COL2, and NXP_CELL_COL3)
actions,	NXP_AINFO_RHS (with NXP_CELL_COL1, NXP_CELL_COL2, and NXP_CELL_COL3)
hypotheses)	NXP_AINFO_HYPO
a hypothesis (contexts)	NXP_AINFO_CONTEXT*
a list of atoms	NXP_AINFO_NEXT/PREV (with NXP_ATYPE_RULE, NXP_ATYPE_OBJECT, NXP_ATYPE_CLASS)
To display a message	NXP_SetData(Code,,,,) with Code equals to NXP_WIN_BANNER
To edit chunks of knowledge	
create or modify a rule	NXP_COMPILE(,)*
delete a rule	NXP_Edit(NXP_EDIT_DELETE, Type, ,) with Type equals to NXP_ATYPE_RULE

Table. 4. Some Nexpert callable interface calls
used in the program. (* available only with the version 1.1 of Nexpert *Object*)

The different modules of analysis

Once the analysis (levels 2 and 3) has been requested, the desk accessory (launchModule() function of kernel.c) identifies what level of analysis must be done. If the Novice mode is ON, the chunk of knowledge created is analyzed first in stand alone and then replaced in its context. If the Expert mode is ON, the chunk of knowledge created is just analyzed in its context.

If the knowledge engineer has created an object or a class using the object or the class editor, the program fired the function objModule() of ObjModule.c. The program successively retrieves the number of parents, children and properties of this chunk of knowledge for analysis purpose and displays, if necessary, a warning on the screen (DisplayDlg() function of Tools.c). It also explores the depth of the tree whose root is the analyzed atom to analyze the depth of this object structure by walking through the whole tree (a leaf is a parent without any parent). The program then analyzes this chunk of knowledge in its global context (StructAnalysis() function of Struct.c); for this, it computes the average number of links per object by counting the number of links for each object in the object agenda of Nexpert. It also computes the average number of objects per class by counting both the number of "real" objects (that is, objects created by the knowledge engineer and not those automatically created by Nexpert) in the object agenda and the number of classes in the class agenda of Nexpert.

If the knowledge engineer has created a rule using the rule editor, the program launch the ruleModule() of RuleModule.c. It first analyzes if the rule created belongs to a linear structure (linearStructAnalysis() function of RuleModule.c) or a deep structure (depthStructure() function of

RuleModule.c). It then pursues with the analysis for needs of objects (Objectneed() function of RuleModule.c) and common conditions with other rules (CommonNeed() function of RuleModule.c). It also detects the presence or not of a negative condition (noSearch() function of RuleModule.c) in the rule created. It finally analyzes the rule created by itself since it can involve some modifications in the rule by retrieving the number of conditions and actions of this rule. Whenever required a warning is displayed or more information queried.

After this quick analysis of the rule, the system will analyze the set of rules created as well as knowledge islands of the knowledge base. This part of the analysis involves more work since Nexpert offers only basic information about rules such as conditions, actions and hypotheses. The program has to detect from these information if a new knowledge island has been created (KISearch() function of KiModule.c) by parsing each condition, each action as well as the hypothesis of the last rule created to identify each atom used in this rule; and then it has to compare each one of such atoms with any atom contained in each condition, each action and hypothesis of any other rules previously entered. We have used for this purpose a simple parser called myParser() (in KiModule.c) to extract atom names from a condition or an action which can be a complex expression such as:

```
>=  Pump1.Width*Pump1.Height*Pump1.Length +
        Pump2.Width*Pump2.Height*Pump2.Length        12.3
```

which stands for "Is the volume of the Pump1 added to the volume of the Pump2 greater than 12.3?". In this example, the atoms to be identified are Pump1.Width, Pump1.Height, Pump1.Length, Pump2.Width, Pump2.Height,

and Pump2.Length. Expressions can even involve more complex formulas such as Log() or Ceil() mixed up with simple data like Alarm_detected. However, some assumptions have been made to decrease a little bit the number of possibilities: the program assumes that the rules created do not contain list for hypothesis. This assumption was implicitly said when we were assuming the knowledge engineer is a novice user of Nexpert *Object*, since such lists are used mainly by experimented Nexpert users and it will involve a lot of overlay in our program. Once the program has been able to detect if a new knowledge island has been created, it can analyze context links between hypotheses (KIStat() function of KiModule.c). A simplified version of the previous parser is used to detect specific information, since only rule conditions are involved in this case (GateParser() function of KiModule.c).

Here again, whenever needed, the program displays a warning on the screen. This concludes the analysis request. The desk accessory and Nexpert are again ready for a new request.

The reader will find in the Appendix 3 a complete description of each function used in our desk accessory.

6.3 Results

The basic source of the analysis comes from my past experience using Nexpert *Object* in expert system development and the application developed at Cornell University (the reader will find a complete analysis of this project in the Appendix 2).

However, as said earlier, the 32 Kb code limitation has implied some restrictions both on the user interface (warning phrasing and user-friendly window) and on the program capabilities . On the interface point of view, the query window for subdividing a rule too many conditions, for example, could

have displayed the list of conditions involved in the modification. We had the source to do it, but it is written in MPW™ C and it appeared difficult to translate it into LightSpeedC™ without reference manuals (toolbox calls do not seem to be the same). On the program capability point of view, the dictionary of meaningless names could not have been implemented, since it was requiring handling a file and consequently needed a lot of more code lines.

We have also been obliged to include another feature in our program because of the difference of capabilities between the Nexpert *Object* versions 1.0 and 1.1. All desired Nexpert calls are not available with the version 1.0 of Nexpert *Object*: for example, the creation of a rule, the access to context links between hypotheses, the association of private data to an object or a class, the notification of the end of compilation of an atom are not available in the version 1.0 of Nexpert. However, they have been implemented in our desk accessory, but they are used only if our program identify the running version of Nexpert *Object* as the version 1.1. Consequently, our program is less powerful when running along with the version 1.0 of Nexpert (a very simple knowledge island analysis is done) than when running with the version 1.1. We have to specify that a specific notification has been implemented in Nexpert so that it will be possible for our program to be notified of the end of compilation of an object, a class or a rule. Until this version we were constrained to fake this detection by requiring the user to click on the button Compilation.

Another kind of problems met during the implementation of our desk accessory comes from the fact we were working with a beta-version of Nexpert *Object*; it was particularly difficult to find out where the bugs came from: from our desk accessory (we are a novice programmer on Macintosh™ and the Macintosh™ is famous for its user-friendly interface and well-known for its

difficulties to program with, especially in C, since the toolbox is originally written in Pascal).

Despite these problems we manage to have a working version even if less powerful on the version 1.0 of Nexpert *Object*.

CHAPTER 7
CONCLUSION

The choice of a specific expert system shell, I already knew, allowed me to focus my attention on knowledge acquisition issues. I was then able to build up a knowledge acquisition assistant running along with this expert system shell. This program is both a knowledge acquisition tool and a knowledge acquisition aid, since it informs the user of knowledge representation mismatches and guides him/her in improving the project organization. However, the knowledge engineer is still free to accept or to refuse the potential corrections to be done on the emerging knowledge base.

About knowledge acquisition expertise

This interaction between the knowledge engineer and the knowledge acquisition assistant merely looks like the interaction between an expert (in our case an expert in knowledge base construction with the Nexpert environment) and an apprentice (here, the knowledge engineer). In fact, we have thought at the beginning of another way to implement a knowledge acquisition assistant which can be more appropriate and more flexible: an approach using knowledge bases instead of C procedures.

Because the different levels of analysis of our program involve expert knowledge in knowledge acquisition which is obviously not complete, constant incremental updatings of the analyses must be done along with new results in knowledge base development. We felt several times during this program development the necessity of implementing new features, that is, new chunk of analysis such as the detection of a non-specific datum or the detection of an early concern about the final-user interface. And these changes have implied

the difficult task of developing new modules and integrating these modules in the existing analysis. It will have been easier to write the analysis itself as a set of rules connected with external programs to retrieve the necessary syntactic information from the Nexpert working memory and compute the values of our different criteria (see Figure 30).

Implementation of new features was not our only concern: the values chosen as thresholds for our different criteria were subject of controversies like some analysis conclusions irrelevant in some cases. And one main argument was the relations between threshold values or analysis conclusions, and domains or tasks. Let us take the example of the object analysis: we defined an average object as an object with less than 6 children. But if the application to be coded in Nexpert is, for example, a design problem, it often involves a lot of objects representing temporary solutions along with a deep object structure. And in this case the average size for an object can quickly reach our limit without, however, representing a major knowledge representation mismatch, since we need to keep the maximum of information for the revision of proposed solutions. But, the situation is different when the application is a model-driven diagnostic where objects represent a functional model: the object structure must then be easy to read for maintenance purpose.

The domain can in the same manner influence the analysis; a medical diagnosis, for example, does not emphazise the subdivision of the problem into independent and individual subproblems as much as an engineering diagnosis (where locating a failure in an equipment is often a well organized task). In a medical diagnosis the subdivision into knowledge islands can even appear

