EXPLANATIONS IN HYBRID EXPERT SYSTEMS

By

LAWRENCE GILL SCOTT

B.S., Michigan State University, 1972
M.U.P., Michigan State University, 1976

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science Department)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

March 1990

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Apr. 4, 1990

ABSTRACT

This thesis addresses the problem of providing explanations for expert systems implemented in a shell that supports a hybrid knowledge representation architecture. Hybrid representations combine rules and frames and are the predominant architecture in intermediate and high-end commercial expert system shells. The main point of the thesis is that frames can be endowed with explanation capabilities on a par with rules. The point is illustrated by a partial specification for an expert system shell and sample explanations which could be generated by an expert system coded to that specification.

As background information, the thesis introduces expert systems and the standard knowledge representation schemes that support them: rule-only schemes, and hybrid schemes that combine rules with frames. Explanations for expert systems are introduced in the context of rules, since rules are the only representation for which explanations are supported, either in commercial tools or in the preponderance of research.

The problem addressed by the thesis, how to produce explanations for hybrid architectures, is analyzed in two dimensions. Research was surveyed in three areas for guiding principles toward solving the problem: frame logic, metalevel architectures, and reflective architectures. With the few principles that were discovered in hand, the problem is then

analyzed into a small number of subproblems, mainly concerning high-level architectural decisions.

The solution proposed to the problem is described in two ways. First a partial specification for expert system shell functionality is offered, which describes, first, object structures and, then, behaviors at three points in time—object compilation time, execution time, and explanation generation time. The second component of the description is a set of extended examples which illustrate explanation generation in a hypothetical expert system. The solution adopts principles of reflective architectures, storing metainformation for explanations in metaobjects which are distinct from the object-level objects they explain. The most novel contribution of the solution is a scheme for relating all the ways that objects' slot values may be computed to the goal tree construct introduced by the seminal Mycin expert system.

The final chapter explores potential problems with the solution and the possibility of producing better explanations for hybrid expert system shell architectures.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

I appreciate the sacrifices of Virgina, Rainier, and Logan during my studies in general and during the preparation of this thesis in particular.

I wish to thank my employer, U S WEST, and especially supervisors John Agnew and Steve Tarr for support of my studies at the University of British Columbia.

I enjoy the friendships I made in Vancouver during my studies.

1 INTRODUCTION

Difficulty in understanding computer systems' behavior is a serious problem for developers, maintainers, and end users. The standard solution, static documentation, is at best a partial remedy. When documentation exists at all, it may be inaccurate or outdated. Even current, correct documentation can be problematic in that it may be understandable only to programmers.

For traditional software, the difficulty makes maintenance a hit-or-miss proposition. For new kinds of software, it can also retard users' acceptance. Endowing software with the ability to explain its behavior has been touted as a way to overcome the understandability barrier for a new class of software, expert systems. Some of the cited benefits of explanation include greater acceptance by users, faster learning, quicker recovery from errors, and easier debugging ([Buchanan84a], [Swartout83a], [Wallis84]).

Despite visions that vendors' hyperbole may bring to mind—of intelligent programs conversing with users in an articulate fashion—the explanation facilities of most expert systems, and the commercial shells used to construct them, are rather limited, in two distinct ways:

- Most shells' explanation facilities only explain expert systems constructed with rules and
- They support a rather narrow view of what constitutes an explanation.

Recent research has focused on the latter limitation from two perspectives: knowledge representation and explanation generation. Seeking to improve the information available to explanation mechanisms, researchers in knowledge representation have devised alternative constructs

for building expert systems. Seeking to improve the final explanations, other researchers have sought to improve the explanation mechanisms, using linguistic and psychological models. These efforts have ignored the first limitation, that of explaining only rules. That neglect is unfortunate because the trend in expert system shells is toward hybrid architectures which combine rules with another knowledge representation paradigm: frames, or objects ([Gevarter87], [Harmon89a, b, and c]).

This thesis takes first steps toward filling the gap between explanation research and expert system building practice. It provides a framework for providing explanations for the hybrid representation found in many intermediate and high-end commercial expert system shells. The main point of the thesis is that frames can be endowed with explanation capabilities on a par with rules. The point is illustrated by a partial specification for an expert system shell and sample explanations which could be generated by an expert system coded to that specification. The framework described in the thesis suffers from the limitation that its view of explanation is too narrow, and the final chapter of the thesis considers whether the framework can scale up to produce better explanations.

The thesis is organized as follows. Standard explanations for rules—i.e those supported by commercial shells—are described in chapter 2. How to produce standard explanations for hybrid architectures—the topic of this thesis—is pondered in chapter 3. A solution is proposed in chapter 4 and illustrated in chapter 5. Finally, producing better explanations for hybrid architectures—beyond the scope of this research—is explored in chapter 6.

2 A SURVEY: KNOWLEDGE REPRESENTATION AND EXPLANATION

Though not yet as commonplace as database systems, expert systems—also known as knowledge-based systems—seem on their way to becoming so. Just ten years ago only a few examples existed in university laboratories. Buoyed by academic successes, professors turned entrepreneurs and founded firms to market expert system tools. In recent years corporate giants like Texas Instruments and International Business Machines have joined the fray. In that ten years, the tools have evolved along size and complexity dimensions. Their explanation capabilities, however, have not kept pace, as we see later in this chapter.

This chapter supplies background information assumed by the remainder of the thesis. A brief introduction to expert systems describes their capabilities, limitations, and typical system architecture. Expert systems attempt to explicitly represent knowledge about the world, and we look at the two knowledge representation schemes in commercial tools: rules only, and hybrid systems which combine rules and frames. Expert systems extended the notion of on-line help to dynamic explanations of behavior; the final sections in this chapter consider the range of explanations currently possible. The inadequacies of canned text are contrasted to the standard approach commonly available in expert system shells.

2.1 EXPERT SYSTEMS

An expert system is computer software that solves a problem thought to require intelligence or reasoning. Many practitioners prefer to call them

knowledge-based systems, since that label does not appear to exclude software that functions as an intelligent assistant or intelligent colleague. However, other artificial intelligence systems can be knowledge-based, e.g. systems for natural language processing, vision, and machine learning. Thus this thesis will bend to the tide of usage and use the term *expert system* for any system that employs expertise and reasoning to solve problems, wherever that expertise lies on the novice-to-expert continuum.

Expert systems have been successfully applied to a broad range of problem types: classification, diagnosis, monitoring, configuration, planning, and design are frequent examples. Within those problem types, individual systems are usually "expert" only in a narrow range of expertise. And some knowledge-based problems are not yet amenable to solution by expert systems, for example problems that require spatial, temporal, or analogical reasoning.

The architecture of a prototypical expert system is illustrated in Figure 1. Its key features are the user/developer interface, inference engine, knowledge base, and some number (perhaps zero) of external access interfaces.

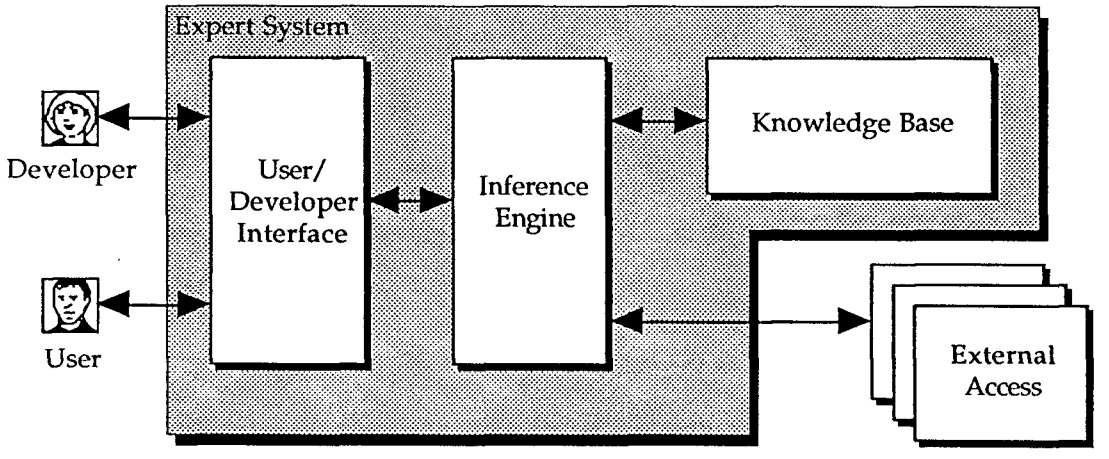

Figure 1: Prototypical Expert System Architecture

The inference engine is the heart of an expert system, since inference is a prime distinction between expert systems and other kinds of software. The nature of the inference engine determines how knowledge will be represented in the knowledge base by the developer. During a consultation, the inference engine dynamically drives the processing, whence its name. The dynamic processing reflects both static knowledge and dynamic, situation-specific facts in the knowledge base.

The developer constructs the static portion of the knowledge base, representing knowledge about the real world domain that the expert system uses to solve problems. The static knowledge base represents facts, complex structures, and/or relationships between structures. The user and external access interfaces add dynamic, situation-specific facts into the knowledge base. Based on those facts, the inference engine infers others to solve its assigned problem.

The user/developer interface is the point of contact between humans and an expert system. The developer uses the developer interface to build the static portion of the knowledge base, to tailor the user interface, and to establish external accesses. At run time, the user interface enables the user to interact with the system in whatever ways the developer has provided. Typically one of those ways is to ask for and receive explanations of the expert system's behavior.

External access interfaces may link to sensors, databases, other conventional software, and/or other expert systems.

Some expert systems are written in programming languages, e.g. Lisp, Prolog, and C. Others are written using tools called shells, e.g. KEE, ART, and Knowledge Craft. *Shells* typically consist of an inference engine, a developer interface, and the generic portion of a user interface. Some shells provide

facilities for external access as well. A shell provides the syntax for representing knowledge during development, and a procedural semantics which effects inference during execution.

Obviously, a shell's ability to explain its behavior is tied to how it represents knowledge. Thus some background in knowledge representation in expert systems is presented before we consider explanation in detail.

2.2 KNOWLEDGE REPRESENTATION SCHEMES

Brachman and Levesque provide a concise, clear statement of the significance of knowledge representation for artificial intelligence research, in the introduction to [Brachman85b] :

> The notion of the *representation of knowledge* is at heart an easy one to understand. It simply has to do with writing down, in some language or communicative medium, descriptions or pictures that correspond in some salient way to the world or a state of the world. In Artificial Intelligence (AI), we are concerned with writing down descriptions of the world in such a way that an intelligent machine can come to new conclusions about its environment by formally manipulating these descriptions.

In his Knowledge Representation Hypothesis, Smith states the notion more formally ([Smith85]):

> Any mechanically embodied intelligent process will be comprised of structural ingredients that ... represent a propositional account of the knowledge that the overall process exhibits and ... [that] play a formal but causal and essential role in engendering the behavior that manifests that knowledge.

Smith's hypothesis introduces the two crucial ingredients of representation: structure and behavior. Schemes for representing

knowledge, such as rules and frames, must supply both ingredients: structure and behavior that can act on the structure. The combination of structure and behavior must enable the scheme to convey meaning about a slice of the world. Minimally, a knowledge-based system encoded in any scheme must be able to determine what it "knows" about the slice of the world it models.

The desirability of a representation scheme can be viewed along two dimensions: expressive adequacy and notational efficacy ([Woods83]). Expressive adequacy is what the representation allows to be said. Notational efficacy concerns several attributes of a representation scheme, such as its computational efficiency for different kinds of inference, how concise the scheme is, and how easy it is to modify. Levesque and Brachman have noted a fundamental tradeoff between the two dimensions: limitations on a representation scheme's expressiveness are necessary if its reasoning is not to become computationally intractable ([Levesque85]).

Hayes notes that to be clear about exactly how a scheme represents knowledge about the world, the scheme must have an associated semantic theory. A semantic theory is an account of the way in which particular configurations of the scheme correspond to particular arrangements in the external world ([Hayes85a]). Some representation schemes have very precise semantic theories, e.g. logic-based rules; other schemes seem to have no formal semantic theory.

Most knowledge representation schemes model the world as a collection of individuals and relationships that exist between them. States are the collection of all individuals and relationships at one point in time. Schemes can be differentiated by their viewpoint into this common model. For example procedural schemes (e.g. Lisp code) are based on the viewpoint of state transformations ([Mylopoulos84]).

2.3 RULE-BASED KNOWLEDGE REPRESENTATION SCHEMES

Rules are a very natural way to represent some forms of knowledge. Newell and Simon report that experts often discuss their knowledge in language corresponding to rules ([Newell72]). Rules are straightforward and easy to understand, due to their simplicity of notation. Thus it is not surprising that rules were the dominant form of representation in the first generation of expert systems.

The structure of a rule can be expressed in either of two equivalent forms:

IF <premise> THEN <consequent>

or alternatively

<consequent> IF <premise>.

The premise can consist of multiple clauses connected by logical connectors (AND, OR, NOT, etc.). In many representation schemes the consequent may also contain more than one statement (in which case it is interpreted as actions to be carried out in sequence).

The behavior of rule-based knowledge representation schemes is uniform and straightforward ([Parsaye88]):

(1) Knowledge exists in the form of rules and facts.

(2) New facts are added.

(3) Combining the new facts with existing facts and rules leads to the deduction of further facts.