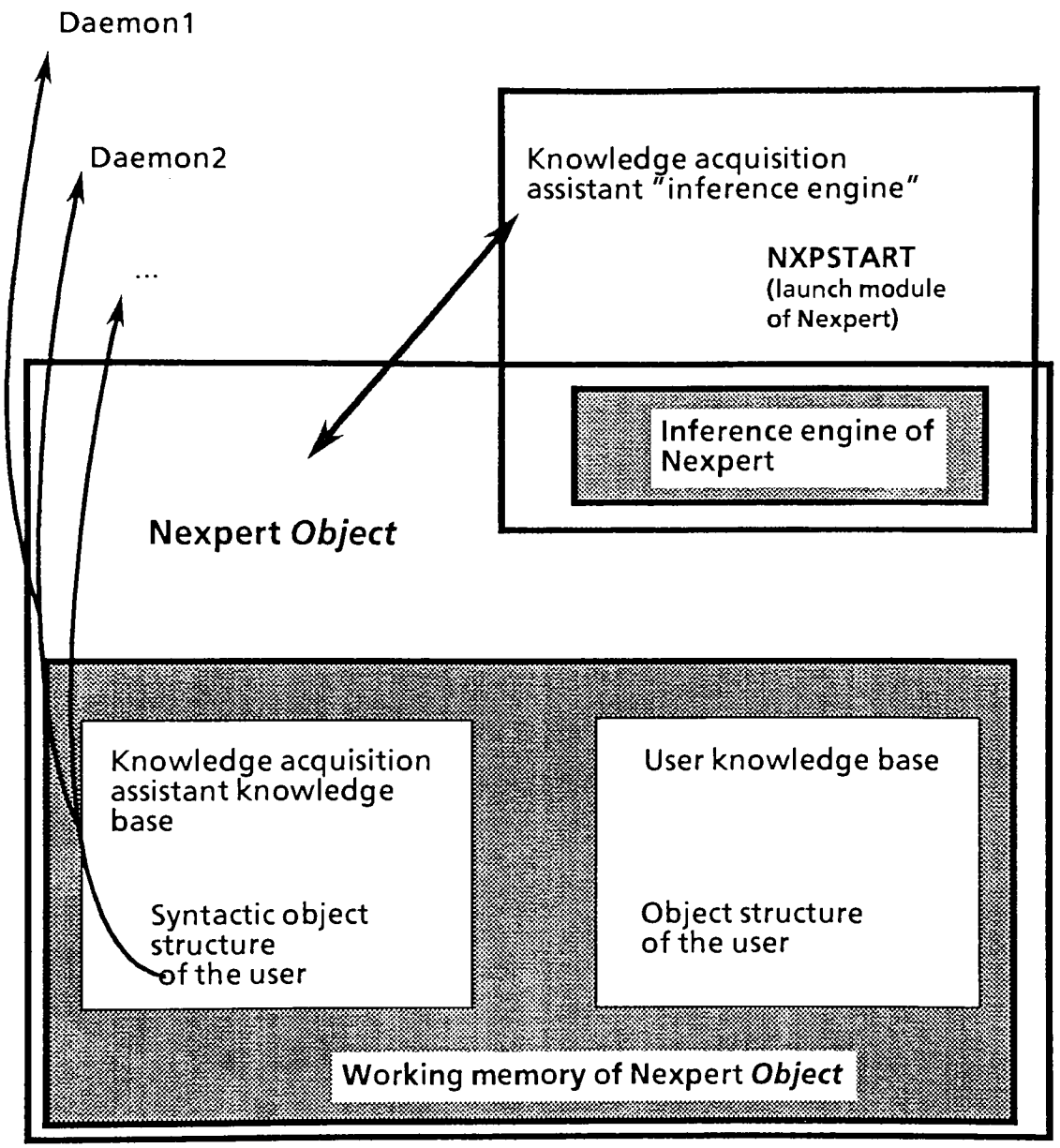


Figure 30. A knowledge acquisition assistant viewed as a knowledge base written using Nexpert.

awkward, which is not the case of an engineering diagnosis where tests can be grouped by independent pieces of equipment.

The easiness in updating and integrating new constraints such as domain or task constraints is not the only advantage of implementing the

knowledge acquisition expertise in a knowledge base. Another built-in facility of expert systems could have been used: explanation. Whenever our knowledge acquisition assistant reaches a conclusion, the knowledge engineer can ask how that conclusion was reached and what rules were used to deduce it. He/she can also have access to the syntactic information concerning his/her knowledge base such as knowledge island references.

Using Nexpert as a meta-language for knowledge engineering support

With this approach we can identify three kinds of knowledge in our knowledge acquisition assistant: procedural knowledge for "inference engine" of the knowledge acquisition assistant (the highest level of control), judmental knowledge for the analyses themselves associated with a set of procedures for retrieving and computing syntactic information about the emerging knowledge base. For example the objModule can be rewritten as it is shown in Figure 31.

If we use an expert system shell as Nexpert for knowledge engineering support (refer to Figure 30), the working memory should be divided into two sections: a first section for the knowledge acquisition expertise, and another section for the user expertise. The system should also not behave the same way with both knowledge base even if the same inference engine is used: the run time mode should be used for the knowledge acquisition expertise while the editing mode should be used for the user expertise. With such an architecture the user is able to edit and test his/her expertise and to gain knowledge from the knowledge acquisition expertise.

A script for a typical knowledge base editing in Nexpert using this kind of implementation can be the following: the user launches Nexpert. In the same time Nexpert looks up in the NXPSTART file to establish the accessory

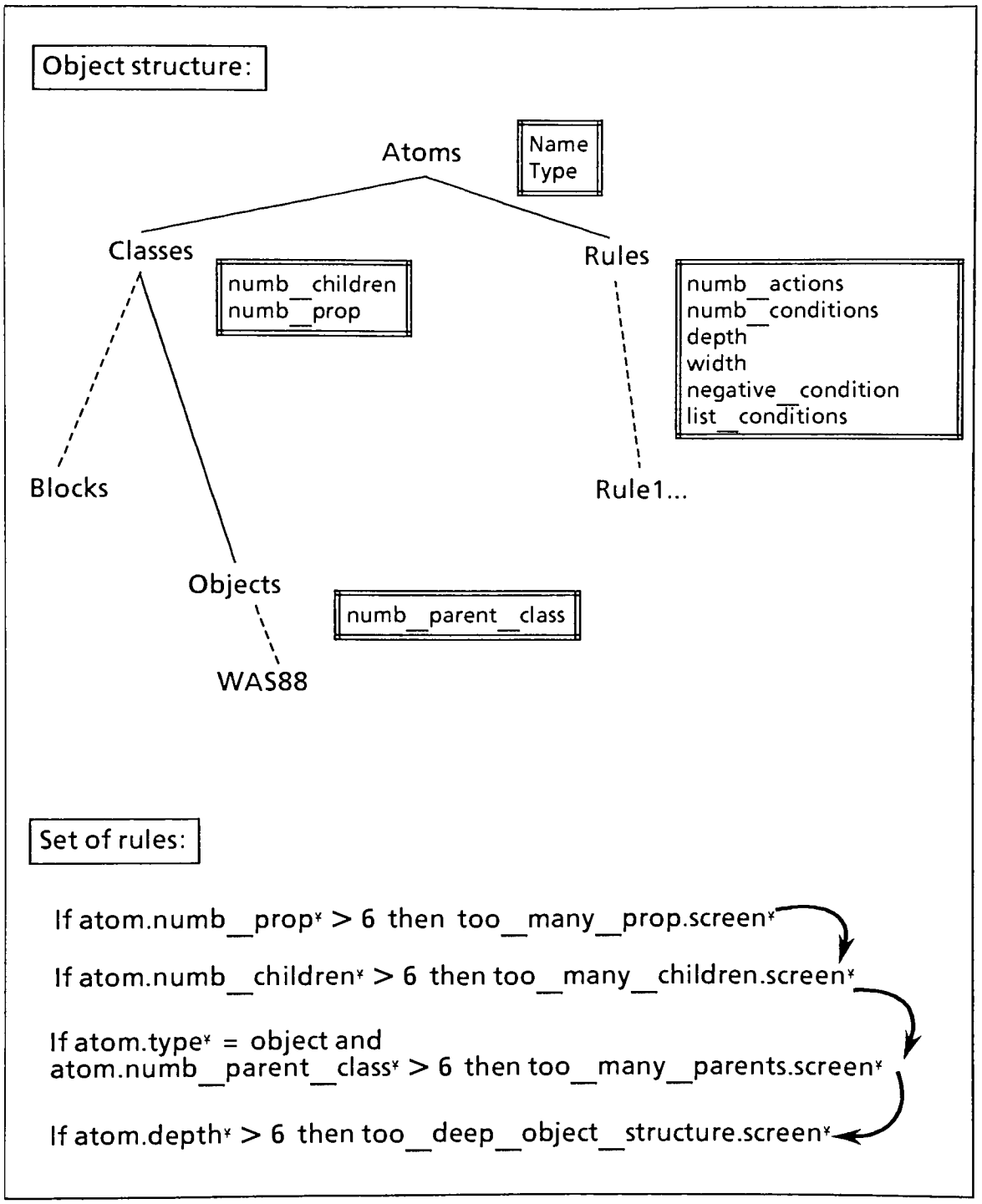


Figure 31. Knowledge representation of the analysis contained in objModule.c
(* stands for daemons).

actions it has to take beside its default behavior as a classical expert system: this is at this time that the highest level of control of the knowledge

acquisition assistant (its "inference engine") will be set up. The user is then in the classical editing environment of Nexpert with, however, in the background a knowledge base ready to be automatically fired when needed. He/she can edit a new knowledge base. At each object/class/rule compilation the "inference engine" of the knowledge acquisition assistant is activated and the knowledge acquisition knowledge base is fired: at this time, the system should automatically switched into the run time mode. The user can then take advantage of the knowledge acquisition expertise stored within Nexpert. However, the system should automatically return into the editing mode oncethe new chunk of knowledge has been analyzed.

An interesting feature of such implementation is that all analysis information are available (via editors and networks) in consultation mode, at any time, and in the same environment as the development environment the user is used to. This view of an knowledge acquisition assistant can correspond to another project.

Another facet: the tutor

Another facet could have been implemented: a tutorial facet. A program with such a feature can keep track of the knowledge engineer's progresses and modify the different thresholds involved in the analyses. For example, if the knowledge engineer keeps writing rules with too many conditions, the threshold allowing the program to detect such a knowledge representation mismatch can be decreased: the knowledge engineer will then be notified more frequently. But, when the knowledge engineer writes rules with an appropriate number of conditions, the corresponding threshold can be increased, and he/she will be less often notified of this kind of errors. Based on this threshold modification, a knowledge acquisition can help a novice user of

the expert system shell by teaching him/her how to choose the most appropriate knowledge representation schemes (the user can in the same time concentrate on his/her application problem) and by focusing on his/her main knowledge acquisition problems.

Such a need for filling the gap between the powerful AI formalisms and the multi-facet human problem-solving formalism becomes more and more important with applications involving wide and deep domain knowledge. To fill this gap is mainly the goal of user-friendly interfaces such as natural language interface or knowledge editors, which in fact add a level between the complex AI techniques and the knowledge acquisition interface. Some commercialized products begin to occur on the marketplace and to be used in companies (such as Boeing with AQUINAS).

REFERENCE LIST

KNOWLEDGE ENGINEERING AND EXPERT SYSTEMS

[Blan 87] Blanchard H.. "Le processus d'extraction des connaissances". [The knowledge extraction process], in Les systemes experts. Switzland: Le Concept Moderne Editions, 1987, 35-42.

[Bonn 86] Bonnet A., J-P. Haton, Truong-Ngoc. "Systemes experts: vers la maitrise technique". [Expert sytems: toward a technical mastering]. France: InterEdition, 1986.

[Coul 87] Coulson R. N., L. J. Folse, D. K. Loh. "Artificial intelligence and natural resource management", Science, 237 (july 17, 1987): 262-267.

[Estr 87] Estrangin B. "Methodologie pour systemes experts de seconde generation". [Methodology for second generation expert systems]. France: Eurequip, 1987.

[Gamm 84] Gammack J. G., R. M. Young. Psychological techniques for eliciting expert knowledge", in Research and development in expert systems, the Proceedings of the fourth technical conference of the British Computer Society Specialist Group on Expert Systems, Warwick, December 18-20, 1984, ed. M. A. Bramer. Cambridge: Cambridge University Press, 1986.

[Magu 88] Maguire B. "An incremental approach to expert systems development", in Proceedings of the eighth international workshop on expert systems and their applications, Avignon, May 30-June 3, 1988, by the European Coordinating Committee for Artificial Intelligence and the Cognitive Research Association. France: EC2, 1988, 249-259.

[Olso 87] Olson J. R., H. H. Rueter. "Extracting expertise from experts. Methods for knowledge acquisition", Expert System, 4 (August 1987): 152-167.

[Veso 88] Vesoul P. "A specification and documentation approach to expert system", in Proceedings of the eighth international workshop on expert systems and their applications, Avignon, May 30-June 3, 1988, by the European Coordinating Committee for Artificial Intelligence and the Cognitive Research Association. France: EC2, 1988, 297-316.

KNOWLEDGE ACQUISITION

[Inte 88] _____. "Kate, widening the Knowledge Acquisition bottleneck", Intellisoft. Orsay: Intellisoft press, 1988.

[Abre 87] Abrett G., M. H. Burstein. "The KREME knowledge editing environment", International Journal of Man-Machine Studies, 27 (1987): 103-126.

[Ayel 86] Ayel M., E. Pipard, M-C. Rousset. "Le controle de coherence et la validite des bases de connaissances". [Consistency control and validation of knowledge bases]. Chambery-Orsay: LIA-LRI, 1986.

[Benn 83] Bennett J. S. "ROGET: acquiring the conceptual structure of a diagnostic expert system", in IEEE, Proceedings of the workshop on principles of knowledge-based systems, Denver, December 3-4, 1984, by the IEEE Computer Society. Silver Spring: IEEE Computer Society Press, 1983, 83-88.

[Boos 84] Boose J. H. "Personal construct theory and the transfer of human expertise", in AAAI-84, Proceedings of the National Conference on Artificial Intelligence, Austin, August 6-10, 1984, by the American Association for Artificial Intelligence. Los Altos: W. Kaufmann, 1983, 27-33.

[Boos 87] _____, J. M. Bradshaw. Expertise transfert and complex problems: using AQUINAS as a knowledge acquisition workbench knowledge base systems", International Journal of Man-Machine Studies, 26 (1987): 3-28.

[Buch 84] Buchanan B. G., E. H. Shortliffe. Rule-based expert systems. USA: Addison-Wesley, 1984.

[Buch 81] _____, E. Feigenbaum. "Dendral and Meta-dendral: their application dimensions", in Readings in Artificial Intelligence: a collection of articles, ed. B. L. Webber, N. J. Nillson. Palo-Alto: Tioga, 1981.

[Buch 87] _____, "Some approaches to knowledge acquisition", in Machine learning: a guide to current research, ed. T. M. Mitchell, J. G. Carbonell, R. S. Michalski. Boston: Kleemer Academic, 1987.

[Cohe 87] Cohen P. R., M. Greenberg, J. Delisio. "MU: a development environment for prospective reasoning systems", in AAAI-87, Proceedings of the sixth National Conference on Artificial Intelligence, Seattle, July 13-17, 1987, by the American Association for Artificial Intelligence. Los Altos: M. Kaufmann, 1987, 783-788.

[Crag 87] Cragun B. J. "A decision-table-based processor for checking completeness and consistency in rule-based expert systems", International Journal of Man-Machines Studies, 26 (1987): 633-648.

[Davi 77] Davis R. " Interactive transfer of expertise: acquisition of new inference rules", in IJCAI-77, Proceedings of the fifth International Joint Conference on Artificial Intelligence, Cambridge, August 22-25, 1977, by the International Joint Conference on Artificial Intelligence. Los Altos: W. Kaufmann, 1977, 321-328.

[Died 87] Diederich J., J. Ruhmann, M. May. "Kryton: a knowledge-acquisition tool for expert system", International Journal of Man-Machine Studies, 26 (1987): 29-40.

[Eshe 87] Eshelman L., D. Ehret, J. McDermott, M. Tan. "Mole: a tenacious knowledge-acquisition tool", International Journal of Man-Machine Studies, 26 (1987): 41-54.

[Garg 87] Garg-Janardan, G. Salvendy. "A conceptual framework for knowledge elicitation", International Journal of Man-Machine Studies, 26 (1987): 521-531.

[Grub 87] Gruber T. R., P. R. Cohen. Principles of design for knowledge acquisition", in IEEE, Proceedings of the third conference on Artificial Intelligence application, Kissimmee, February 23-27, 1987, by IEEE Computer Society. Washington: IEEE Computer Society Press, 1987, 9-15.

[Kahn 87] Kahn G. S. "Knowledge acquisition: investigation and general principles",in Machine learning: a guide to current research, ed. T. M. Mitchell, J. G. Carbonell, R. S. Michalski. Boston: Kleemer Academic, 1987, 119-121.

[Kass 87] Kass R., T. Finin. "Rules for the implicit acquisition of knowledge about the user", in AAAI-87, Proceedings of the sixth National Conference on Artificial Intelligence, Seattle, July 13-17, 1987, by the American Association for Artificial Intelligence. Los Alto: M. Kaufmann, 1987, 295-300.

[Kitt 87] Kitto C. M., J. H. Boose. "Choosing knowledge acquisition strategies for application tasks", in IEEE, Proceedings of the Western Conference on Expert Systems, Anaherim, June 2-4, 1987, by the Western committee of the Computer Society of the IEEE. Washington: Computer Society Press of the IEEE, 1987, 96-103.

[Klin 87] Klinker G., J. Bentolila, S. Genetet, M. Grimes, J. McDermott. "KNACK- Report-driven knowledge acquisition", International Journal of Man-Machines Studies, 26 (1987): 65-79.

[Lann 86] Lannuzel P. "Programmes d'apprentissage automatique: finalite et description de quelques algorithmes". [Automated learning programs: goals and description of some algorithms]. France: CERGRENE, note about learning programs, 1986.

[Marc 87] Marcus S. "Taking backtracking with a grain of SALT", International Journal of Man-Machine Studies, 26 (1987): 383-398.

[Mich 83] Michalski R. S., J. G. Carbonell, T. M. Mitchell. Machine learning: an artificial intelligence approach. Los Altos: M. Kaufmann, 1983.

[Mich 86] Michie D., S. Muggleton, C. Riese, S, Zubick. " Rulemaster: a second generation knowledge-engineering facility", in IEEE, Proceedings of the first Conference on Artificial Intelligence Applications, Sheraton, December 5-7, 1984, by IEEE Computer Society. Washington: IEEE Computer Society Press, 1986, 591-597.

[Moor 87] Moore E. A. "INFORM: an architecture for expert-directed knowledge acquisition", International Journal of Man-Machine Studies, 26 (1987): 213-230.

[Mori 87] Morih K. "Acquiring domain models", International Journal of Man-Machine Studies, 26 (1987): 93-104.

[Moze 87] Mozetic I. "Knowledge extraction through learning from examples", in Machine Learning: a guide to current research, ed. T. M. Mitchell, J. G. Carbonell, R. S. Michalski. Boston: Kleemer Academic, 1987, 227-231.

[Muse 87] Musen M. A., L. M. Fagan, D. M. Combo, E. H. Shortliffe. "Use of domain model to drive an interactive knowledge-editing tool", International Journal of Man-Machine Studies, 26 (1987): 105-121.

[O'Ban 87] O'Bannon R,. M. "An intelligent aid to assist knowledge engineers with interviewing experts", in IEEE, Proceedings of the Western Conference on Expert Systems, Anaherim, June 2-4, 1987, by the Western committee of the Computer Society of the IEEE. Washington: Computer Society Press of the IEEE, 1987, 31-35.

[Pazz 86] Pazzani M. J. "Refining the knowledge-base of a diagnostic expert system: an application of failure-driven learning", in AAAI-86, Proceedings of the fifth National Conference on Artificial Intelligence, Philadelphia, August 11-15, 1986, by the American Association for Artificial Intelligence. Los-Alto: M. Kaufmann, 1986. 1029-1035.

[Phil 85] Phillips B., S. L. Messick, M. J. Freiling, J. H. Alexander. "INKA: the *IN*GLISH knowledge acquisition interface for electronic instrument troubleshooting systems", in IEEE, Proceedings of the third conference on Artificial Intelligence Application, the engineering of knowledge-based systems, Miami Beach, December 11-13, 1985, by IEEE Computer Society. Washington: IEEE Computer Society Press, 1986, 676-682.

[Quin 82] Quinlan J. R. "Semi-autonomous acquisition of pattern-based knowledge", in Machine Intelligence 10, ed. J. E. Hayes, D. Michie, L. I. Mikulich . Chichester: Ellis Hierwood Limited, 1982, 159-172.

[Rous 86] Rousset M-C. "Sur la coherence et la validite des bases de connaissances". [About the consistency and the validation of knowledge bases].. France: PRCIA technical report, n°1, 1986.

[Salz 88] Salzberg S. "Machine learning moves out of the lab", Artificial Intelligence expert the magazine for the artificial intelligence community, 3 (February 1988): 41-52.

[Voye 87] Voyer R. "Representation et utilisation des connaissances". [Representation andutilization of knowledge], in Moteurs de systemes experts. France: Eyrolles, 1987, 57-76.

[Walk 87] Walker A., M. McCord, J. F. Sowa, W. G. Wilson. "Checking incoming knowledge", in A logical approach to expert systems and natural language processing, knowledge systems in Prolog. USA: Addison-Wesley, 1987, 275-284.

[Wilk 87] Wilkins D. C., W. J. Clancey, B. G. Buchanan. "Overview of the Odysseus learning apprentice", in Machine Learning: a guide to current research, ed. T. M. Mitchell, J. G. Carbonell, R. S. Michalski. Boston: Kleemer Academic, 1987, 369-373.

[Wins 87] Winston H., R Smith, M. Kleyn, T. M. Mitchell, B. Buchanan.

"Learning apprentice systems research at Schlumberger", in Machine Learning: a guide to current research, ed. T. M. Mitchell, J. G. Carbonell, R. S. Michalski. Boston: Kleemer Academic, 1987, 379-383.

NEXPERT-OBJECT™ REFERENCES

[Neur 87a] "Nexpert callable interface, Mac version, version 1.0". Palo-Alto: Neuron Data, 1987.

[Neur 87b] "Nexpert-Object reference manual (version 1.0)" . Palo-Alto: Neuron Data, 1987.

[Neur 87c] "Nexpert on microVax, IBM-AT, Mac".Palo-Alto: Neuron Data, 1987.

[Neur 88a] "Nexpert callable interface, Mac version, version 1.1". Palo-Alto: Neuron Data, 1988.

[Neur 88b] "Nexpert-Object reference manual (version 1.1)" . Palo-Alto: Neuron Data, 1988.

[Chau 86] Chauvet J-M. "Nexpert: systeme expert de seconde generation". [Nexpert: a second generation expert system]. France: Neuron Data, 1986, synthesis note.

[Rapp 86] Rappaport A. " Le systeme expert de Neuron Data". [The expert system of Neuron Data]. Palo-Alto: Neuron Data, 1986.