Two strategies are available to control inference in the third step. In the first, reasoning proceeds forward with a rule whose premise matches the facts; its firing adds one or more new facts to the knowledge base, and steps 2 and 3

repeat. This approach is called data driven, or *forward chaining*. Alternatively a goal can be established, and rules considered which conclude that goal; if some rule concludes the goal, but part of its premise is unknown, the premise becomes the new goal. This approach is goal driven, or *backward chaining*. The two control strategies behave differently and are appropriate for different kinds of expert systems. However a given rule can be used with either or both strategies.

Rule-based representations rank high in expressive adequacy. They can be about specific objects ("If the drill has a damaged cord,...") or entire classes of objects ("If any device has a damaged cord,..."). Rules about other rules, called metarules, can provide more focused behavior than simple forward or backward chaining (see e.g. [Parsaye88]). Indeed rules can be used to express the same knowledge as other representation schemes (see e.g. [Walker87] and [Thayse88]).

Rules may or may not rank so high in notational efficacy, depending on the size and nature of a rule-based expert system. We consider the three aspects of notational efficacy individually

Rule-based inference requires attention if it is to be computationally efficient in large applications. On the one hand, the goal directed nature of backward chaining insures that only the rules appropriate to situation-specific facts are considered. And Forgy has devised a way to significantly improve the performance of forward chaining processing, the Rete Algorithm ([Forgy82]). On the other hand, rule organization techniques such as Rete, metarules, and rule groups had to be developed to ameliorate control and performance difficulties that develop during inference in large rule bases, e.g. poor performance or unfocused inference ([Davis80], [Chandrasekaran84], [Parsaye88]).

Rules are concise in some contexts, and not in others. Mylopoulos notes rules encourage "conceptual economy", in that a rule need be stated just once, even if used in different ways in a knowledge base ([Mylopoulos84]). Others note problems that can result from forcing knowledge not well suited to rules into a rule scheme. For example, a rule-based approach to procedural control makes flow of control implicit and context explicit—exactly opposite from the desired state for some problems ([Georgeff86]). Chandrasekaran notes a 20%/80% effect in some problem domains: while much of the domain is represented in relatively few rules, the remaining domain knowledge requires many more ([Chandrasekaran84]).

Rules are generally viewed as easy to modify. A developer can often create rules without worrying in advance about the order in which actions should be taken ([Parsaye88]). New rules can be added to flesh out a problem domain with little or no impact on previously debugged rules. However Woods notes that adding significant new perspectives, e.g. time or situation variables or intermediate steps and agents, can require rewriting a rule base from the ground up ([Woods83]).

The discussion thus far has obscured an important point about rules: there are actually two varieties in widespread use. Rule schemes close to first-order predicate logic, such as Prolog, have a clean, well understood, and accepted formal semantics. Production rules, found in most expert system shells, do not. Chandrasekaran notes that existence of a rigorous semantics does not necessarily make a scheme better for building systems [Chandrasekaran84]. We see in section 2.6 that lack of a formal semantics does not preclude a scheme from having a viable explanation facility.

Logic rules and production rules may be further contrasted by their viewpoints into the world model consisting of individuals, relationships, and

states. Production rules are generally considered to be procedural, i.e. based on a view of state transformations like Lisp code. Logic-based rules are based on a viewpoint of true assertions about states; however in endowing a logic-based scheme with a procedural semantics, their behavior becomes very much like production rules. Henceforth we ignore the difference between the two kinds of rules, and continue to refer to *rules*, meaning both varieties.

In summary, rules are an effective scheme in which to represent many, but not all, kinds of knowledge in expert systems. While their expressive adequacy is high in theory, in practice their notational efficacy constrains their usefulness.

2.4 FRAME-BASED KNOWLEDGE REPRESENTATION SCHEMES

This section discusses frames as a knowledge representation scheme. The intent here is not to suggest frames (only) as an alternative to rules for representing knowledge. Rather, frames are described as a prelude to discussing hybrid representation schemes in the following section. If contrasts are to be drawn at all, they should be between the rule only vs. hybrid (rule and frame) schemes.

Frames are organizational devices for modeling explicitly the objects in a real world problem domain, relationships between those objects, and contexts in which different relationships apply. Like rules, there is evidence that frames are a natural way of organizing knowledge. Early research by Bartlett demonstrated that human memory contains knowledge structures that aid in interpreting new information ([Bartlett32]).

In a seminal paper (reprinted as [Minsky85]), Minsky argues that frames are more desirable than traditional logic for representing knowledge to solve

many realistic, complicated problems. First order logic fails in expressive adequacy in that it is poorly suited to representing approximate solutions, which Minsky sees as vital in human problem solving. Logic also fails in terms of notational efficacy. Minsky claims a human thinker reviews plans and goal lists, and while one can program these with theorem proving, he argues one really wants to represent them directly, in a natural (perhaps procedural) way.

The notion of frames was conceptualized independently from object-oriented programming. Minsky notes the origins of the former in work by Bartlett and Kuhn. The latter did not originate with, but was popularized in, the language Smalltalk. As Parsaye and Chignell note, the distinction between the two concepts is rapidly disappearing ([Parsaye88]). Hence in this thesis we use the terms *frame* and *object* interchangeably.

There are two aspects to the structure of frame systems: the internal structure of objects and the relationships between objects. Figures 2 and 3 illustrate the two aspects.

An object is composed of a collection of attributes, commonly called *slots*. In simpler frame systems each slot has just an associated value. However as shown in Figure 2, in many modern frame systems, slots themselves are composed of collections of *facets*. One of the facets is typically the **Value** facet[1]. Thus a slot's value in the simple view equates to the value of the slot's **Value** facet in the more complex view. Other facets can define a slot's allowable values, its default value, how many values it may have, etc. Not all slots need have the same facets.

[1]Henceforth in body text, words in the object language appear in **bold Helvetica Narrow** font, while words in ordinary (meta-)language appear in Palatino font.

Object Name:	Sample Object	
Inheritance:	Is Instance Of	Sample Objects
Slot Name:	Sample Slot	
Facets:	Value	true
	Legal Values	(true false)
	Default Value	false
	How Many Values	1
	...	...
Slot Name:	Another Sample Slot	
Facets:	Value	(red white blue)
	Legal Values	(from-class 'Colors')
	Default Value	(white)
	How Many Values	unlimited
	...	...
Slot Name:	Third Sample Slot	
Facets:	Value	3.1714
	...	...
...		

Figure 2: Internal Structure of a Frame, or Object

Figure 3 illustrates some sample objects and the hierarchical structure that relates them. The links between individual frames are of two types: superclass-subclass links and class-instance links, indicated by solid and dotted lines respectively. That is, (super)class **Vehicle** has two subclasses, **Volvo** and **Mercedes**. And class **Volvo** has two instances, **My Old Volvo** and **Richard's New Volvo**. A class object may have any number of subclasses and/or instances. Instance objects are prohibited from having descendents in some frame schemes; in other schemes there is no such prohibition. The two types of links are both referred to as *IS-A links*, as in "**Volvo** is a **Vehicle**" or "**My Old Volvo** is a **Volvo**".

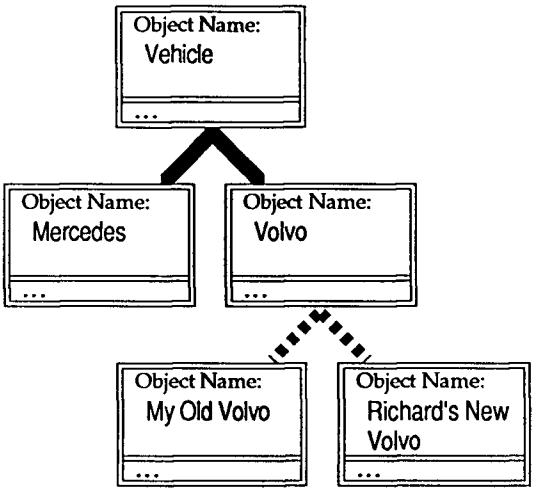

Figure 3: Frame Hierarchy

Frame hierarchies are similar to semantic networks in many respects. Both schemes represent knowledge in conceptual units like frames with role descriptions like slots. And both schemes relate the conceptual units with a hierarchy expressing generalization relationships ([Brachman85a]). Although semantic nets were popular during the 1970s, Parsaye notes that they are used on their own in relatively few systems today ([Parsaye88]). Nor do any commercial expert system shells support them. Thus we do not discuss semantic nets further in this thesis.

Frames' behavior extends the notion of static record structures in two ways. The two ways directly relate to objects' internal structure and the relationships between objects.

Internally, a slot's value is not restricted to static data, but can be an active data element, i.e. procedural code. In early frame formalizations this capability was called procedural attachment. In the object-oriented programming tradition, the same capability is achieved via messages passed

between objects that activate methods in the target object. This thesis uses the message and method terminology.

A slot may be associated with a special kind of method, called a demon. Demons are methods augmented with an invocation condition. Demons are not activated by the explicit message-method mechanism; instead, a demon activates whenever its invocation condition is satisfied ([Parsaye88]).

The frame representation provides a flexible mechanism for inheriting information between objects. That is, a class object normally passes some (in some schemes, all) of its slots down the IS-A links to its descendents, i.e. its subclasses and/or instances. Inheritance occurs down the hierarchy to all descendents, regardless of the number of intervening objects; that is, **My Old Volvo** in Figure 3 would inherit from both the **Volvo** and **Vehicle** objects.

Inherited slots can contain data or methods, and all facets of an inherited slot are inherited. An object may specify nothing about an inherited slot, in which case the slot behaves for the heir just as it does for the object where originally defined. Alternatively, an object can specify information about an inherited slot locally, overriding its inherited behavior to create tailored local behavior.

Besides capturing structural information in their IS-A links, frames also allow the full range of expression of whatever language their methods are coded in—Lisp for example. Their great flexibility permits a developer to define and use any datatype that the combined method language and IS-A links formalisms allow.

With such broad expressiveness, frames' notational efficacy might be expected to suffer (remember the tradeoff between expressiveness and tractability). Fortunately, the hierarchical organization useful for structuring real world knowledge also provides efficiencies in storage as well as inference

([Chandrasekaran84], [Parsaye88]). The inheritance capability of frames adds to their modularity and compactness of expression ([Parsaye88]). Finally frames facilitate modification: since knowledge is structured in units that correspond to the real world, it is usually obvious which object needs to be modified to effect a desired change.

Little early work on frames presented a formal semantic theory for them. In comments preceding their reprint of Minsky's proposal for frames, Brachman and Levesque complain that "vagueness and general lack of rigor has followed many who pursued the frame ideas, as if the topic itself demanded a certain informal style of research". They continue that "frames as a representational framework seem to have much more to do with cognitive memory models than with mathematical logic and logical inference" ([Brachman85b]). In other work, Brachman supplies some answers about the semantics of frames, which we consider more closely in section 3.3.

Frames are based on multiple viewpoints into the world model of individuals, relations, and states. The external, hierarchical structure of frames is explicitly based on the viewpoint of individuals and relationships. Frames' methods, consisting of procedural code, adds the procedural viewpoint of state transformations. Frames' slots are based on the view of relations between objects, à la semantic networks.

Frames represent an intuitive and flexible way to model a slice of the world. Objects in the real world can be placed in one-to-one correspondence to objects in the model. Inheritance is a powerful mechanism for efficient, sharing of structure and behavior. Frames alone represent a powerful, emerging programming paradigm—object-oriented programming. In the next section, we introduce the marriage of rules and frames in hybrid expert system shells.

2.5 HYBRID KNOWLEDGE REPRESENTATION SCHEMES

Some researchers purposely eschew hybrid representations, and advocate representations based entirely on rules, specifically logic rules, because of their well-defined semantics. [Jackson89] makes a particularly strong argument for this view:

> Logical formalisms are rich enough to provide different concrete architectures, while on the other hand the use of logic provides a unifying framework for the system which saves it from the unstructured richness of hybrid systems.... Hybrid systems are not useful for building expert systems, precisely because they offer a bewildering array of possibilities and little if any guidance as to which are appropriate to what tasks.... Although logic's semantic and computational properties are somewhat negative, the important point is that these properties are known at all[2].

Jackson et al's purist view stands in stark contrast to the pragmatic approach maintained by hybrid systems' proponents ([Bobrow85], [Parsaye88], [Chandrasekaran84]). Even relative purists Levesque and Brachman argue that "there is no single *best* language" ([Levesque85]). Thuraisingham calls tools in which the object-oriented model is augmented with logical deduction "generic representations" and praises their power to model structural entities and rules ([Thuraisingham89]). Jackson, et al call them "high level programming environments" and admit their tools prove useful in appropriate applications ([Jackson89]).

This thesis does not join the argument, but obviously sides with the pragmatists. One motivation for studying the hybrid tools is that they are the model which developers of powerful commercial expert system shells have

[2]It should be noted that Jackson et al are arguing for a more powerful logic than first order predicate logic.

adopted. The list of such shells includes Knowledge Engineering Environment (KEE), Automated Reasoning Tool (ART), Knowledge Craft, Nexpert Object, Aion Development System, Knowledge Base Management System, Goldworks, and others.

These shells have different foci. KEE for example is at heart frame-based, with rules added on. ART's view is the converse, primarily rule-based, with frames added on. Knowledge Craft is a loosely coupled collection of tools, providing forward chaining rules (OPS), backward chaining rules (Prolog), and a frame language (CRL). In chapter 3 we clarify the thesis' view of a prototypical hybrid shell, and define the explanation facility we provide for it.

There are advanced representations used in expert systems we could, but are not, considering part of hybrid shells. Some of these are multiple worlds and truth maintenance, scripts, and blackboard architectures.

Having described hybrid expert system shells, we are now ready to turn to our main concern with them, namely capability to explain their behavior.

2.6 EXPLANATION IN EXPERT SYSTEMS

An expert system's explanation facility provides explanations of its actions and conclusions to a variety of users, including developers and end users. Almost since the expert system era began, researchers have considered explanation to be one of AI's most valuable contributions ([Wick89a]). This has led to a difference in users' expectations between expert system and other software: users may anticipate weeks or months learning other software, but expect to begin useful work with an expert system almost immediately ([Wexelblat89]).

The simplest mechanism for generating explanations is canned text. The developer attempts to anticipate all situations requiring explanation, and creates and stores text strings to recall at an appropriate later time. Problems with applying the canned text approach to systems of any significant size or complexity are well known ([McKeown85a], [Swartout83b]). As with any documentation, it is hard to maintain consistency between the text strings and the actual functioning of the system as it changes over time. In a complex system, it may be impossible to anticipate questions and responses for all situations that may occur. Difficulties in achieving high quality explanations result directly from the fact that the system's explanations are not based on a conceptual model. Thus, while canned text may suffice for error messages and help systems, the approach is not considered to be robust enough for intelligent, dynamic explanations ([Wick89b]).

Mycin, an early rule-based expert system for diagnosing infectious blood diseases developed at Stanford University in the 1970s, has been enormously influential in the spread of expert systems. Figure 4 (modified from [Clancey86]) illustrates other research systems that Mycin spawned at Stanford. It also indicates the importance of one derivative, Emycin on the commercial development of expert systems. Emycin—short for Essential or Empty Mycin—removed Mycin's domain specific knowledge and was applied to other problem domains. Emycin demonstrated the viability and potential of expert system shells. Many first generation shells were simply reimplementations of Emycin, e.g. M.1, Personal Consultant, and Expert Systems Environment.

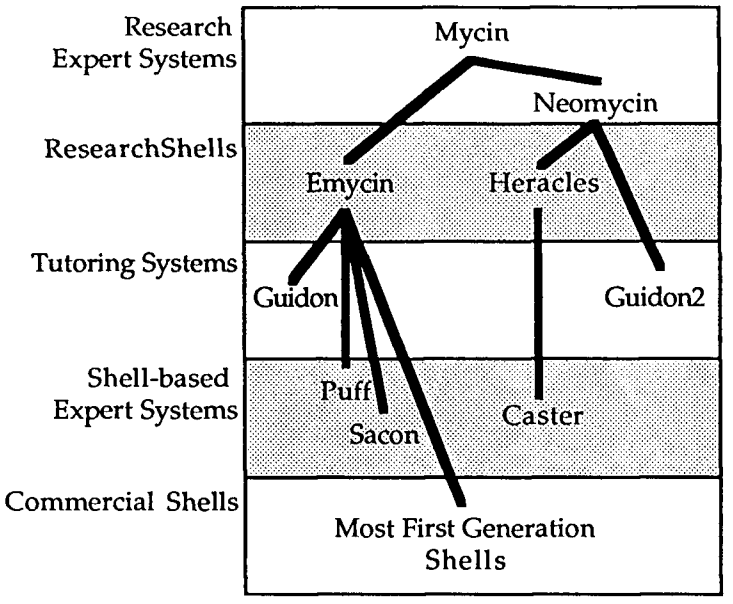

Figure 4: Influence of Mycin Expert System

The Mycin system introduced the explanation mechanism that is the foundation of most commercial explanation facilities today: templates ([Scott84]). Templates are text phrases with slots that can be filled by different words for different situations; individual templates can be strung together to produce an explanation of a multi-step process. In Mycin, a template is associated with each rule, and the slots in the templates are filled by translations of variables instantiated in the rules. Templates are more flexible than simple canned text, but they can suffer from the same problems and they require effort to produce readable output ([McKeown85a]).

Mycin's templates are used to generate explanations by interrogating two knowledge structures: a goal tree and Mycin's rules. The dynamic, consultation-specific goal tree consists of nodes representing goals and subgoals pursued during the consultation. Goals are simply the need to know the values of variables. When a goal may be satisfied by application of one or

more rules in Mycin's static knowledge base, the corresponding node in the goal tree is indexed to the relevant rule(s). See Figure 5 for a portion of a goal tree ([Scott84]).

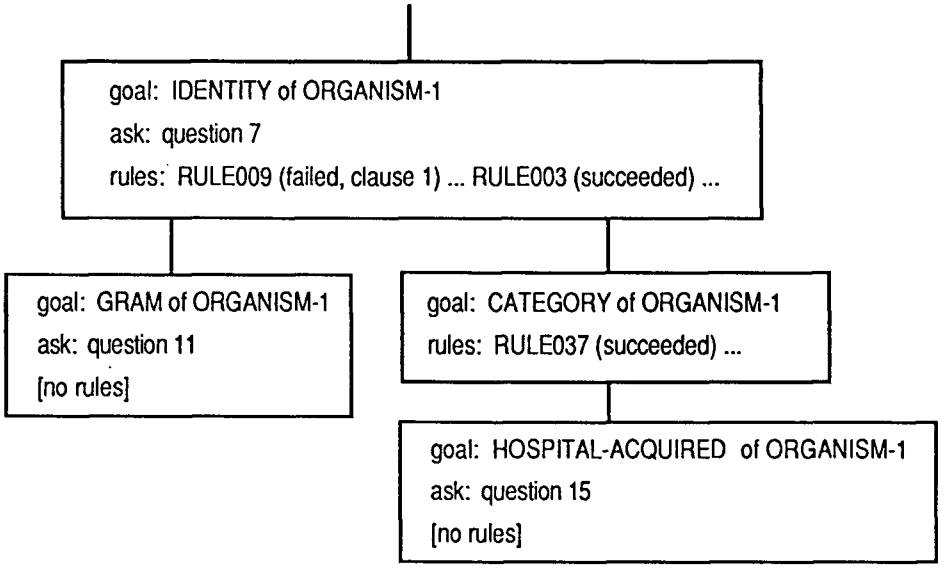

Figure 5: Portion of a Mycin Goal Tree

These two knowledge structures, i.e. static rules and dynamic goal tree, enable Mycin's template mechanism to produce two kinds of explanations: *why* a piece of information is being asked of the user and *how* a goal was achieved. Because these are not the only possible interpretations of the queries how and why, we distinguish the Mycin interpretations as HOW? and WHY? in this thesis.

The explanation facilities in commercial expert system shells are limited to answering these HOW? and WHY? queries for rules. A few shells provide some minimal augmentation, such as a graphical view of the goal tree or

simple explanations of the "objects" implicit in rules (which we refer to as WHAT-IS-IT? explanations). As Wick and Slagle note, almost all research into explanations relies on additional knowledge not available in existing shells ([Wick89a]), a topic we take up again in chapter 6.

It seems ironic that hybrid shells, with all their richness of knowledge representation, provide explanations only for slot values that are inferred by rules. We are now ready to explore what it means to explain slot values computed by any of the variety of mechanisms found in hybrid shells.

3 THE PROBLEM: GENERATING EXPLANATIONS IN A HYBRID SHELL

This chapter makes explicit the problem addressed by the thesis. Its first section describes important features of the hybrid shells for which explanation functionality is specified in chapter 4. In the second section, the problem of adding explanation functionality is refined, by clarifying what structure within a hybrid architecture to explain, and what queries to answer.

Three areas of research appear to bear on the problem of explaining frames, and a section in this chapter is devoted to each. Taking a cue from Georgeff and others, that the semantics of a scheme can be important for providing explanation capability [Georgeff86], the logic of frames is explored, with disappointing results. Explanation is a form of metareasoning, i.e. reasoning about reasoning; thus the section after that considers whether research into metalevel architectures provides any guiding principles or pragmatic suggestions for explaining frames. Explanation may also be viewed as a form of computational reflection, in that a system's explanation facility may involve structures representing aspects of the system itself; hence a third exploration section considers whether research into reflective architectures provides principles or suggestions. Of the three potential sources of ideas, the last is the most helpful.

A summary and restatement of the problem concludes the chapter.

3.1 FEATURES OF HYBRID EXPERT SYSTEM SHELLS

The hybrid shells we target for explanation are those with essentially the mix of rule-based reasoning and object-oriented programming provided in

commercial expert system building tools such as KEE, ART, Knowledge Craft, etc. These shells provide full object-oriented programming capabilities including IS-A hierarchies, inheritance and message passing. They also provide robust rule-based inference with both forward and backward chaining.

The rules and frames in these systems are integrated. That is, rule-based inference can stimulate methods, and conversely, methods can stimulate rule-based inference. As a concrete example, consider KEE, which provides two different languages to effect rule-based and frame-based reasoning. KEE's rule language is called TellAndAsk and its method language is Common Lisp. Both the premise and consequent of TellAndAsk rules may contain TellAndAsk's well formed formulas (wff's) and/or Lisp expressions that invoke methods. Forward and backward chaining are invoked by the expressions:

(ASSERT <wff> <ruleclass>)

(QUERY <wff> <ruleclass>).

Rule-based reasoning can be invoked by using these expressions within either rules or methods ([IntelliCorp87]).

It is important to note that these frame systems typically include accessible generic classes for objects, rules, methods, and demons. Assuming these classes are first class objects, then adding structure and behavior to them for explanation purposes will make that structure and behavior apply to all instances of objects, rules, methods, and demons.

3.2 ADDING EXPLANATION FUNCTIONALITY TO A HYBRID SHELL

A hybrid tool's integration of rules and frames makes it possible to accomplish the same behavior in a variety of ways. This is the "unstructured richness" that Jackson, et al lament. While it may indeed be advisable to heed the advice of Parsaye and Chignell, to "maximize inference in the rules and retrieval in the frames" [Parsaye88], we can reasonably assume that in an imperfect world, expert systems have been and will be built that mix rules and frames in a less optimal fashion. This assumption provides the motivation to explain frames on a par with rules. Like Wick and Slagle, this thesis aims to do so using only using the technology currently available in hybrid shells [Wick89a], i.e. not requiring advanced explanation methodologies described in chapter 6.

Just what is it then that we wish to explain? The objects in frame-based systems have three levels of structure we might focus on: objects themselves, objects' slots, or slots' facets. The level that is usually asserted by rules and assigned values by methods is slots (more precisely the slots' **Value** facets, but this distinction is not important). Thus our problem is to explain a slot's value for any of the ways that value might be assigned.

There are three general ways that slots' values can be determined. Some slots, representing static information, never change values during a consultation; we label this situation *initialization values*. Secondly, a slot's value may not be specified locally at all, but rather be determined by the frame hierarchy's *inheritance* mechanism. Finally a slot's value may be determined dynamically during a consultation by any number of ways, which we refer to collectively as *slot-changing agents*. Slot-changing agents include rules;

methods, including demons; user supplied values; and values returned from access to external processes.

At a minimum then, we want to answer Mycin-style HOW? and WHY? queries for all the ways a slot's value can be determined in a hybrid shell. We prefer the explanation mechanism to be as uniform as possible, regardless of how a slot's value may be determined. Ideally we would also like our solution to scale up, that is, to accommodate other kinds of explanations.

The topic of explaining frame-based reasoning has been largely ignored by other researchers of expert system explanations. The search for ideas of how to attack the problem led naturally to the topics presented in the next three sections.

3.3 THE LOGIC OF FRAMES

A semantics was not specified by early proponents of frames, who were more interested in richness of representation and the possibility to express situations which were awkward in first order predicate logic. Remember that logicians complained about the "informal style" of frame research.

When logicians applied their own rigor to the question of frame logic, they uncovered a subtle but severe constraint to formally defining a frame semantics. If a frame notation "allows cancellation, but provides no mechanism for noting certain facts as uncancellable, then it simply cannot express universal truths, ... [or] more precisely, it can only represent universal truths extensionally (by explicitly indicating all cases)" ([Brachman85a]).

It is interesting to note that software engineering perspectives on object-oriented programming have observed that object-oriented design for reusability of classes leads to a principle which would avoid Brachman's

concern: the appropriate use of inheritance is to model a type hierarchy, in which every class should be a particular kind of its superclasses. "Subclasses should add responsibilities to their superclasses; they should not cancel inherited responsibilities, or override them to become errors, or no behavior at all" ([Wilkerson89]).

Ignoring procedural attachment, i.e. methods, Brachman has carefully analyzed the semantics of IS-A links ([Brachman83]). He clarifies that IS-A links are not synonymous with inheritance, which he calls an implementation issue, rather than one of expressive power. Noting that the semantics of a general-purpose IS-A mechanism cannot be predicted in general, Brachman (and also Hayes ([Hayes85b])) have categorized the variety of inferences that the various uses of IS-A links suggest. Brachman then goes on to make a concrete proposal for what IS-A links should be. His suggestion involves enhancing IS-A links with information that looks ("suspiciously", he says) like formalisms that make up special cases of standard logical statements: assertional force, modality, and quantifiers.

If we leave aside both methods and the matter of overriding defaults, it is generally agreed that frames as data structures are essentially just bundles of properties. As a representational language, they are otherwise mostly equivalent to first order predicate logic [Hayes85b]), although their form emphasizes certain compelling patterns in knowledge representation that do not emerge from predicate logic based representations ([Brachman83]).

Study of frame-based reasoning was not particularly helpful to finding principles or techniques useful for explaining frames. Existing frame representations have a semantics that can only be described as ad hoc, if indeed it can be described at all. The most concrete proposal for endowing

frames with a formal semantics lies in enhancing their IS-A links, by making explicit their logical characteristics.