[Rapp 87] _____. "Multiple-problem subspaces in the knowledge design process", International Journal opf Man-Machine Studies, 26 (1987): 435-452.

NEXPERT-OBJECT™: APPLICATION METHODOLOGIES

[Neur 87d] "Nexpert-object applications: an in-service expert for configuring thusters on orbiting spacecraft -COMSAT-, intelligent process development of foam molding for the termal protection system of the space shuttle external tank -NASA-". Palo-Alto: Neuron Data, 1987.

[Folk 87] Folkeringa D. "Nexpert-Object: applications". [Nexpert-Object: applications]. France: Eurequip, pratical training report, 1987.

[Lann 87a] Lannuzel P. "Methodologie suivie pour la construction du logiciel Pilote". [Methodology followed during the construction of the expert system Pilote]. France: CERGRENE-CGE, 1987.

[Lann 87b] _____. "Organisation de la maquette Pilote". [Organization of the prototype Pilote]. France: CERGRENE-CGE, 1987.

[Pero 87] Perot F. "Les differentes approaches vis a vis de Nexpert-Object". [The different approaches to Nexpert-Object]. France: Eurequip, pratical training report, 1987.

MACINTOSH REFERENCES

[Appl 86a] "Inside Macintosh Volume I". USA: Addison-Wesley Publishing Company, 1986.

[Appl 86b] "Inside Macintosh Volume II". USA: Addison-Wesley Publishing Company, 1986.

[Appl 87] "Inside Macintosh Volume IV". USA: Addison-Wesley Publishing Company, 1987.

[Thin 87] "LightSpeedC reference manual version 3.11". USA: Think's press, 1987.

[Swan 87] Swan T. "Programming with Macintosh Turbo Pascal".USA: John Wiley and Sons Inc., 1987.

CORNELL UNIVERSITY PROJECT

[Corn 87a] "Sampling forms for STLM, OBLR, ERM, mites, apple maggot (from the simplified IPM program for apple growers in Western New-York)". Geneva: Cornell, 1987.

[Corn 87b] "1987 Cornell chemical recommendations for Commercial Tree-fruit production. Geneva: Cornell, 1987.

APPENDIX 1: MORE INFORMATION ABOUT NEXPERT™

Knowledge representation

Unique rule format (for backward and forward chaining)

Knowledge islands and contexts

Multi-hierarchical object structure: class, object, property, methods

Multi-inheritance: user-defined or default inheritance stgrategies

Procedure attachment to slots: if_needed, if_changed

Inference mechanisms

Integrated forward/backward chaining

Automatic goal generation

Possible event-driven reasoning

Non-monotony reasoning and revisions

No predefined uncertainty mechanism but predefined incompleteness strategy

User-defined or default inference strategies and contexts

Several levels of pattern-matchings through classes/objects

User-friendly interface

User friendly graphic window environment

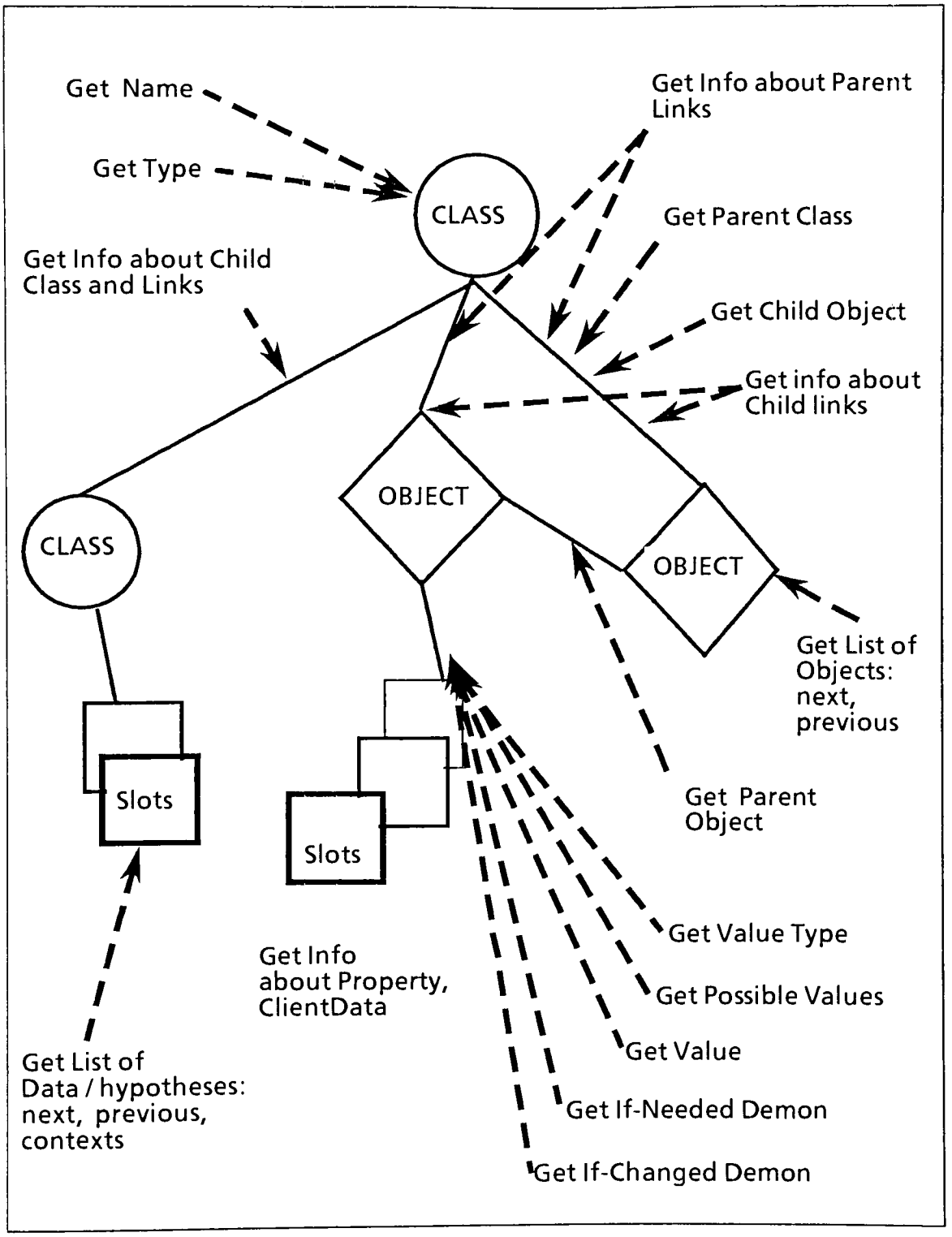
All information available at any moment (rules, objects, classes, properties, methods, report generator supports)

Dynamic networks (rules and objects) available also during execution for an interactive design with the editors

Integration

Direct access to relational databases, spreadsheets

Library of calls, which gives complete access to the inference and representation mechanisms as well as the run-time or execution interface.



Information about the object structure that an extern program can obtain via Nexpert callable interface
(from Neuron Data)

APPENDIX 2: CORNELL APPLICATION, ANALYSIS AND KNOWLEDGE BASES

This application has been developed at Cornell University, and I have followed it since December, 1987.

I will briefly present in a first part the project and its different constraints (time , computer, and objectives). In the second part I will report different issues encountered during the knowledge acquisition phase of this project recently called EASY-CAPS (Expert Advisory System for Managing Apple Cropping Systems). Three dimensions will guide my assessment of this process: session organization, expert system approach, and interview techniques.

1. Description of the project and the application domain

The domain involved in this application is relevant to enthomology and agriculture areas, but this domain knowledge is not yet well-defined, since parts of this domain are still in the research field. In fact, this expert system corresponds to an enhancement of a computer project called IPM (developed for aiding the growers in the management of their apple-tree orchard), which was implemented only last year. This project consists of building a knowledge-base for recommending pesticides to be used at different phenological periods (from half-inch-green to summer) on apple-trees.

The global semantic architecture of the expert system is the following: the final user (the grower) logs in on the system (if possible) at the various critical periods. He enters the data needed for the system to determine the critical state of the apple-tree orchard. He may also be asked to carry out a simplified sampling procedure in order to determine the level of various pest populations in his orchard. Corresponding explanations/instructions are provided to help him carrying out these sampling procedures. These results can then be interpreted by the system to give the user recommendations for applying pesticide. Each such recommendation will be accompanied by an explanation. An emphasis will be put on the interface since the expert system has an educating goal in the sense that the grower will be trained to use pesticides as little as possible.

For the design, four experts are available once a week for a meeting. And one knowledge designer accompanied by a knowledge designer is in charge of the knowledge-base development, to be completed by April 1988 for the first periods of advice.

Finally, the expert system must run on small systems (Macintosh or IBM-PC) so that the final user can easily access the system whenever needed.

1.1 Description of the problem

Global Approach

The problem addressed in this application belongs to the classification analysis problem class (according to Clancey's classification, 1985); and it includes both a diagnostic stage, to determine in which state the orchard is, and an advice stage, to specify the actions to be done in the studied orchard at a specific period.

In the first stage the expert tries to determine the critical threshold the studied orchard has reached, based on descriptive data provided by the user such as cultivar, past history, or phenological stage of the block. The connection between potential "symptoms" and critical threshold of the orchard is mainly done by rules of thumb. For example, if it is summer and the grower did not spray the orchard in the previous periods with Lannate, Vydate or a synthetic pyrethroid, it is likely that no mite predator will be found in the next pest sample. We can even say that the rules used are more or less empirical since the knowledge is still compiled and the main goal of this application is to provide the user with an operational system more than a purely educating system. An attempt at the start of the project has been done to "decompile" the application knowledge, but the time factor and the difficulty of the task -the domain is still in the research field- have forced the developer team to postpone such a task. Consequently, only a series of predicable situations can lead to a diagnostic, and a set of default advice has been set up to deal with novel situations. Among them, we find the advice to login later in order to provide the system with more data when the user has logged at a period for which the state of the research cannot handle the situation.

In the second stage of a session, the expert matches the identified situation with a set of spray procedures and other advice. Here again, the choice of a sequence of actions does not rely on a domain model.

Problem Characteristics and Sources of Knowledge

We have seen so far that the expert mainly uses heuristics to guide his reasoning whatever the situation. The beginning of the reasoning seems to be the same all the time.

The expert queries the grower to find the exact phenological state of the orchard, that is, half-inch-green, tigh-cluster or early-pink, pink, bloom, petal-fall, or summer. Given this information, he can orient his reasoning, since each period requires a different type of reasoning. Then, depending on the period, he concentrates on the treatment against specific bugs (usually, ERM, STLM, OBLR, RAA, and TPB): for this, different types of data are necessary, but none can be deduced from other data, and they are asked to the grower (such as for sprays that has been used) or retrieved from a data base (such as for the block cultivar), as the reasoning goes along. If the grower does not know what to answer, the expert uses a set of

default values. For example, when the expert queries the grower if a specific bug, let say OBLR, has been a concern in the past, and if the grower does not remember the real impact of this bug in the past, the expert automatically interprets this as no concern in the past. Thus, even if at the start the reasoning is forward chaining, the expert quickly uses a means-end-analysis. The reasoning also relies on predefined data.