The notion of enhancement of links was considered, but discarded for this thesis. It is more complicated than the solution proposed in chapter 4. As concrete evidence, this thesis' proposal could be implemented in existing shells because of the existence of objects, rules, methods, and demons as first class objects. The IS-A links, on the other hand, are not first class objects in these shells.

A second reason that the study of semantics was not helpful for explaining frames is that methods are not addressed at all. However, despite frames' lack of formal semantics, we need not despair that they are unexplainable. Remember that production rules do not have an established semantics either, but have been successfully explained nonetheless.

3.4 METALEVEL ARCHITECTURES

Section 2.3 introduced the notion of metarules, used to focus rule-based processing. As used in philosophy, linguistics, and knowledge representation, the prefix *meta-* is a kind of word schema: meta-x is interpreted as "x about x". What a meta-x is "about" is called the "object-level". Thus a metarule is a rule about object-level rules. Metalanguage is language about some object-level language. A metalevel architecture includes some components which are about other components (those at the object-level). And, unfortunately but unavoidably, a metaobject is about some object-level object.

Explanation in expert systems is an example of metareasoning, that is, reasoning about the object-level reasoning the expert system is doing (or has

done) in solving its problem. Therefore the metareasoning literature is reviewed as a possible source of principles or design suggestions for explaining frames. We consider first the range of applications of metareasoning, then discuss whether the literature provides useful suggestions.

Davis proposed metarules as a way to control inference in the Mycin system ([Davis80]) and control remains the most strongly and widely advocated application of metareasoning. The aim is to improve performance by pruning a search tree. This usage relies on an ability to represent properties of object-level knowledge and the state of the inference process. Most approaches are rule-based, and hence rely, like Davis, on metarules ([Aiello88]).

Extending the application of metareasoning beyond control, some researchers have used metareasoning as an abstraction mechanism to increase expressive power. One example occurs in automated deduction systems, where inference rules can be defined at the metalevel. The search space at the metalevel is smaller than at the object-level, since a single metaproof represents multiple object-level proofs. Other problems that have been attacked in this manner include: awareness of what a system knows, awareness of beliefs, non-monotonic reasoning, reasoning about changing situations, reasoning about different (and possibly inconsistent) evolving theories, and reasoning about multiple views of objects ([Aiello88]).

Metalevel architectures also appear in some object-oriented languages. Cointe contrasts a number of such languages, which vary greatly in the access to the metalevel permitted by the language to users. For example in Smalltalk-80, the metalevel is not at all accessible to the user of the language,

whereas the ObjVlisp language permits access equally to instances, classes, and metaclasses ([Cointe88]).

Finally Aiello and Levi survey some applications of metareasoning to interfaces. Some examples are mechanized interfaces, e.g. between separate bodies of knowledge, or between knowledge bases and databases. Other examples are between knowledge bases and users, in applications such as knowledge acquisition, user modeling, and (finally!) explanation.

Three principles or suggestions resulted from this review of metareasoning that have importance for the question of explaining frames. The key observation is what data structures support explanation facilities. Two other principles concern the choice of language(s) at the meta- and object-levels, and the advantages of control knowledge at the metalevel.

Sterling and Shapiro devote an entire chapter in [Sterling86] to metainterpreters, which they define as "an interpreter for a language written in the language itself". The chapter provides several examples of Prolog metainterpreters which can answer HOW? and WHY? queries. The technique of metainterpretation they present consists of metaprograms in which predicates take object-level programs as one of their arguments. Other arguments that are carried along in the metaprogram's predicates provide insight into the data structures needed to support HOW? and WHY? explanations. Those additional arguments are the equivalent of Mycin's goal tree and the currently active rule.

Metalevel architectures vary as to whether they are monolingual or bilingual. Jackson et al argue for the latter approach, a purely declarative object-level "with no concern for efficiency" and a metalevel for control with efficiency "its most prominent aspect" ([Jackson89]). This debate is not resolved, however; Hayes argues that factual and control information should

be represented in the same scheme, so that control can be involved in inference ([Hayes85a]).

Whether mono- or bilingual, Jackson et al cite numerous advantages to separating control knowledge from domain knowledge. One of these is the possibility of deeper explanations. What Jackson et al refer to are the kinds of explanations with deep knowledge sources that are taken up in the final chapter, not our concern with better explanations using existing tools and knowledge sources.

Thus, this review of metareasoning has proven only somewhat more relevant than the review of frame logic to the problem addressed in this thesis. It reveals that there are many alternative uses of metareasoning and some variety in metalevel architectures. While the review found no clear principles for metalevel architectures in general, at least metainterpretation techniques indicate the flexibility of Mycin's data structures in producing explanation facilities for a variety of rule formalisms. However Mycin-style production rules and metainterpretation of logic programs each deal with a single, uniform representation scheme. The metareasoning review has provided little guidance for explaining hybrid systems' wide variety of mechanisms for assigning slots' values.

3.5 REFLECTIVE ARCHITECTURES

Smith ascribes much of the subtlety and flexibility that humans bring to bear on the world to our ability to reflect: that is, our ability to not only think about the external world, but also to think about our own internal ideas, actions, feelings, and past experiences. Interest in the self-referential aspect of reflective thought has sparked interest in psychological and knowledge

representation circles, both of which apply the prefix *meta-* to the resultant theories and systems ([Smith85]).

A computational system is said to have a reflective architecture if it incorporates structures representing aspects of its own structure and behavior. The object-level system solves problems in the (external) problem domain. The self-representation at the reflective level makes it possible for the system to answer questions about and to support actions on its object-level system.

Though metareasoning and reflective reasoning are similar, Maes draws a distinction between them. Metareasoning may be implemented in a different language than the object-level system and may have only static access to object-level system. In Maes' view a reflective metalevel must be implemented in the same language as the object-level system and must have dynamic access to the object-level system ([Maes88]).

Much research into reflective architectures has dealt with constructing general purpose reflective programming languages or theorem provers. For example Maes describes the virtual infinite tower of circular interpreters implemented in most reflective languages, required to permit concepts like metametaobjects ([Maes88]). And Aiello and Levi describe the deductive apparatus that is necessary at both the object-level and metalevel for exportation of results from one level to the other ([Aiello88]). These architectural concerns have little bearing on the more limited problem pursued in this thesis.

However Maes presents properties of a reflective architecture for an object-oriented language, some of which do provide useful principles for the problem in this thesis ([Maes87]). We consider three basic architectural questions that must be resolved to explain frames, and whether Maes' answer is appropriate.

The most basic question that must be resolved is where to store metainformation for explanations about the expert system's objects that model the real world. Maes argues for maintaining a disciplined split between the object-level and metalevel. Her solution is to associate a metaobject with every object-level object. The proposal in the next chapter adopts Maes' solution, rather than alternative solutions such as storing that metainformation inside the same object, in metaslots or metafacets.

Slot-changing agents—rules, methods, etc.—also require meta-information for explanations. The second question is where should it be stored? Maes suggests that a self-representation should be uniform. If all entities (instance objects, class objects, slot-changing agents) are objects, they can all be reflected upon. Since we wish to explain all objects, the proposal adopts this suggestion as well.

Given that each object-level object will have an associated metaobject, it is necessary to decide where to draw the line between objects and metaobjects. Maes argues for a self-representation being complete, i.e. metaobjects should contain all the information about objects. This seems to leave object-level objects just a **Value** facet, contrary to the case in most expert system shells. Since the metaobjects proposed here are for explanation only, not general purpose metareasoning, this suggestion seems less important. The convention adopted is that all metainformation for explanations is stored in metaobjects; facets not used in explanations are stored in object-level objects.

Maes has one other important observation to contribute: if a self-representation is causally connected to the aspects of the system it represents, then the self-representation is always accurate. The weakness of canned text is an example of a self-representation that is not causally connected to its system, which explains the manual maintenance problem it presents. The

thesis proposal attempts to have as much as possible of the information stored in explanation metaobjects be causally connected to the object-level objects.

Some of the principles suggested by Maes for reflective systems are helpful in deciding on an architecture for explaining hybrid expert systems. Summarizing the two key principles this proposal adopts: all object-level entities are first class objects and have an associated metaobject, and the metalevel contains all the information needed to explain object-level objects.

3.6 THE PROBLEM RESTATED

Hybrid shells permit rule- and frame-based processing to be as tightly integrated as a given object-level problem demands. The thesis' problem is to define a hybrid explanation facility that is equally integrated, affording frames the same level of explanatory power Mycin made standard for rules. This means that HOW? and WHY? queries must be answerable for all ways that frames's slots are computed—whether by initialization behavior, by the frame inheritance mechanism, or by any slot-changing agent. Ideally the problem's solution should also accommodate other kinds of explanations, and indeed a mechanism supporting WHAT-IS-IT? explanations is taken on as part of the problem in the thesis. The problem is constrained by relying, like Wick and Slagle, on existing technology and knowledge sources.

The problem's solution is hindered by the fact that exploration into related topics uncovered little direct research into frame explanations. Frames' semantics is ad hoc or nonexistent, and the best remedy requires complicating links, i.e. redesigning frames from the ground up. Even that radical solution merely addresses the logic of links, ignoring the logic of

methods. The solution must work around this gap, if methods are to be explained.

Another hindrance: much research into metalevel architectures has been directed toward applications other than explanation, such as control. Furthermore, while that research has enough breadth to indicate alternative archtectures, it has not yet produced enough results to make choices between the alternatives apparent.

The problem can be viewed as reducing into a set of four related subproblems. The brief surveys presented in the preceding three sections have provided some first steps to solutions of three of these four subproblems.

Subproblem #1: What general approach to providing explanations to use? Since it is Mycin's functionality that is being extended, an obvious approach to explore is extending Mycin's mechanisms to explain other schemes besides production rules. This approach has been successfully applied, using metainterpretation, to explain logic programs. The approach requires explicit construction of and access to a goal tree, and constant awareness of what slot-changing agent is acting at any point in time.

Subproblem #2: How are slot-changing agents other than rules to be related to the goal tree? This is the subproblem most neglected by other researchers, and hence is the thesis' most original (and perhaps most controversial) contribution.

Subproblem #3: How to extend the Mycin approach to the entire hybrid architecture? The approach for slot-changing agents is straightforward. Assume the Mycin approach can be recast into object terms, and then can be generalized so as to be applicable to all slot-changing agents. Then, supplying the generalized Mycin capability to generic, first class classes for all slot-

changing agents—i.e. generic classes for rules, methods, demons, user input, and external access interfaces—allows all instances of rules, methods, etc., to inherit the generalized explanation capability. Initialization time values and inherited values must be explainable also.

Subproblem #4: Where to store metainformation for explanation behavior, as opposed to information for object-level behavior? As already indicated, Maes' recommendation for a disciplined split between metaobjects and object-level objects is adopted in this thesis.

This chapter has explicated the problem of explanation in hybrid expert systems. Moving from a discussion of the hybrid architectures to a survey of related research, the chapter has analyzed the problem in sufficient detail to point the way toward the solution presented in the next chapter.

4 A SOLUTION: EXPLAINING HYBRID EXPERT SYSTEMS USING A GOAL TREE OBJECT AND METAOBJECTS

This chapter proposes a solution to the problem of explaining expert systems constructed in a hybrid architecture. The solution extends the Mycin notion of a goal tree for explanations of rules so that it provides equivalent explanations for all ways of determining slots' values. The metaknowledge about objects used to generate explanations is stored in two places: in an **Interface Manager** object and in metaobjects. The four sections that define the proposal describe, first, structure, and then, behavior at three points in time: compile time, execution time, and explanation time. (Of course the last two times are often intertwined in actual consultations.)

The viewpoint of the proposed solution is epistemological, in the sense defined in [McCarthy85]:

> The epistemological part of AI studies what kinds of facts about the world are available to an observer with given opportunities to observe, how these facts can be represented in the memory of a computer, and what rules permit legitimate conclusions to be drawn from these facts. It leaves aside the heuristic problems of how to search spaces of possibilities and how to match patterns.

4.1 OVERVIEW: EXPLAINING HYBRID EXPERT SYSTEMS

The solution proposed here for providing a hybrid expert system shell with explanation capability on a par with Mycin's rule explanations is straightforward, and can be described in two conceptual steps. First, Mycin's explanation mechanism is recast into a frame architecture, i.e. it is *objectified.* In a sense this step just describes a view of the rule-based explanation

component of existing hybrid shells. Second, that mechanism is generalized to provide explanations for the other ways slots values are determined in a hybrid system, namely inheritance, initialization values, and slot-changing agents. To illustrate the flexibility of the solution, a third conceptual step extends the generalization of Mycin's HOW? and WHY? explanation mechanism to enable a third type of explanation: WHAT-IS-IT? explanations tell the user what a slot is and its significance in the expert system.

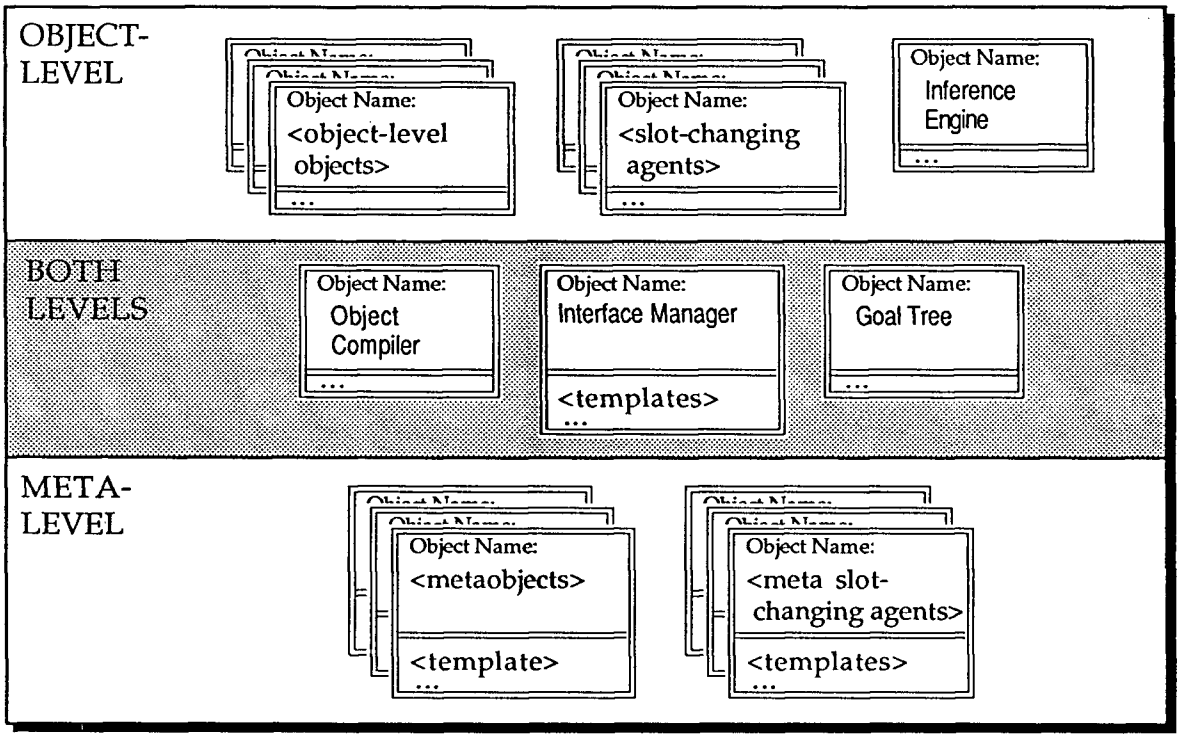

Figure 6: Key Objects in Shell to Explain Hybrid Expert Systems

Figure 6 illustrates the key objects in a shell which exemplifies the solution. The objects in the figure are grouped according to the level of their behavior: object-level performance, metalevel explanation, or a combination of the two. The topic here is explanation, so object-level behavior unrelated

to explanation is described only to the extent necessary to clarify explanation behavior.

Two kinds of objects are illustrated in Figure 6. Four of the objects are distinguished objects supplied by the expert system shell: the **Inference Engine, Object Compiler, Interface Manager,** and **Goal Tree**. Object-level objects, including slot-changing agents, are added by a developer building an object-level expert system using the shell. At run time more object-level objects may be created. The shell creates a metaobject for every object-level object created at development or run time. The distinguished objects do not normally have associated metaobjects, but one or more could, in experimental applications in which it was desirable to explain their (normally background) behavior.

An introduction to Mycin's explanation mechanism for rules is given in section 2.6 and much more detail is provided in several chapters of [Buchanan84b]. Two key structures from Mycin are objectified and generalized in order to explain hybrid expert systems: the goal tree and explanation templates.

The distinguished object, **Goal Tree,** is one cornerstone of this objectification. Its internal structure and behavior for rule-based explanations is explicated here, to the extent required to generalize it for other slot-changing agents. (Its behavior in concert with the inference engine to effect rule-based inference, though important, is not described here, per the previous comments.)

For all slot-changing agents' behavior to be reflected in **Goal Tree,** their behavior must be interpreted in terms of goals. That interpretation is provided at compile time for slot-changing agents by the **Object Compiler**. The result of the interpretation is reflected in the structure and run time behavior of the associated metaobject for the slot-changing agent (*metaagent* for short).

The **Object Compiler**'s interpretation of slot-changing agents supports the crucial reflective behavior in the proposal: the fact that it is causally connected to object-level behavior insures that explanations reflect actual behavior.

Metaobjects and the **Interface Manager** contain the other cornerstone of the proposal: templates. The **Interface Manager** is the only object that interacts directly with the user and when the user asks for an explanation, the **Interface Manager** fills out a template specific to the question asked. Filling out its template typically involves accessing the **Goal Tree** and other templates stored in metaobjects. As indicated in Figure 6, all metaobjects include one or more templates for explanation.

In this proposal, as in Mycin, templates are supplied by the developer at development time. Should the developer decline to do so, the metaobject supplies default generic substitutes at explanation time, e.g. source code for rules or methods, or simply slot names. Relying on user-supplied templates is admittedly a weakness of the solution, since they are not causally connected to object-level behavior and they require effort to produce understandable text.

This section has introduced in a general way objects to explain hybrid expert systems. The next four sections provide more detailed descriptions of those objects' structure and behavior.

4.2 OBJECT STRUCTURES FOR EXPLAINING HYBRID EXPERT SYSTEMS

This section looks more closely at the internal structure of the objects introduced in the previous section. In some cases examples are presented, but more often the reader is directed to chapter 5, where extended examples are presented in the context of actual explanations.

The **Interface Manager** Object

As already mentioned, the **Interface Manager** object handles all interactions between an expert systems and its users. One aspect of that interaction is providing explanations. The **Interface Manager** has a method and template for each explanation it can supply. The behaviors of the methods are described in section 4.5. The templates are illustrated in chapter 5: Figures 26, 52, 61, and 65 for HOW? explanations; Figure 33 for WHY? explanations; and Figure 69 for WHAT-IS-IT? explanations.

The **Interface Manager** also includes two methods, **Ask User Method** and **Ask User If Unknown Method**, which it uses whenever a slot-changing needs information from the user. These methods' behaviors, important for explanations involving user input, are described in section 4.4.

The **Goal Tree** Object

The **Goal Tree**'s structure supports both the object-level behavior of the inference engine and explanations about that behavior. The **Goal Tree** is a collection of **Goal Tree Node** objects, all related by supergoal-subgoal relationships to the root node of the tree. The frame system's IS-A links are not the appropriate mechanism to express those relationships, since the goal tree relationships do not represent either superclass-subclass or class-instance relationships. Instead each **Goal Tree Node** contains two slots, **Supergoals** and **Subgoals**, that represent the tree structure of the **Goal Tree**.

Each **Goal Tree Node** represents information at a point in time when some slot-changing agent fired, i.e. set a slot's value. (In chapter 6 a broader definition of "firing" is considered, to enable other explanation queries.) The

Goal Tree Node slots that represent this information about the expert system's object-level behavior are: **Agent, Goal Slot, Trigger Slots,** and **Input Slots.**

Metaobjects for Slot-Changing Agents

Each slot-changing agent, with one exception, has a metaobject associated with it. (The exception, user queries, is handled by the **Interface Manager** in an ad hoc fashion.) For brevity, we call a slot-changing agent simply an *agent* and an associated metaobject simply a *metaagent*. A metaagent consists entirely of template slots that are involved with producing explanations of the agent's behavior. There are four templates altogether. One, the **Goal Tree Template,** contains information used to populates nodes in the **Goal Tree.** The other three we refer to collectively as the metaagent's *explanation templates.*

The metaagent's **Goal Tree Template** is a complex list containing a sublist for each goal slot the agent potentially solves, or computes. Each sublist makes explicit the slots which are crucial to that portion of the agent's behavior which computes that one goal slot's value. The slots which appear in conditions that govern the goal slot's computation are the *triggers* for that portion of the computation. Those slots which are used in the computation of the goal slot's value are its *inputs.*

The explanation templates for a metaagent are combined by the **Interface Manager** to produce a textual explanation for the portion of the agent's behavior that solves one goal (this explains the agent completely if it only solves one goal). Like the **Goal Tree Template,** each of the explanation templates is a complex list indexed by the goals the agent solves.

Three templates comprise the metaagent's explanation templates. The **Explanation Template for Consequent** paraphrases the setting of a goal slot's value.

The **Explanation Template for Premise** similarly paraphrases the conditions which govern whether the consequent behavior occurs, e.g. the premise of a rule or the invocation condition of a demon. The **Connector Template** is a syntactic device to connect the explanations of consequent and premise with text appropriate to the type of explanation.

The **Goal Tree Template** and explanation templates in a metaagent are closely related to the parsing of its object-level agent by the **Object Compiler.** The templates' structure is clarified further in the discussion of compile time behavior in section 4.3.

Metaobjects for Other Object-Level Objects

The structure of metaobjects for other object-level objects—i.e. excluding slot changing agents—is simply described: they have the same slots as their object-level object. The metaobject's slots, however, have different facets, that support explanations rather than object-level behavior. A sample object and its corresponding metaobject are illustrated in Figure 7.

For example instead of a **Value** facet, metaobject slots have a **History List** facet. The **History List,** as its name implies, maintains a record of all the values that the object-level slot acquires during a consultation. Each value in the **History List** is indexed to the **Goal Tree Node** that was created when that value was set. The incremental building of the **History List** facet at run time is described in section 4.4 and illustrated in the example in section 5.1.

Each slot in the metaobject has a **Translation** facet, a template which supplies the textual description of the object-level slot. The **Translation** facet is supplied by the developer. It is discussed further in section 5.1.

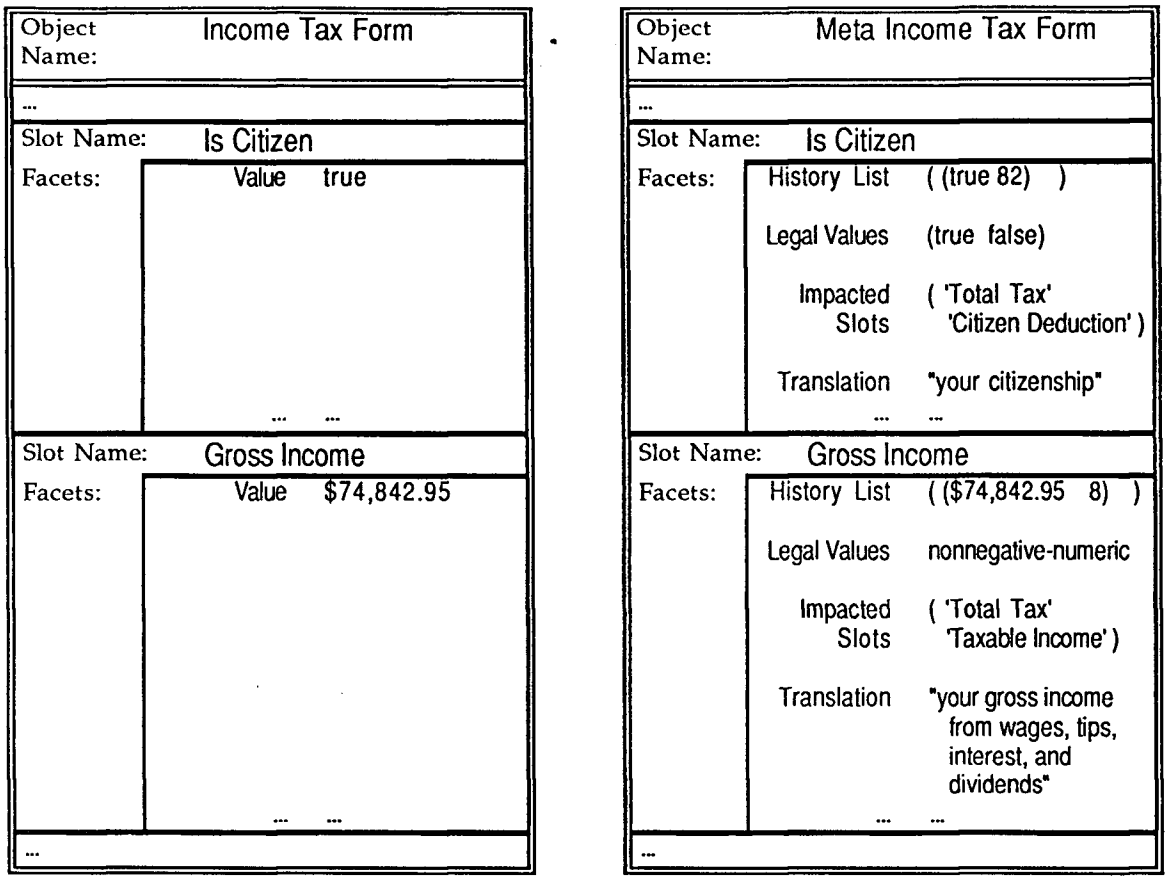

Figure 7: Sample Object-Level Object and Its Metaobject

In section 2.4 the **Legal Values** facet was introduced as a facet in object-level slots (as is the case in many shells). Since it has a role to play in WHAT-IS-IT? explanations, it is included as a metaobject facet in this proposal. The **Legal Values** facet is supplied by the developer.

The **Impacted Slots** facet for a slot (call this slot **Slot-x**, for clarity) is a list of slots that are impacted by **Slot-x**. That is, in at least one computation of an impacted slot, **Slot-x** is either a trigger or an input for the computation. The **Impacted Slots** facet is built incrementally at compile time by the **Object Compiler**.

Figure 7 compares a sample object-level object with its corresponding metaobject.

This completes the detailed look at the internal structure of objects in the proposed hybrid expert system shell that will support explaining the behavior of an expert system coded in that shell.