However, only one-line-reasoning is used to solve the problem. By this, I mean that the expert can match a real situation with only one known case using only one direct reasoning path. He does not need to combine and weigh hypotheses to determine to which case the actual orchard situation can be attached. This match is always possible since only a limited set of options for each datum is possible. If the grower proposes a value for which the expert has encountered no previous case, the expert associates this datum with a comparable value with which he can pursue his reasoning. A counter-part of this technique is to provide a mis-diagnostic, but this is the method used by the expert.

Having identified the situation, the expert advises the grower about the possible actions to be done. Several pesticides, if needed, and/or sampling procedures (i.e. solutions) are given to the grower, but, the final decision whether to follow the recommendations still remains the grower's responsability, and mainly for this reason several pesticides and not only one with the same action are proposed.

Input Characteristics: origin, reliability

As we have already seen in the description of the problem characteristics, the reasoning results mainly rely on different data -depending on the period entered by the grower. An assumption has been made, here, since no previous expert systems have been tried in this area: the final user is assumed to be "cooperative and knowledgeable". It is not an absurd assumption, since for some questions -such as for the cultivar- the system will never be able to check if the answer is correct or erroneous.

However, the system must provide sufficient information about the required answer. For example, when the phenological state of the orchard is queried, the grower must be able to consult charts showing the different criteria of identification. In fact, it is usually possible to anticipate the uncertainty and the unreliability of data, since the expert is often in contact with growers and knows their reactions to questions similar to those the system will ask. We have already seen how the expert deals with unknown answers (use default values) and with not-yet known answer (advice differred).

A last feature of the input data concerns their validity through time. The expert refers in his reasoning to previous actions on the orchard and recommends some other actions. But, from one consultation to another, the grower may have not followed the

recommendations. The expert must then ask about the last spray or action done on the orchard to be sure to deal with trustworth data.

Solution Characteristics

Whatever the situation, the expert must provide the grower with an answer as quickly as possible even though the expert has not necessarily encountered a similar situation in the past. Consequently, it is possiblethat he proposes a default set of procedures in order to obtain more information (such as sampling) or wait to a more advanced stage of the orchard to be able to give a solution. This is mainly because the expert does not deal with an entirely known solution space.

Nevertheless, when the expert can provide the grower with a complete piece of advice, he is often obliged to propose several solutions, that is, several pesticides. With only one solution, the grower may not use it, especially when he does not know this pesticide or does not like it (classic psychological phenomenon).

1.2 Knowledge representation

The application thus refers to a case-matching process relying on non-hierarchical, sometimes incomplete data and involving different steps (situation identification, bugged orchard state analysis for each bug knowing that some pesticides interfer) .

Static and Dynamics Knowledge

Since the data do not present a hierarchical structure and a great amount of cases are possible, frames or objects alone do not seem to fit to this application; and the potential uncertainty on some data rejects the solution of a logic programming system. However, it is possible to identify entities defined by a set of characteristics: the block and its physical characteristics, the recommendations and actions done by the grower for each period. The best solution then seems to be a mix of *rules and objects*, where the rules can be used to control the reasoning (rules of thumb) or guide the reasoning, and the objects mainly for matching with default cases. This allows the system to present a clear division between the outline of the reasoning and particular case values. It is also possible to use only rules, but the maintenance of the knowledge base could become obscure with a rule-based knowledge base; and since the only available domain knowledge is not complete, maintenance is a crucial issue.

Control Knowledge

As we have seen in the previous section, the reasoning is mainly means-end-analysis, but not necessary event-driven. A *forward and backward* chaining system is required, but a *non-monotonic* system can allow the grower to review his answers and change his mind in view of the recommendations proposed by the system.

A last feature required by the application is a knowledge representation that allows a *division of the reasoning into subparts*: a global structure for the system can be viewed as a control module initializing the session and identifying the type of situation (and consequently, the type of reasoning) surrounded by a set of small knowledge bases solving a particular period (i.e. containing a particular type of reasoning). This division in different knowledge bases or modules is a way to represent the change of focus of attention according to the identified situation.

Interface With the rest of the World

The main constraint relying on data is that there is a fixed number of data, but depending on the problem only some of them will be queried. Consequently, the system needs to be able to store certain data characteristics of one given block and to retrieve part of them. It must also be able to advise the grower for several blocks. The system must them have a powerful *interface with a database*.

The interface with external programs is not the only required interface. The user interface is also fundamental, since the application has a secondary educational objective. Also, since the data are the basis of the reasoning, the system must be able to control the input by justifying each question and providing all the necessary information relevant to the question. In the display of solutions, the system must also provide the grower with a set of explanations: different levels of explanations can be also viewed. The system must provide *graphics output, with different levels of help facilities.*

1.3. Design decisions and expert system architecture for this project

About the Expert System Shell

The expert system shell that has been chosen for this project is Nexpert-Object™, running on Macintosh SE. This choice has been made due to the restriction concerning the type of computer available to the final user; in the marketplace Nexpert™ was nearly the only shell possessing rules and objects that runs on a Macintosh. However, this shell was chosen before the project has been completely defined; among issues which were not elicited are the user objectives, the different types of problems the system will be able to solve, and the different levels of abstraction the system will address. The principal and necessary

components of the project design were determined along with the expertise elicitation and the construction of a prototype. The necessity to store data from previous sessions is among those components. It is for these reasons that the search for an expert system shell was oriented toward a shell presenting various schemes of knowledge representation (i.e. rules, object structure, method, open-architecture, attractive display functions, etc.).

Once the choice of the expert system has been made, the expert system design took advantages of several features available with this shell such as hypercard-like graphics display, or the possibility of loading a knowledge base in the middle of a session.

About Knowledge Representation Issues

The object structure has been used to represent the knowledge about the block to be analyzed: we distinguish the object block containing all its physical characteristics (cultivars, age, identification) and the objects block histories containing all relevant data concerning its bug history and concerning the sprays used in previous periods. The hierarchical structure of the objects has not been used since there is no hierarchy involved in this application, and methods are used for default value passing. Finally, rules are used to control the block state analysis. They mainly contain case-identification knowledge.

The general organization of the "knowledge-base" is the following: a main knowledge base, which identifies which phenological period and which block is concerned - relevant data are eventually retrieved from a data base and the corresponding object created dynamically since the expert system can analyze several blocks - and a knowledge base per period which contains the reasoning concerning this particular period. These knowledge bases are loaded by the main knowledge-base when needed.

Backward chaining is used in all knowledge bases. But this choice can be highly argued and a set of "knowledge islands" linked together by context could have been implemented to represent changes in focus of attention (refer to the different steps in block analysis). The automatic goal generation facility of the shell could also have been used to switch from one case to another one, instead of using explicit rules for the same purpose. Finally, non-monotinicity will be used to allow the user to change his mind about datum values he entered with regard to the recommendations made by the system.

A last knowledge representation scheme has been chosen to present all necessary knowledge for pesticides or sampling recommendations: the graphics facility available in Nexpert™, called AIScreen; AIScreen is a interactive and graphical visualization of information in Nexpert™ (version Macintosh). It consists of pictures and named frames. It can also contain fields pointing to other hyperimages in a fully recursively way.

1.4 Knowledge acuqisition process

For the design and the knowledge extraction, four experts were available once a week for a meeting (1 to 3 hour long) from December to March. Among them, we can identify one "main" expert, whose advice prevails: he is also a grower which increases the weight of his opinion on the user behaviors. The three other experts are enthomologists and are mainly concerned with the IPM program, which is the knowledge basis of this expert system application. Among these three researchers, we find the project coordinator, whose role is to ensure that the different phases of the project go on without any problem. He is also in charge of organizing the meetings. One knowledge designer accompanied by a knowledge designer is in charge of the knowledge-base development, to be completed by April, 1988, for the first periods of advice. The knowledge designer must design the interface, while the knowledge engineer designs the knowledge base itself. I was personally concerned with the general development of the project and the knowledge representation available in the expert system shell used for this project. The four experts were available during the first two months and are really involved in the project.

Session Organization and interview techniques

The first sessions were used to define the general structure of the expert system and the organization of the project. Since no deep analysis of the project had been previously done, definition of the project scope and goals, expected specifications of the final expert system, type of final users, and validation of the knowledge base were the main concerns of these sessions. This has since been useful but incompletely achieved; numerous modifications of the project goals have been made lately. And these modifications were not related to obstacles met during the design process, but to extraneous factors such as the workshop about expert systems in agriculture where similar projects were presented or research concerns of the moment, which has introduced new features discarded later.

I think the lack of previous analysis of the project is not the only cause of this problem; the fact that the project coordinator is himself an expert leads to this kind of trouble. Fortunately, the structure of the knowledge base was flexible enough to allow these modifications without being obliged to start everything again from scratch. However, it brings useless strains on the solitary knowledge engineer working on this project. Thus, the same questions asked over and over at each session until some commitments were made by the knowledge engineer.

Another factor has influenced the progress of the knowledge acquisition process: the structure of the "knowledge" team. The assignment of the tasks of interface design and knowledge base design to different persons has interfered with the knowledge extraction,

which shouldn't happen. The experts were sometimes disorientated by questions about the interface, in which they have no concerns. I think those questions should involved only the knowledge designer and the project coordinator and not the experts. One goal of the early sessions was to obtain enough knowledge to have a knowledge base and to focus sessions on the expert system interface so that the knowledge designer can really begin his job.

Each session was recorded, which made it for the knowledge engineer easier to review the knowledge after part of the session if he had forgotten something. He also won't try during a session to take as many notes. However, a drawback is that the knowledge engineer tends to rely more on the tape, and in our case, since the experts spoke without any intervention from the knowledge engineer, some sessions were just a soliloquy with questions postponed to the next session. Maybe meetings on Friday afternoons are not a good idea!

A last influence which has a common source with the factor described previously, is the impact of the expert system shell formalism on the formulation of the questions. A hybrid expert system shell was chosen before the start of the project, but the project coordinator in some way has influenced the experts to formalize their knowledge by IF-THEN-ELSE statements. However, they have chosen a good expert system shell for the application, since the knowledge acquisition process was never disturbed by a lack of flexibility of the expert system shell formalism (rule, object, data base links, graphics user interface, etc). Since the formulation of the application knowledge is part of the knowledge acquisition project, it is important that the knowledge engineer employs an expert system shell with enough power so that the knowledge extraction won't be influenced by the limits of a specific AI tool and can be guided by the expert's reasoning.