4.3 BEHAVIOR AT COMPILE TIME FOR EXPLAINING HYBRID EXPERT SYSTEMS

Important behaviors for explaining hybrid expert systems occur at compile time. The shell's **Object Compiler** compiles objects defined by the developer, checking for inconsistencies in objects' specification. If the compilation finds no errors, then other object-level actions are performed, e.g. implementing specified inheritance relationships, followed by metalevel actions to support explanation.

Figure 8 illustrates the behavior of the of the **Object Compiler** when compiling a slot-changing agent. Five behaviors for explanation occur in this context:

(1) the agent is parsed into units, each of which computes a goal, either unconditionally or conditionally, and the parsing is reflected in the agent's compiled form (①);

(2) the use of methods that ask the user for information is restricted in the agent (①);

(3) the **Impacted Slots** facets in metaobjects are updated (③);

(4) the metaagent for the slot-changing agent is created and its **Goal Tree Template** constructed (②);

(5) the metaagent's explanation templates are constructed (②).

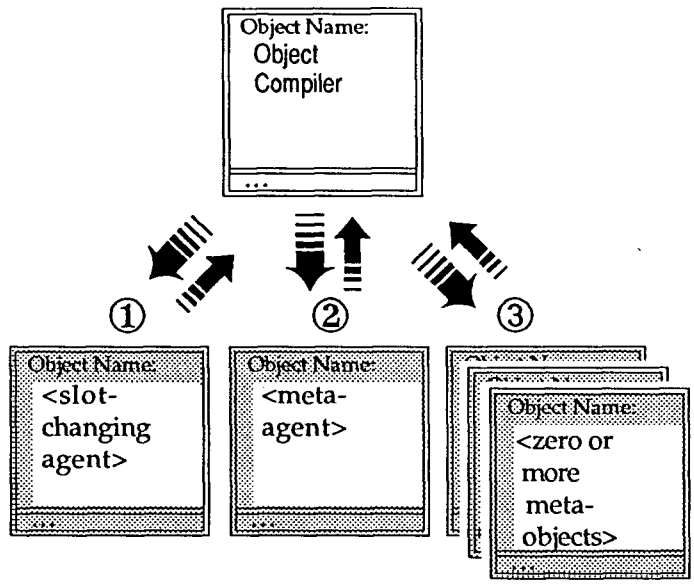

Figure 8: **Object Compiler** Behavior Supporting Explanations

Parsing Slot-Changing Agents into Goal Units

It is straightforward for the **Object Compiler** to parse a slot-changing agent into goal units based on the syntax of the agent's specification. The goals of rules and methods, including demons, are simply the target slots of assignment statements in their respective syntaxes. The goal of a user query is obviously the slot queried for. And the goals of external access are the slots which get populated from the external source.

The **Object Compiler** retains the parsing into goal units as it stores the slot-changing agent in its internal compiled form. The exact nature of that internal form is not important for explanation, just that it retain the explicit parsing of a method into goals.

Object Name:	Accrue Deferred Interest Rule
Inheritance:	Is Instance Of Rules
Slot Name:	Premise
Facets:	Value 'PRP Coverage' ... ...
Slot Name:	Consequent
Facets:	Value (set-slot 'Deferred Interest' ; TO THE VALUE (+ ('Savings Account Interest ' 'Savings Bonds Interest' 'Personal Retirement Plan (PRP) Interest'))) ... ...
Slot Name:	Compiled Code
Facets:	Value (('Deferred Interest' (if 'PRP Coverage' (set-slot 'Deferred Interest' (+ ('Savings Account Interest ' 'Savings Bonds Interest' 'Personal Retirement Plan (PRP) Interest'))))) ... ...
...	

Figure 9: A Rule Parsed into One Goal Unit

Figures 9 and 10 illustrate two slot-changing agents and their parsing into goal units[3]. Figure 9 is a simple rule that sets just one goal. Figure 10 is more complex, a method that is parsed into three goal units.

[3]The Lisp-like object-level language in these illustrations is used only for conciseness. It is not meant to represent any real hybrid shell's language, nor is it claimed to be syntactically correct Lisp code. When it is expedient to do so, object-level "code" is mixed with metalanguage; the latter is usually bracketed <in this manner> and distinguished by font from the object-level code.

Object Name:	Compute Total Tax Method
Inheritance:	Is Instance Of Methods
Slot Name:	Source Code
Facets:	Value ((set-slot 'Base Tax' ; TO THE VALUE (* ('Taxable Income ' 'Tax Rate'))) (if 'Is Citizen' (set-slot 'Citizen Deduction' $3000.00) ; ELSE (set-slot 'Citizen Deduction' $0.00)) (set-slot 'Total Tax' (* (- 'Base Tax' 'Citizen Deduction') 'Legislative Factor'))) ... ...
Slot Name:	Compiled Code
Facets:	Value (('Base Tax' (set-slot 'Base Tax' (* ('Taxable Income ' 'Tax Rate')))) ('Citizen Deduction' (if 'Is Citizen' (set-slot 'Citizen Deduction' $3000.00) (set-slot 'Citizen Deduction' $0.00))) ('Total Tax' (set-slot 'Total Tax' (* (- 'Base Tax' 'Citizen Deduction') 'Legislative Factor')))) ... ...
...	

Figure 10: A Method Parsed into Three Goal Units

Restricting Methods That Ask the User

There are two sets of restrictions on asking the user for information that the **Object Compiler** enforces in order to enable explanations about those queries.

The first restriction ensures that methods used to ask the user are explainable. The second ensures that the **Object Compiler** will be able to determine why the user is being asked, i.e. the supergoal of the query.

For the shell to explain user queries, all methods to implement those queries must ultimately utilize the **Ask User Method**. The shell itself uses that method for queries that result due to backward chaining rule inference. The **Ask User If Unknown Method** supplied by the shell also uses it. It is the responsibility of the **Object Compiler** to prohibit a slot-changing agent from posing queries to the user that do not rely on **Ask User Method**. (Obviously **Ask User Method** must be robust enough to permit a variety of means of user input, e.g. menu selection vs. keyboard entry; but this is an implementation detail we need not consider.)

To enable the **Object Compiler** to determine why the user is asked for information, the **Object Compiler** restricts the placement of user queries to occur immediately preceding an unconditional or conditional assignment in which the slot sought by the query is either a trigger or an input to the assignment. The restriction enables the **Object Compiler** to determine the supergoal of the user query: it is the goal of the subsequent assignment statement.

Updating **Impacted Slots** Facets in Metaobjects

When a user asks WHAT-IS-IT? about a slot, it is desirable to respond with more than just a template, which is not causally connected to system behavior. One possibility is to explain how the slot is used in the expert system. The **Impacted Slots** facet in metaobjects stores information about a slot's use. The information in the facet is generated incrementally as slot-changing agents are compiled, and hence is causally connected to system behavior.

The parsing of slot-changing agents produces the information the **Object Compiler** stores in **Impacted Slots** facet. For each goal that the **Object Compiler** finds in a slot-changing agent, it updates the **Impacted Slots** facets for the slots that are triggers and/or inputs to the computation of that goal.

As examples, consider the rule and method illustrated in Figures 9 and 10. Their compilation results in the updates to **Impacted Slots** facets shown in Figure 11. As the **Object Compiler** parses the rule, it adds the rule's only goal slot, **Deferred Interest**, to the **Impacted Slots** facets of four slots: **PRP Coverage** because it is a trigger, and the other three because they are inputs to the computation. In similar fashion the compilation of the method adds its three goals to the **Impacted Slots** facets of trigger and input slots. The behavior of the **Object Compiler** in updating metaobjects' **Impacted Slots** is described in more detail in section 5.6.

When this slot-changing agent is parsed	this goal slot	is added to this slot's Impacted Slots facet because it is a trigger	and is added to these slots' Impacted Slots facet because they are inputs
Accrue Deferred Interest Rule	Deferred Interest	PRP Coverage	Savings Account Interest Savings Bonds Interest Personal Retirement Plan (PRP) Interest
Compute Total Tax Method	Base Tax		Taxable Income Tax Rate
	Citizen Deduction	Is Citizen	
	Total Tax		Base Tax Citizen Deduction Legislative Factor

Figure 11: Agent Compilation Updates **Impacted Slots** Facets

Constructing the **Goal Tree Template** in Metaagents

Although its structure appears complex at first, the **Goal Tree Template** in the metaobject associated with a slot-changing agent actually comes rather straightforwardly from the compilation process. The **Object Compiler** simply packages up the information about each goal's computation and its triggers and inputs into a sublist, then concatenates the sublists for all the agent's goals to produce the **Goal Tree Template**. Figures 12 and 13 illustrate **Goal Tree Templates** for the rule and method that are shown in Figures 9 and 10, respectively.

Object Name:	Meta Accrue Deferred Interest Rule
Inheritance:	Is Instance Of MetaRules
Slot Name:	Goal Tree Template
Facets:	Value (('Accrue Deferred Interest Rule' 'Deferred Interest' ('PRP Coverage') ('Savings Account Interest ' 'Savings Bonds Interest' 'Personal Retirement Plan (PRP) Interest'))) ... ...
...	

Figure 12: A Rule's **Goal Tree Template**

Object Name:	Meta Compute Total Tax Method
Inheritance:	Is Instance Of Meta Methods
Slot Name:	Goal Tree Template
Facets:	Value (('Compute Total Tax Method ' 'Base Tax' nil ('Taxable Income ' 'Tax Rate')) ('Compute Total Tax Method ' 'Citizen Deduction' ('Is Citizen') nil) ('Compute Total Tax Method ' 'Total Tax' nil ('Base Tax' 'Citizen Deduction' 'Legislative Factor'))) ... ...
...	

Figure 13: A Method's **Goal Tree Template**

Constructing the Explanation Templates in Metaagents

The internal list structure of the explanation templates for metaagents is the same as the **Goal Tree Template**'s. That is, each one of the individual templates—**Explanation Template for Consequent, Explanation Template for Premise,** and **Connector Template**—is a list consisting of sublists indexed by the goals the slot-changing agent solves. Thus the **Object Compiler** easily builds the framework for the explanation templates at compile time. It then notifies the shell's developer interface to prompt the developer to fill in the framework with valid, readable templates.

Making templates produce readable, meaningful text is no easy task. It is reasonable to conjecture that the shell's developer interface, in concert with the **Object Compiler** could assist the developer in that effort. Metatemplates for

various configurations of slot-changing agents could supply reasonable initial explanation templates, which the developer could then tailor for readability.

This thesis has little to offer toward the engineering of valid, readable explanation templates. It simply assumes that they can be created. Evidence of that possibility are the templates presented in the examples in chapter 5: Figures 27, 34, 40, 43, 47, 50, 56, and 58. Given that assumption, the thesis proposal provides a mechanism for combining explanation templates with the run time information in the Goal Tree to produce valid explanations.

This section has described the behavior of the Object Compiler as it compiles slot-changing agents. The crucial concept is the parsing of those agents into goal units. The goal units are reflected in the compiled form of agents and in templates stored in metaagents. The parsing supplies the information required to produce WHAT-IS-IT? explanations. The next section illustrates how the parsing is reflected in run time behavior to support explanations.

4.4 BEHAVIOR AT RUN TIME FOR EXPLAINING HYBRID EXPERT SYSTEMS

Previous sections have described object structure and compile time behavior to support explaining hybrid expert systems. This sections describes the run time behavior toward that end. To do so it describes four sets of behaviors:

(1) the **Inference Engine**'s behavior upon initialization for a consultation,

(2) the behavior for user queries,

(3) metaagents' behavior when their object-level slot-changing agent fires, and

(4) the **Goal Tree**'s behavior as it adds a new node to the tree.

Initializing the Knowledge Base for a Consultation

At initialization, the **Inference Engine** initializes the **Goal Tree** and all metaobjects. A root node is created for the **Goal Tree**, with a **Node Index** value of **1**. Then the **Inference Engine** examines each object-level object and initializes the **History List** facet of its corresponding metaobject. If an object-level slot's value is undefined at initialization time, its metaobject slot's **History List** facet is set to **nil**. If the slot's value is defined at initialization time, the **History List** facet's value is set to the list containing one order pair: ((<slot value> **1**)). This behavior enables HOW? explanations of initialization time values, as described in section 5.5.

Asking the User for Information

During processing the **Inference Engine** interacts with the **Goal Tree** and slot-changing agents to effect object-level behavior. Occasionally user input will be appropriate; for example a method may include **Ask User Method** messages, or rule-based inference may decide that a query is an appropriate way to satisfy a goal during backward chaining. In any case the **Inference Engine** directs the **Goal Tree** to ask the **Interface Manager** to query for the information. As noted earlier, the query is always ultimately handled by the **Ask User Method**, regardless of the slot-changing agent that generates it.

If the user answers the query (including the case where the answer is "unknown"), the **Ask User Method** behaves like any slot-changing agent, as

described in the next section. If the user asks WHY? instead of answering, the behavior is that described in section 4.5.

Behavior of Metaagents when Their Object-Level Agent Fires

In this thesis a limited definition of firing for slot-changing agents is adopted: we say an agent fires when it sets a slot's value. (Agent firing can be viewed more broadly, as discussed in chapter 6.) Under the limited definition adopted here, if a rule fires, all the slots assigned in its consequent are set. But a method could be activated, and not actually set any slots, if they were all assigned conditionally and those conditions were not met; in that case we would not say that the method had fired, since no slot-changing behavior would occur. (Thus firing and being activated are not synonymous for methods.) Similarly we might envision an external access that attempts, but fails, to set one or more slots' values, but we consider it to fire only if it actually succeeds in setting some values.