Expert System Approach

One main mistake was made at the beginning of the project. The coordination of the session was done in such a way that the experts didn't know what to explain. They went on explaining domain knowledge during the first session: it was most like sessions to train the knowledge designer and knowledge engineer in the application domain, and since the experts spoke so willingly, it was difficult to stop them and to change the focus of discussion of such meetings. In one way, it was useful, since no previous paper has been written for non-specialist readers on this specific knowledge: the book-knowledge doesn't exist here.

The lack of specified project goals and the unbalanced team (more experts than knowledge engineers) have led to a difficult start and to some strange misunderstandings. In fact, the knowledge engineers didn't follow the experts'reasoning; they were too concerned by the structure of the knowledge base. One thing a knowledge engineer shouldn't do during the knowledge extracting phase is to explain his AI concerns about the application. He should

focus his attention on extracting the expert's reasoning, by this I mean trying to identify the main dimension on which the expert based his reasoning, and to find out what are the subtasks or steps of the problem with regard to the goals of the project. In this application, the main factor was the TIME; the different subtasks were identification of the situation, analysis of the situation and identification of the recommendations to give to the grower. These major data were identified only during the third session. Using specific cases would have helped to identify these major chunks of knowledge, but they were never used during any session.

Consequently, the experts have expressed their knowledge via decision tables to be exhaustive, but when, in later sessions, more precise questions about specific cases were asked, the experts changed their way of answering and willingly gave the necessary justifications for each answer using expressions such as "because", "since", "depending on" and so on.

A last remark concerns the vocabulary used at the beginning which led later to some confusion. Since terms such as half-inch green, pink and bloom didn't mean a lot to the knowledge engineers, they asked the experts to use calendar periods to help them. Later, in other sessions they used both terminologies and finally the experts'terminology has prevailed. It would have been easier to learn the correct terms from the beginning .

2. Problems encountered when learning how to use Nexpert *Object*: Cornell application

In this section, I will give an overview of knowledge representation issues encountered with Nexpert *Object* .

2.1. General architecture of Cornell application

- Subdivision of the knowledge between knowledge islands and knowledge bases

One characteristic of Cornell application was that it is decomposable into a set of reasonings for each period of the year. At the beginning of the project, the knowledge engineer was tempted to group the global reasoning into one knowledge base, in spite of the obvious possible decomposition of the problem into smaller independent problems. But, the idea of assigning knowledge to different knowledge bases has been quickly adopted: it is far much simplier to test each individual knowledge base than the global application given its complexity. Consequently, the following architecture of the application has been built: a

general module to identify what kind of problem the system is dealing with, and a set of modules to solve the identified problem.

Another aspect of the architecture of Cornell application is the possibility to subdivide each problem into a sequence of related steps. We could have chosen to keep subdividing into smaller knowledge bases, but the knowledge bases would have lost their semantic meaning. Since a knowledge base can be viewed as a reasoning which allows the system to solve a specific problem, it won't mean anything to map a step with a knowledge base. A step in a reasoning corresponds to a change of focus of attention, and Nexpert *Object* allows the knowledge engineer to represent such changes in attention by clustering rules (i.e. creating knowledge islands). This is however possible if there is no connection between these steps. The general module shows an example of the use of knowledge islands (see Figure 1): the system has to identify the block to be studied, find its characteristics (by asking the user or retrieving from a database) and then switches to the "real" reasoning which will allow the system to solve the problem. Here, each stage can be viewed as a knowledge island.

- Multiple goal versus one-goal: how to use knowledge to control the focus of attention of the engine: influence of programming background

As the diagram of the Figure 1 shows, the general structure of rules, chosen by the knowledge engineer was mainly sequential (like a program) with one final goal: solve the problem at a given period. The problem with this kind of structure relies on the difficulty to see the influence of one step on another step. A better approach using Nexpert *Object* facilities in knowledge representation will be to isolate the different independent steps using knowledge islands and to use the automatic goal generation associated with categories to allow the system to focus on the next step connected with the new conclusions .

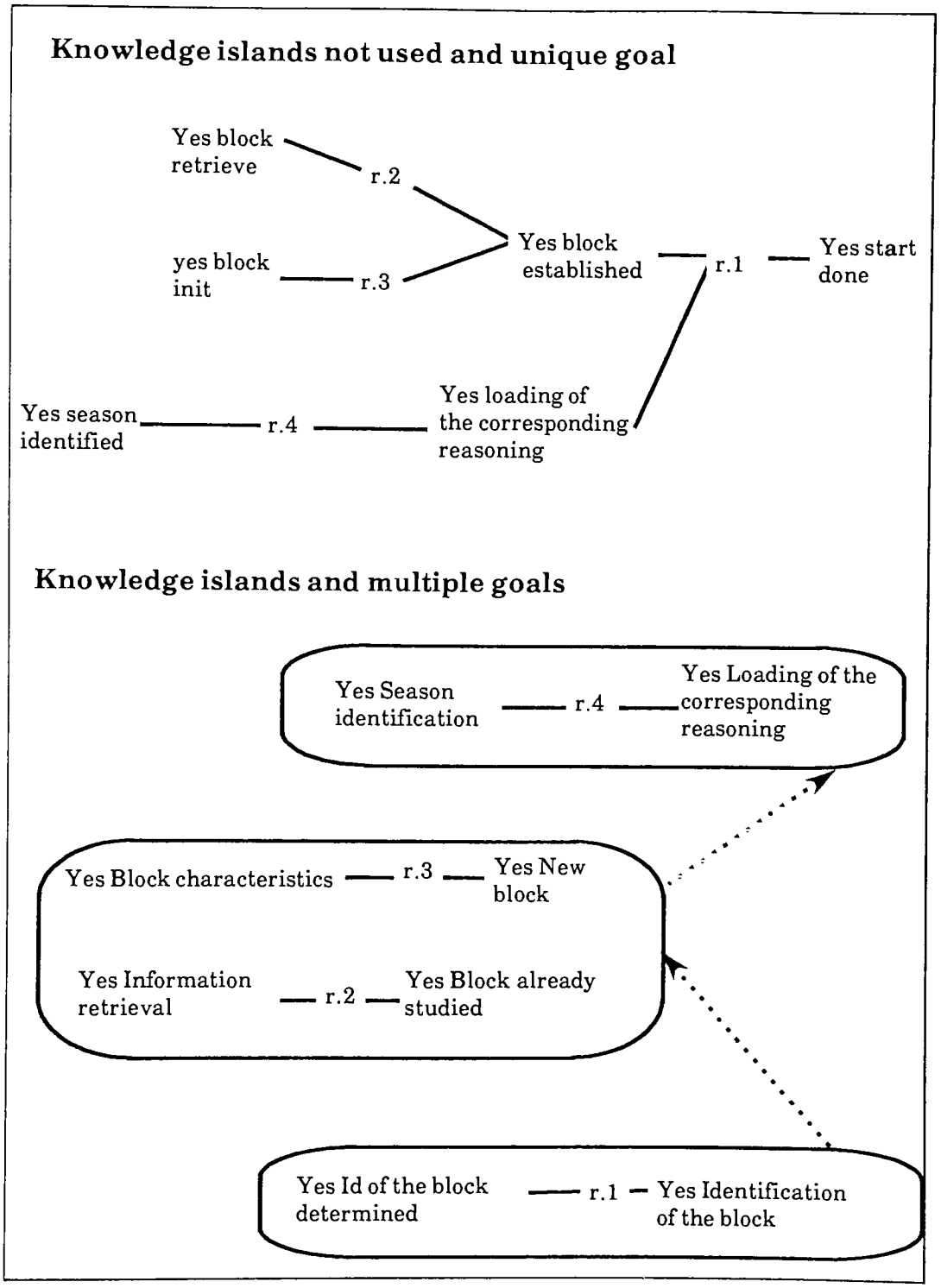


Figure 1. Knowledge islands - Multiple goals versus unique goal

Handling several entry points to a knowledge base (control knowledge)

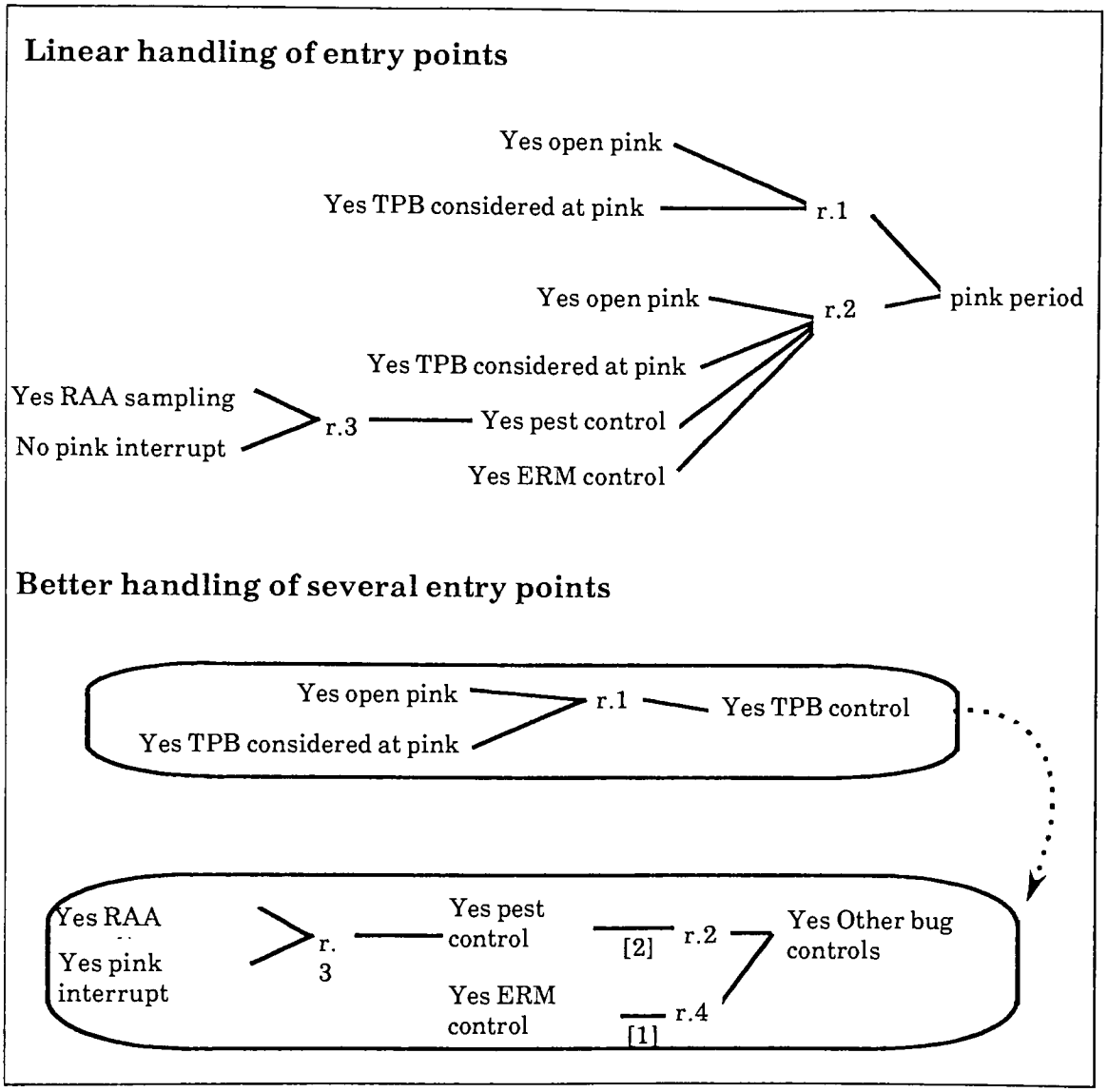


Figure 2. Handling several entry points to a knowledge base

It was one of the main problems in this application. At a certain point of the reasoning, the system may ask the user for the results of a specific sampling. If the user has already done the sampling, he can pursue the session; otherwise, the system has to save the actual state of the reasoning and requires the user to log in later when he has the necessary information for the system to propose a solution to the problem. The knowledge engineer then needs to set up two entry points to the knowledge base. This can be achieved by using a multiple goal

structure for the rules (refer for an example to the Figure 2) and a variable which will fire the right group of rules (in our case: pink interrupt).