Figure 14 is a schema that illustrates the behavior of slot changing agents as they fire. When an agent fires, it updates one or more slots in one or more object-level objects (①); this is the object-level behavior of the expert system). As it sets a goal slot, the agent also sends a message to its metaagent (②). The metaagent in turn updates the **Goal Tree** (③) and the metaobjects of the object-level objects being changed (④).

In its message to **Goal Tree**, the metaagent sends along the **Goal Tree Template** for the goal slot being set. The **Goal Tree** then stores the template of information about the change in the knowledge base's state in the new **Goal Tree Node** it creates.

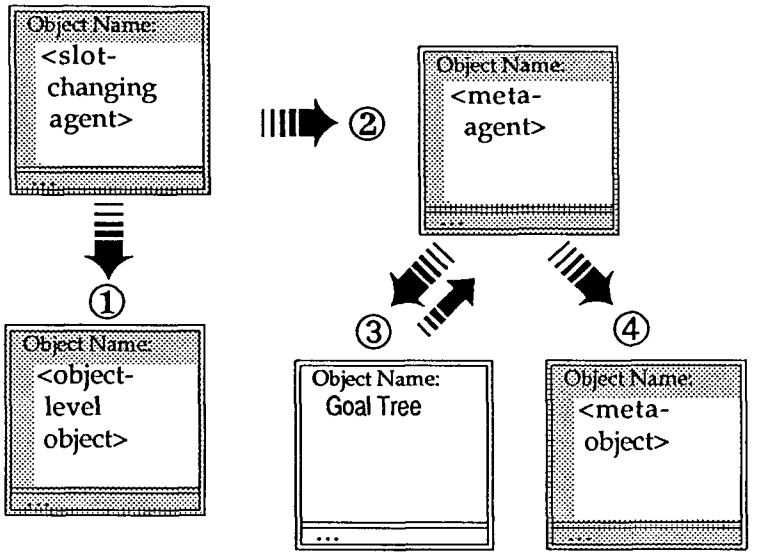

Figure 14: A Slot-Changing Agent Stores Information for HOW? Explanations

The **Goal Tree** assigns a **Node Index** when it inserts a new node. These **Node Index** values are always assigned in ascending order, providing an explicit temporal ordering of **Goal Tree** events. Since the **Goal Tree** reflects all object-level changes in the knowledge base, that temporal ordering applies to the knowledge base as a whole. The ordering is used for retrieving values from slots' **History List** facets when producing HOW? explanations.

The **Goal Tree** returns to the metaagent the **Node Index** of the node it just added to the tree. The metaagent then sends a message to the metaobject of the goal slot that changed. That message includes the new value of the goal slot and the **Node Index** where that change was reflected in the **Goal Tree**. The metaobject inserts the ordered pair (<new slot value> **Node Index**) at the head of the slot's **History List** facet.

The behavior of agents and metaagents at run time is illustrated in depth for a rule in section 5.1, and more generally in section 5.3. In the latter

section, Figure 37 reproduces Figure 14, and the schema is discussed in the context of the examples presented in that section.

Adding a Node to the **Goal Tree**

Besides maintaining the temporal ordering of its nodes, the **Goal Tree** maintains its internal structure, i.e. the supergoal-subgoal relationships of the nodes. For backward chaining rule inference, the **Goal Tree** behaves exactly as Mycin's goal tree mechanism. Since other slot-changing agents are not explicitly goal-directed like backward chaining, the **Goal Tree** must infer appropriate supergoal-subgoal relationships.

The **Goal Tree** has a general mechanism for inferring supergoal-subgoal relationships for forward chaining rules, demon activation, and linear procedural code in methods. The mechanism proceeds in two steps for all agents except rules being used for backward chaining.

(1) In the first step, **Goal Tree** adds a new **Goal Tree Node** with its **Supergoal** slot pointing to the root node of the tree and a Subgoal slot that is **nil**; the **Subgoal** slot of the root node is of course updated to reflect its newly added subgoal.

(2) The **Goal Tree**'s second step scans all the slots mentioned in the **Trigger Slots** and **Input Slots** of this newly added node. If any of those slots are the goal of a node that is a subgoal of the root node, the relationships are modified so that that node becomes a subgoal of the newly added node. (Should there be more than one node for the same goal slot attached to the root node, the one with the higher index would be modified.) This modification of the tree requires updates to the **Subgoal** slot of the root node, the **Subgoal** slot of the newly added node, and the **Supergoal** slot of the modified node.

Figure 15 illustrates a portion of the **Goal Tree** that results from the application of this mechanism during the execution of the method shown in Figure 10.

Note that the links in Figure 15 are conceptual supergoal-subgoal links (top to bottom), not the frame hierarchy's IS-A links.

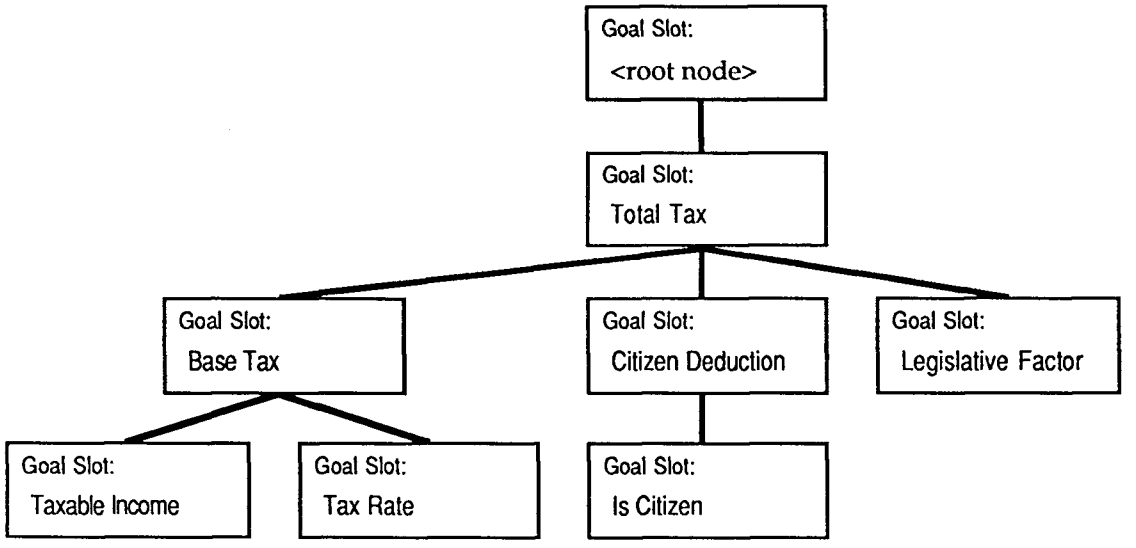

Figure 15: Portion of **Goal Tree** Instantiated by Execution of a Method

This completes the discussion of run time behavior to support explanation in a hybrid expert system. Compile time behavior has established information to answer WHAT-IS-IT? queries, while the run time behavior supports HOW? explanation, via the trace of object-level behavior stored in the **Goal Tree** and metaobjects' **History List** facets. The final section in this chapter describes the mechanism to produce the explanations.

4.5 BEHAVIOR AT EXPLANATION TIME FOR HYBRID EXPERT SYSTEMS

This section describes how the structure and behavior described in the previous three sections combine to supply the explanations we desire. The **Interface Manager** has a method and template for each explanation type it can handle, HOW?, WHY?, and WHAT-IS-IT?. For each type, its relevance to

different slot-changing agents, the explanation method's overall strategy, and the method's specific behavior are described.

HOW? Explanation Behavior

HOW? explanations are the most generally applicable of the three types presented here. To explain a hybrid expert system completely, not only slot-changing agents, but also inherited values and initialization values, must be explained.

The general strategy of the **Interface Manager's HOW? Method** is to explain the mechanism that set the current value of a slot. If that slot has been changed by one or more slot-changing agents, the method explains the last slot-changing agent that set the value of the slot.

The **HOW? Method** contains code to detect and explain initialization and inherited values. For a slot populated by an initialization time value, the method can both detect the condition and fill out its simple template with a single message to the slot's metaobject. For a slot with an inherited value, the **HOW? Method** must access the slot's metaobject and object-level object to detect the inheritance, and also must access the superclass' metaobject to complete its **HOW? Explanation Template**. Examples of explanation behavior for both conditions are given in section 5.5.

The **HOW? Method**'s behavior to explain slot-changing agents is illustrated by the behavior schema in Figure 16. The method must access information in the metaobject of the slot the HOW? query is about, to determine from the slot's **History List** what node in the **Goal Tree** reflects the last change in value for the slot (②). It then accesses the **Goal Tree Node** to discover what slot-changing agent was responsible for the change (③). Its next access is to that agent's

metaagent, since the method uses the metagent's explanation templates to fill out its **HOW? Explanation Template** (④). Finally the method may need to access the metaobjects for objects mentioned in the metaagent's explanation templates, i.e. inputs and/or triggers (⑤). Examples of HOW? explanation behavior are presented in section 5.1 for rules; section 5.3 for methods, including demons; and section 5.5 for external access.

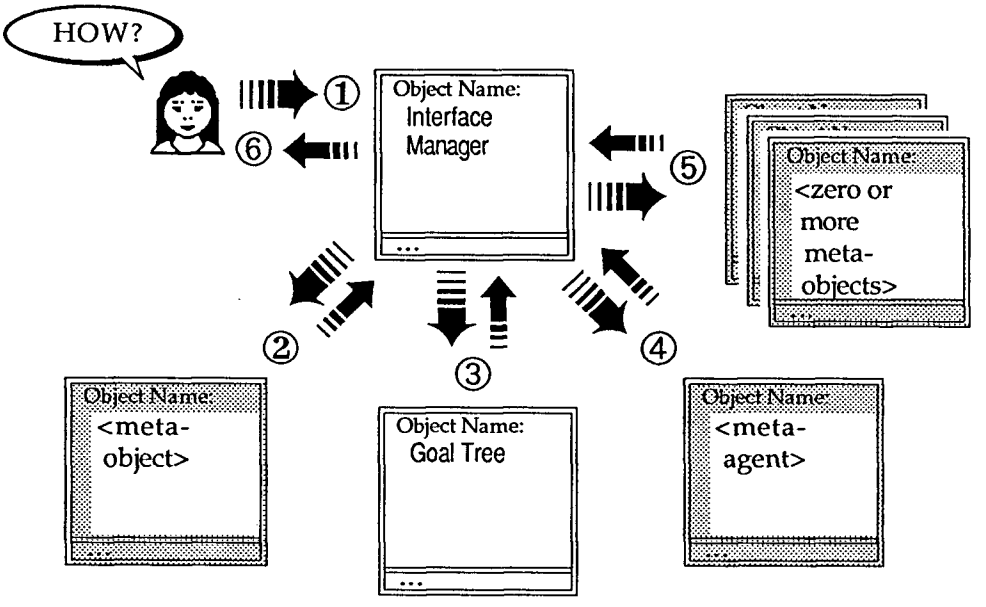

Figure 16: HOW? Explanation Schema for Slot-Changing Agents

HOW? explanation behavior is simpler for slots whose values were last set by user input. The **HOW? Explanation Template** detects user input values from **Goal Tree Node** information, and supplies an explanation requiring no further access to any metaobjects. An example of a HOW? explanation for user input appears in section 5.5.

WHY? Explanation Behavior

A WHY? explanation is in response to a slot-changing agent requesting the **Interface Manager** to query the user for a slot's value. Rather than answering, the user asks WHY? the expert system wants to know the information. The general strategy of the **Interface Manager's WHY? Method** is to explain the slot-changing agent that is requesting the value of the slot from the user. Initialization values, inheritance, and user input never generate user queries, thus WHY? explanations do not occur in those contexts.

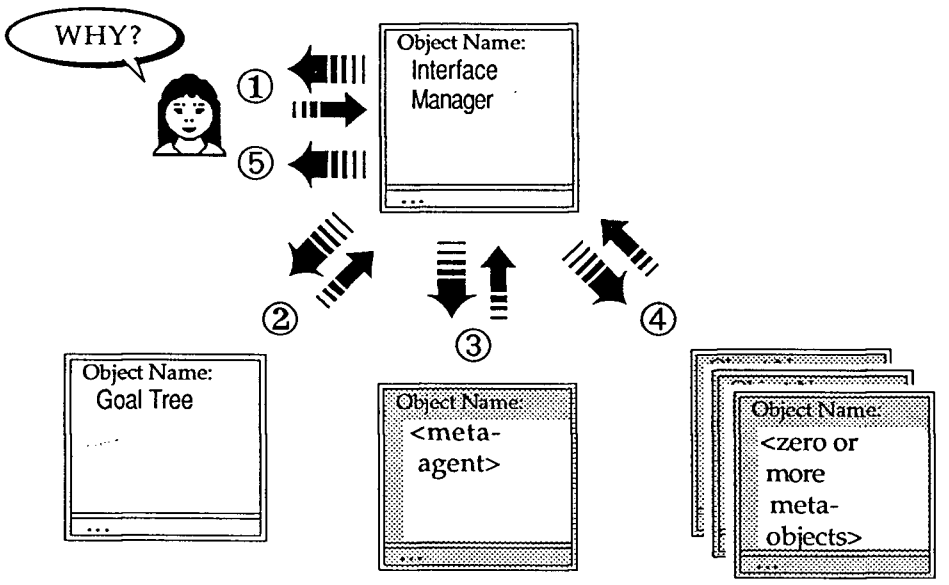

Figure 17: WHY? Explanation Schema

The general strategy of WHY? explanations looks very similar to that for HOW? explanations. Indeed the two are mainly differentiated by a viewpoint in time; the WHY? explains the agent's behavior before it has been done, while the HOW? explains the same behavior after it has occurred. Thus the

messages that the **WHY? Method** sends to access information follows a pattern that is very similar to that just described for HOW? explanations, as is evident from Figure 17. Examples of WHY? explanation behavior are presented in section 5.2 for rules; section 5.4 for methods, including demons; and section 5.5 for external access.