- **Use of database: how much to save/retrieve, how to split data over different files: database as a solution to handle the factor time.**

The previous remark on handling several entry points in a knowledge base has introduced the problem of the connection of a knowledge base with a data base. Data have been queried through the expert system interface and split over the object structure. The knowledge engineer needs to reconstruct this structure in the data base using different files if necessary: it is a classical problem of data base (see the example of the Figure 3., where the object

File for static data, that is characteristics of the block: attributes

Block number	cultivar1	cultivar2	cultivar3	disposition

File for "dynamic" data, that is sprays used or to be used: attributes

Block number	ERM spray for period 1	ERM spray for period 2	....	OBLR spray for period 4

Figure 3. Database information

structure was an object block associated with the properties number, cultivar1, cultivar2, disposition, and another object related to the block object and associated to with the properties about the recommendations done for each period and against each bug).

3.2 Vocabulary: readability of the code

- Use of meaningless names for data, objects, properties or hypotheses

Let us take the following example: the object "on" associated to the property "dummy", slot which represents a dummy variable. It might be difficult 3 or 4 months later to remember the role played by this dummy variable. The updatings might then be all the more difficult if this variable has an important role in the way rules are fired.

For the same reasons, the use of abbreviations for datum names, object names, property or hypothesis names creates troubles. The following example is explicitly by itself:

OBLR__r__1 which stands for "recommendation against OBLR at half inch green".

- Keeping useless objects/properties/classes/rules

Since the system doesn't clean the knowledge base if an object, a property, or a rule is no more used, it is the responsibility of the knowledge engineer to throw away useless information. It is also on a maintenance purpose that the knowledge engineer must keep only useful rules and objects in the knowledge base. Useless rules can lead to unexpected results, since the system can fire rules connected to a group of useful rules.

2.3. Rules: representational mismatches

- Handling a Negative condition in a rule

Handling a negative boolean statement in a rule is difficult, since several paths can falsify this condition. Such a rule structure is used for exhaustivity, but it can be difficult to test even with explicit names. A good solution is to shorten as much as possible the path going from the variables to the hypothesis used in a negative statement.

- Handling several rules leading directly to the same hypothesis

Among rules leading to the same hypothesis, the system will fire first, by default, the earliest created rule. Using this particular behavior of the system can be really dangerous, when the

order of evaluation of the rules is important in the application. If during the updating of the knowledge base a new rule leading to the same hypothesis is created, it can disturb the reasoning. A solution is to keep track of the desired order of evaluation of the rule using categories on the first condition of each rule. It avoids bad surprise later.

- **Local strategy versus global strategy**

In Cornell application, the non-monotonocity was not a required feature for the expert system shell. Since Nexpert Object fires rules, by default, using non-monotonocity, the knowledge engineer should be careful to turn off this facility. The problem is to determine when to be aware of such a problem. If the control strategy is done along with the construction of the knowledg base, there is a risk to have a distributed strategy instead of a coherent global strategy which facilitates the understanding of the system inferences.

How to split knowledge into rules

Examples of errors made during the construction of rules are creating a rule only to execute an action, creating a chain of 1-boolean condition rules, splitting a chunk of knowledge over a set of nested rules or overloading rules (rules with more than 6 conditions, see the Figure 4). These examples show a misunderstanding of the concept rule. A rule must be an entity by itself and carries a meaning. In the example of the Figure 5, the hypothesis rec__apply__oil is in fact the results of the control__ERM; rules r.0 and r.1 can then be collapsed together with the label "initial control of ERM". The rule r.2 corresponds to an alternative solution to the initial control of ERM.

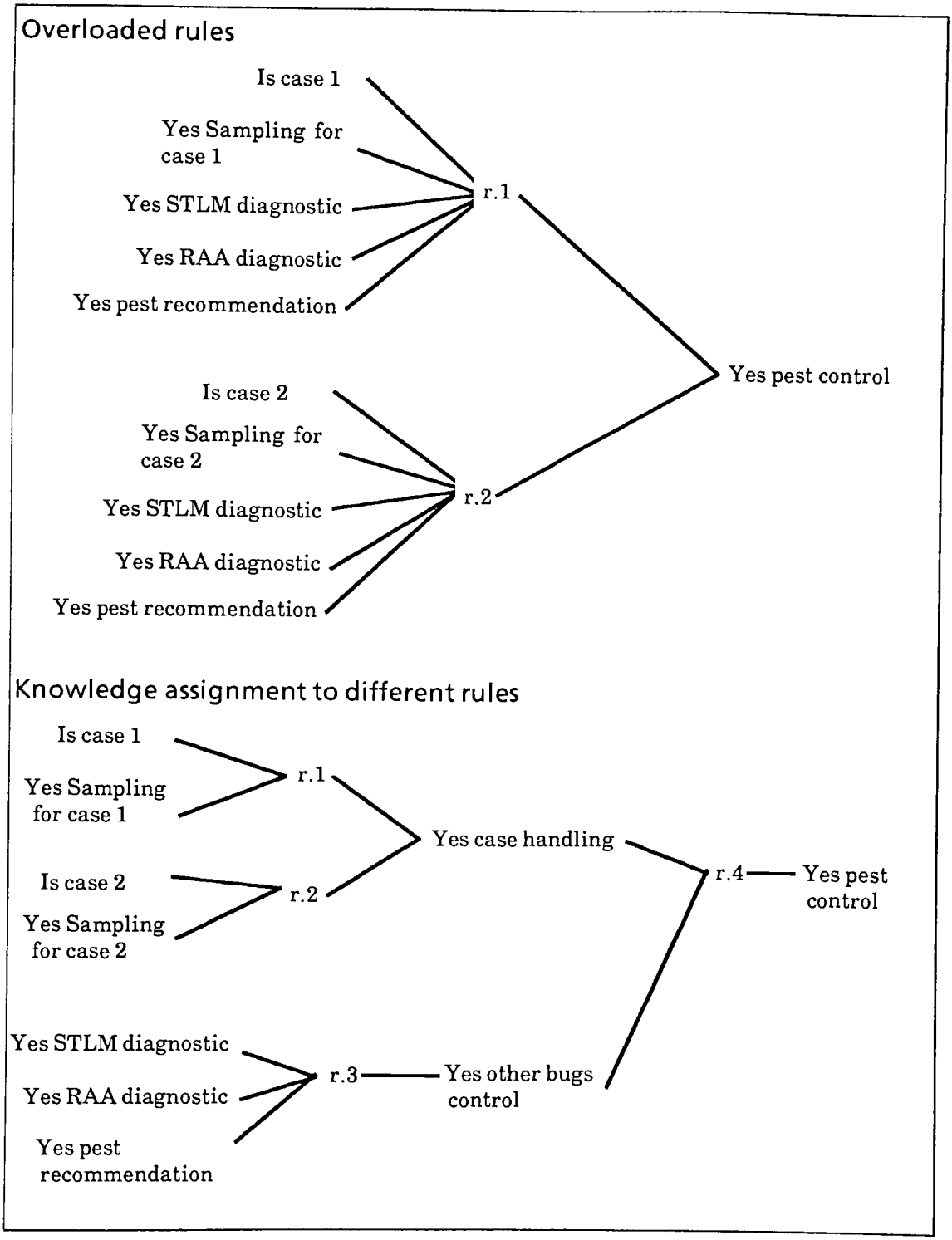


Figure 4. Overload rules and knowledge assignment to rules

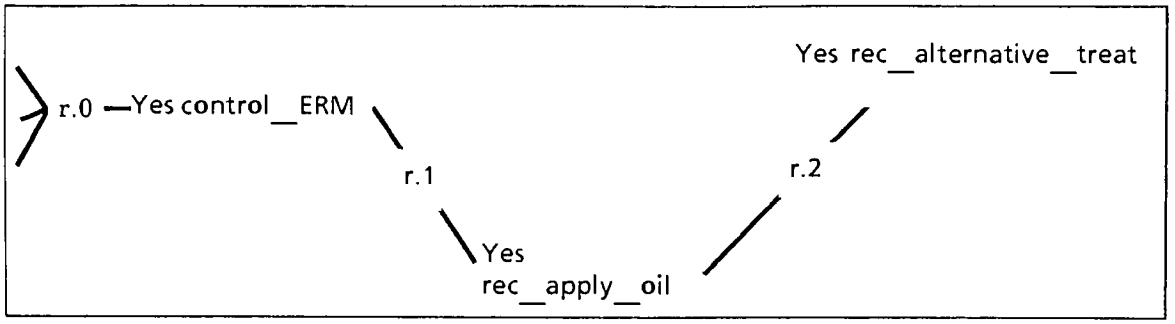


Figure 5. Assigning knowledge to rules

3.4. Objects: representational mismatches

- Bad use of objects

A typical representational mismatch occurred at the first stages of the project: the knowledge engineer was using object and property to represent a simple variable, that is an object with only one property. For example, the object this_run associated with the property time in the condition "Is this_run.time first?". In this case, the property time doesn't characterize the object this_run; the object this_run itself doesn't mean anything. In fact, the knowledge engineer just wants to know whether it is the first session. It will be better to create a simple variable called session_run, to be used in the condition "Is the session_run first?". The object should have an existence by itself; the property is just a characteristic of the object and can have a meaning only when associated to an object. Different types of links can connect a property to an object: structural or static links such as the physical characteristics of a block, or dynamic links such as a recommendation against a bug at a given period.

- Not use of objects

As seen in the previous note, the knowledge engineer is tempted to use objects since it is an available feature of the expert system shell. The other tend is the lack of structure among the objects or to build a complex object structure without any use in the rules. This is maybe a

consequence of the fact that Nexpert automatically creates an object with a default property Value from a simple variable.

- Use of two names for the same entities

Another example of representational mismatch is the use of two variables for the same semantic content. This results from the mixing of the content of a variable and the variable itself: for example, the variables "no_OBLR_sampling" and "OBLR_sampling".

2.5. Explanatory knowledge

- How much to explain?

Different levels of explanations are always possible. It depends how well the user objectives of the expert system have been defined. In this sense, the amount of explanations will vary if the system helps a naive user or a well-trained user. However, this has a great influence on the manner the knowledge base will be structured.

- Different possible orientation for the explanation: enthomology, economy

Which one to choose? It is mainly a problem of defining the goals of the project.

We have reviewed a set of typical errors knowledge engineers do when they use for the first time Nexpert Object. However, these errors are not all shell-dependent such as overloading rules or creating a useless object structure.

APPENDIX 3

IMPLEMENTATION CODE

Name of the function	Role of the function
Kernel.c (analysis, level 1):	
Main	Controls the DA.
DRVROpen	Controls the opening of the DA.
DRVRClose	Controls the closing of the DA.
DRVRPrime	Not used in this DA.
DRVRStatus	Not used in this DA.
DRVRControl	Controls system events sent to the DA by Nexpert (keyboard, mouse, autokey, window activation, window updating).
doCtlEvent	Execute the different actions in response to a mouse event; controls the buttons (Novice/Expert, Stop/Continue, Help, Compilation)
launchModule	Fake the detection of the end of compilation of an atom and launch the required analysis (levels 2 and 3).
MyNotify	Execute the different actions in response to the notification of an event specific to Nexpert (object Update/Create) and launch the analysis (level 1) if necessary.
Voc.c (analysis, level 1):	
CheckObj	Check the validity of the vocabulary.
KiModule.c (analysis, level 2):	
KIModule	Analysis of the created knowledge islands.
GateSearch	Search for a non specific information.
KISearch	Search for the creation of a new knowledge island.
KIStat	Evaluate the structure of knowledge islands.
GateParser	Parser to find a non specific information.
myParser	Parser to find a common object between two conditions.
ExtractName	Extract the name of an object from a condition/action.
Struct.c (analysis, level2):	
StructAnalysis	Analyze the global structure of objects.
NumbList	Compute the number of elements in a given agenda (rules, objects, classes,...).
NumbLinkObject	Compute the number of object links.

Name of the function	Role of the function
RuleModule.c (analysis, level 3):	
ruleModule	Analyze the last rule created.
linearStructAnalysis	Compute the depth of the rule structure.
depthStructure	Compute the width of the rule structure.
ObjectNeed	Search if the last created rule has at least 3 conditions in common with another rule.
CommonNeed	Search if there are at least 3 rules with at least 3 identical conditions.
noSearch	Search for a negative condition in the last rule created.
FlagTechNeed	Search if the "flag technique" can be used.
rule_comp	Compare two rules for at least 3 conditions.
NextRule	Search for the next rule with a given hypothesis.
ObjModule.c (analysis, level 3):	
ObjModule	Analyze the last created object.
otherParentSearch	Compute the depth of the object structure with for root the last created object.
Tools.c:	
GetExtInfo	Get the address of the "working memory" of Nexpert.
max	Compute the maximum of two integers.
InRange2	Compute if a given integer is between two other integers.
PushButton	Controls the display of radio buttons.
okDrawProc	Outline the ok button.
DisplayDlg	Display the dialog box for warning.
DisplayQues	Display the dialog box for questions about rules.
DisplayQuesHypo	Display the dialog box for questions about new hypotheses.
MyStrcat	Concatene two pascal strings.
mystrcmp	Compare two Pascal strings.
CToPasStr	Translate a C string into a Pascal string.
PasToCStr	Translate a Pascal string into a C string.
CStrcpy	Copy of C strings.
CStrcat	Concatene Cstrings.

WARNINGS

Criteria	Warning
Knowledge engineering point of view (analysis, level 1):	
Start the kb design by rules	Nexpert is a hybrid system. If you have chosen to start the kb design with rules, you might also think about a potential object structure.
Start the kb design by objects	Nexpert is a hybrid system. If you have chosen to start the kb design with objects, you might also think about their connection with the rules.
Use of SHOW in rule and number of rules	It is a bit early to think about the user interface.
Use of EXECUTE in rule and number of rules	It is a bit early to think about interfaces.
Number of rules created	It is time to test the kb. It is easier to test a small set of rules.
End of the program	You'd better start to write the documentation of the project before it's too late.
Length of the atom name	The object ** has a too short name. A meaningful name helps a lot for the maintenance. The object ** has a too long name. A meaningful name helps a lot for the maintenance.
"Semantics" of the atom name	The name ** is meaningless. A meaningful name helps a lot for the maintenance.
Knowledge representation point of view (analysis, level 2):	
Number of rules / kis	There are too little kis. You might think to break down your kb into several kbs. It will ease the maintenance. There are too many kis. You might think to group some rules together.
Number of hypotheses/ context	Some kis are not connected. There is a chain of kis. Does the expert solve the problem sequentially?
Number of rules with the hypo and at least 3 identical conditions	There is a lack of objects. You might use objects in these rules.

Criteria	Warning
Knowledge representation point of view (analysis, level 3):	
Depth of the rule structure	There is linear rule structure. Does the expert solve the problem sequentially?
Width of the rule structure	There is a deep rule structure.
Number of occurences / rules	The object ** is a non specific datum. You might try to isolate it.
Number of rules with at least 3 identical conditions	There is at least 3 rules with at least 3 identical conditions. You might use another level of rules.
Condition	The rule no ** has a no in condition. You must be aware it is difficult to test such negative conditions.
Number of conditions	The rule no ** has too many conditions. You might split knowledge over several rules.
	The rule no ** might use the flag technique.
Number of actions	The rule no ** has too many actions. You might split knowledge over several rules.
Knowledge representation point of view (analysis, level 3):	
Number of links / object	There is no object structure. You might think about a more structured organization of objects.
Number of objects / class	There is a lack of classes. You might group some objects into concepts.
Number of levels in the object structure	The object ** has a deep structure.
Number of properties / object or / class	The object ** has too many properties. Are they all representative behaviors?
Number of parents / object	The object ** has too many parents. Are they really distinct concepts?
Number of children / object or / class	The object/class ** has too many children. You might use another level in the object structure.

APPENDIX 4: GLOSSARY

AIScreen, hyperimage (Nexpert *Object*).

Interactive and graphical visualization of information in Nexpert(version MAcintosh). It consists of pictures and named frames. It can also contain fields pointing to other hyperimage in a fully recursively way.

Anticipatory knowledge (MOLE).

Additional information that the presence of event E2 tends to rule out event E1, given the covering knowledge that E1 explains E2.

Circumstancial knowledge (MOLE).

Knowledge which associates evidence with hypotheses, but the evidence does not have to be explained or covered.

Combining knowledge (MOLE).

Information which allows to pick the best combination of viable hypotheses that will explain all of the symptoms.

Conceptual structure (ROGET).

Particular problem-solving tasks and their related set of abstract categories for subgoals and evidence which form a skeletal design for an expert system.

Conclusion items (ETS, AQUINAS).

Traits from qhich the system will determine differences and similarities (with the help of the expert).

Covering knowledge (MOLE).

Hypotheses which explain or cover a set of given symptoms.

Cross-compilation (Nexpert *Object*).

The notion according to which any structure mentioned in the description of another will be automatically created when the latter is compiled.

Default knowledge.

Knowledge which applies when no information is available.

Differentiating knowledge (MOLE).

Knowledge information which differentiate among the hypotheses covering any symptoms.

Direct interviewing methods.

Methods which ask the expert to report on the general model he/she can articulate.

Domain model.

Structural characterization of the application area.

Functional schemata (INGLISH).

sentence structure which allows an inference engine to match words with their functions in the sentence.

General expert system architecture.

Knowledge representation techniques and accompanying interpreter that allow the programmer to encode domain knowledge in a knowledge base separate from the algorithm that interprets it.

Hybrid system (Nexpert *Object*).

A system for formalizing knowledge with an object representation and a rule-based reasoning mechanism.

Indirect interviewing methods.

Methods which do not rely on the expert's abilities to articulate the information that is used.

Influence diagram (INFORM).

Conceptual and operational representation for domain expertise and involving three layers, relational layer, functional layer, and numerical layer.

Knowledge acquisition.

The extraction and formulation of knowledge derived from extant sources, especially from experts.

Knowledge acquisition aid.

Software support for application of knowledge acquisition techniques with guidance of its own.

Knowledge acquisition interface.

Set of tools that together present a "user illusion" of a language of application specific instantiations of constructs provided by the architecture.

Knowledge acquisition techniques.

Set of procedures, heuristics, or guidelines for preforming knowledge acquisition or knowledge engineering.

Knowledge acquisition tool.

Software support for application of knowledge acquisition techniques without any guidance of its own.

Knowledge editor.

Editor which aids in updating and reviewing the contents of a knowledge base.

Knowledge encoding and structuring.

Process resulting in an initial description of the knowledge base in the computational representation.

Knowledge engineering.

The discipline that addresses the task of building expert systems, the tools and methods that support the development of an expert system.

Knowledge framework eliciting.

Phase of determining the characteristics of the domain, the expert, the user and the application.

Knowledge island (Nexpert *Object*).

Set of rules related together by hypotheses or data.

Knowledge level analysis.

Analysis of a domain at the "knowledge level", regardless of whatever notations might be used to encode such knowledge.

Knowledge refinement.

Process of model focusing and validation.

Problem-solving method.

Heuristic for control. Weak methods are domain independent, while strong methods exploit domain knowledge to achieve greater performance.

Repertory grid technique (ETS, AQUINAS).

Clinical psychotherapeutic interviewing technique to categorize experiences and to classify their environment. This method is based on the description of constructs, that is, elements and their opposites.

Report (KNACK).

Contains information about general design objectives, tradeoff, decisions, analysis and test results, and detailed parts specifications.

Representational mismatch.

Mismatches between the way that an expert formulates domain knowledge and the way the knowledge is represented in a implementation. It typically occurs when the knowledge engineer imposes implementation-level primitives on the expert.

Rule model (ROGET).

Abstract description of subsets of rules built from empirical generalization about those rules - it corresponds to meta-level knowledge-.

Second generation expert systems.

Expert systems which emphazise accessibility and adaptability of the domain knowledge contained in the expert system (for the user and for the final user), and also includes a complete integration in their exploitation environment.

Task-level primitive, problem-solving primitive (and sometimes problem-solving strategy).

Notion of a "trigger", that is, a special relation between data and hypotheses such that when the data are found, a hypothesis is immediately activated.

Task-specific architecture, task-oriented architecture.

Expert system architecture which integrates particular knowledge representation formalisms and problem-solving strategies to perform a well-defined task such as hierarchical classification. Applications: classification, diagnostic, design, configuration.