WHAT-IS-IT? Explanation Behavior

WHAT-IS-IT? explanations are different from HOW? and WHY?, in that WHAT-IS-IT? explanations describe a slot, rather than a slot-changing agent. The general strategy of the **Interface Manager's WHAT-IS-IT? Method** is to explain a slot by supplying three facets from the slot's metaobject: its **Translation, Legal Values,** and **Impacted Slots.**

Thus the behavior schema for WHAT-IS-IT? explanations, illustrated in Figure 18, is quite simple. The **WHAT-IS-IT? Method** sends a message to the slot's metaobject, and zero or more additional messages to the metaobjects of that slot's **Impacted Slots.** Section 5.6 illustrates the generation of a WHAT-IS-IT? explanation.

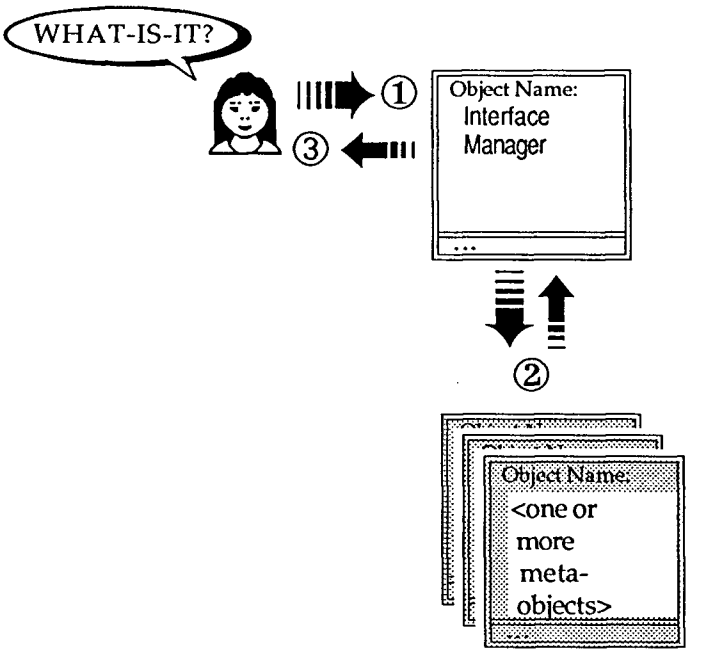

Figure 18: WHAT-IS-IT? Explanation Schema

4.6 CHAPTER RECAP

This chapter presents a partial specification for functionality to explain hybrid expert systems. In that specification Mycin's goal tree mechanism and explanation templates are objectified: the goal tree mechanism becomes a **Goal Tree** object, and explanation templates become slots in metaobjects that are associated with slot-changing agents and other object-level objects. The Mycin mechanisms are generalized so to apply to methods, demons, user input, and external access as well as to rules. The solution is further generalized to explain slots whose values are defined at initialization time and never change, as well as slots that inherit values from a superclass in the IS-A hierarchy. Finally, to illustrate the flexibility of the resulting hybrid

system explanation solution, it is extended beyond Mycin's HOW? and WHY? explanations to answer WHAT-IS-IT? for slots.

The descriptions of structure and behavior presented in this chapter are general but terse. Extended concrete examples in the next chapter illustrate the structure and behavior in the context of actual explanations.

5 SAMPLE EXPLANATIONS AND THEIR GENERATION

This chapter illustrates explanations from an expert system built within a shell designed with the structure and behavior described in sections 4.1 through 4.5. The vehicle for the illustration is a mythical expert system built on a mythical shell. We examine how such an expert system would handle explanations for a progression of cases.

In the next two sections, we consider the cases of HOW? and WHY? questions for rules. In so doing, Mycin's goal tree mechanism is recast, as a **Goal Tree** object. These illustrations are rather detailed, in an attempt to convince the reader that the explanation capability of Mycin has been correctly recast, or objectified.

Moving beyond a straight objectification of Mycin functionality, sections 5.3 and 5.4 extend the illustration of the object-based Mycin adaptation. The extension provides answers to HOW? and WHY? questions for some other agents besides rules that set slot values, namely ordinary methods and demons. These illustrations are not so detailed, because of the commonalities between the approach used for rules and the approach for methods and demons.

Section 5.5 extends the framework to all other ways a slot's value can be determined, illustrating HOW? explanations for user input, external access, inheritance, and initialization time values. WHY? explanation is illustrated for external access, the only one of those four ways for which it is meaningful.

The final section in this chapter demonstrates that the specification can easily support explanations other than HOW? and WHY? The query depicted is WHAT-IS-IT?, as in "What is this slot?"

For the illustrations we observe the mythical Ms. Maria Doe as she uses for the first time her expert system, Ex-Tax. Ex-Tax calculates her national income tax, and Doe has been told that it is the best system for her nation's byzantine tax codes. But this is the first time she has not used a human accountant, and she wants some assurance that Ex-Tax is making the best decisions for her situation.

5.1 HOW? EXPLANATION FOR A RULE

At some point Doe sees that Ex-Tax (ET for short) has filled in the space on her income tax form for Deferred Interest, and she asks HOW? the value in that space (which of course is a slot in the **Income Tax Form** object) was calculated. ET's response is shown in Figure 19.

The value of **Deferred Interest**, **$1045.24**, resulted from the
application of the **Accrue Deferred Interest Rule**,
which computed it as the sum of interest from three sources:
Savings Account Interest ($504.74),
Savings Bonds Interest ($50.47), and
Personal Retirement Plan (PRP) Interest ($490.03).
The computation was done because the following was true:
You are covered by a Personal Retirement Plan (PRP).

Figure 19: Sample HOW? Explanation for a Rule

Let's look behind the user interface to see what has occurred to produce this Mycin-style HOW? explanation for a rule. Consider the **Income Tax Form** object's slot named **Deferred Interest**. At the time Doe asks about **Deferred Interest**, its value is **$1045.24**. Assuming the explanation shown above is accurate, the last agent to set the value of the **Deferred Interest** slot was the **Accrue Deferred Interest Rule**. Let us go back to the moment when the rule (call it the **ADI Rule**) fired.

Figure 20 illustrates the **ADI Rule** as it fired. It sent two messages in the sequence shown in the figure, where they are designated ① and ③. The messages were sent to:

① the **Income Tax Form** object. The **Income Tax Form** method which processed the message changed **Deferred Interest**'s value to the new value just inferred by the **ADI Rule, $1045.24** (②), and returned nothing to the rule object. This action is the object-level consequence of the rule firing; all the information for explanation is accessed from the metalevel—even **Deferred Interest**'s value.

③ the **Meta Accrue Deferred Interest Rule** metaagent object. The message included the name and value of the slot just computed by the rule. The **Meta Accrue Deferred Interest Rule** (let's call it **Meta ADI Rule**) method which processed the message sent two mores messages in response, and returned nothing to the **ADI Rule**.

The **Meta ADI Rule** metaagent sent its two messages (④ and ⑦ in the figure) to:

④ the **Goal Tree** object. The **Goal Tree** method which processed the message added a new node to **Goal Tree**, populated with complete information about this change of the knowledge base's state. The information contained in the newly added **Goal Tree Node** is described later in this section. Adding a new node involved changing the **Goal Tree** object's internal state and sending messages to various **Goal Tree Node** objects (⑤). The method returned to **Meta ADI Rule** the **Node Index** of the new node in **Goal Tree** (⑥). To make our example more concrete, let's assume that the newly added node had a **Node Index** value of **995**.

⑦ the **Meta Income Tax Form** metaobject. The **Meta Income Tax Form** method which processed the message added a new ordered pair to the head of the **History List** facet for slot **Deferred Interest** (⑧). We will look more

closely at this change below. The **Meta Income Tax Form** method returned nothing to **Meta ADI Rule.**

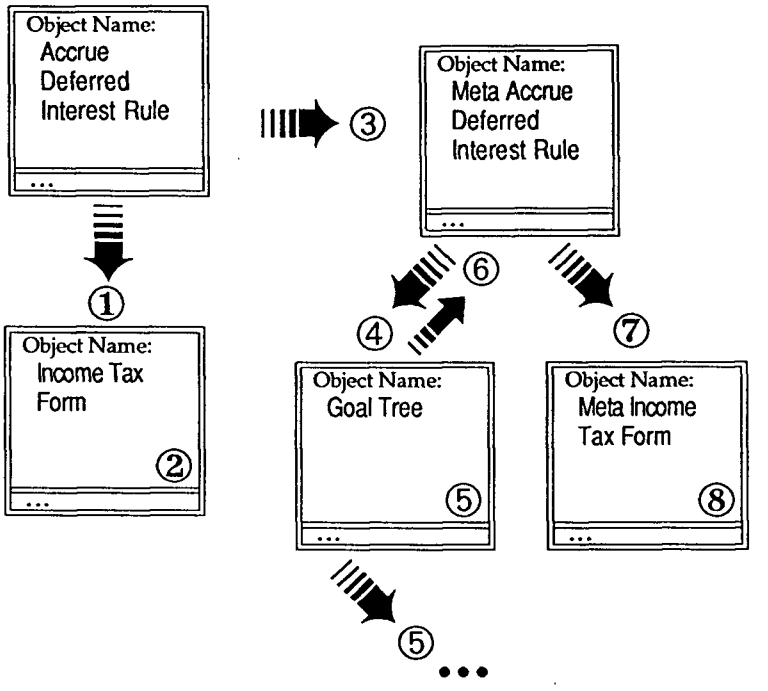

Figure 20: A Rule Stores Information for HOW? Explanations

The information stored in the new **Goal Tree Node** (i.e., at ⑤ in Figure 20) is provided by **Meta ADI Rule.** The rule and its metaagent are depicted in Figures 21 and 22[4]. Figure 21 illustrates that the rule applies (in the contexts in which it is relevant) if the slot **PRP Coverage** has a value of **true.** The consequent follows our expectation, given ET's explanation.

[4]In these examples a notational shortcut is taken for readability: slot names are indicated with no corresponding object name; the implicit object is the **Income Tax Form** object in all cases.

Object Name:	Accrue Deferred Interest Rule
Inheritance:	Is Instance Of Rules
Slot Name:	Premise
Facets:	Value 'PRP Coverage' ... ...
Slot Name:	Consequent
Facets:	Value (set-slot 'Deferred Interest' ; TO THE VALUE (+ ('Savings Account Interest ' 'Savings Bonds Interest' 'Personal Retirement Plan (PRP) Interest'))) ... ...
...	

Figure 21: **ADI Rule**

Object Name:	Meta Accrue Deferred Interest Rule
Inheritance:	Is Instance Of MetaRules
Slot Name:	Goal Tree Template
Facets:	Value (('Accrue Deferred Interest Rule' 'Deferred Interest' ('PRP Coverage') ('Savings Account Interest ' 'Savings Bonds Interest' 'Personal Retirement Plan (PRP) Interest'))) ... ...
...	

Figure 22: **Meta ADI Rule's Goal Tree Template**

Meta ADI Rule passed its **Goal Tree Template** slot to the **Goal Tree** in message ④. The **Goal Tree** in turn stored the information in the newly created **Goal Tree Node 995**. Figure 23 illustrates the slots in **Goal Tree Node 995** that represent the information passed from **Meta ADI Rule's Goal Tree Template**. The information in a **Goal Tree Template** and its subsequent **Goal Tree Node** are redundant, so

remaining illustrations will only show the resultant **Goal Tree Node**. The **Goal Tree Template** was structured when the **Object Compiler** compiled the **ADI Rule** and consequently created or updated **Meta ADI Rule**. The behavior creating the **Goal Tree Template** was described in section 4.3 and is illustrated, with additional detail, in section 5.6.

Object Name:	Goal Tree Node 995
Inheritance:	Is Instance Of Goal Tree Nodes
Slot Name:	Agent
Facets:	Value Accrue Deferred Interest Rule ... ...
Slot Name:	Goal Slot
Facets:	Value Deferred Interest ... ...
Slot Name:	Trigger Slots
Facets:	Value ('PRP Coverage') ... ...
Slot Name:	Input Slots
Facets:	Value ('Savings Account Interest ' 'Savings Bonds Interest' 'Personal Retirement Plan (PRP) Interest') ... ...
...	

Figure 23: **ADI Rule**'s Template of Information Stored in a **Goal Tree Node**

To complete the discussion of the change to **Meta Income Tax Form** (⑧), we describe the ordered pair that was inserted into the **History List**, and show how it was inserted.

The new ordered pair consists of two elements: the new value of **Deferred Interest** inferred by the **ADI Rule**, and the **Node Index** value returned from the message to **Goal Tree**. For our example then, the ordered pair is **($1045.24 995)**.

The **History List** is simply a list of such ordered pairs, for example: **(($547.23 990) ($225.00 755))**. (The example used in this paragraph is for illustrative purposes local to the discussion of the **History List**; the reader should not try to relate these **History List** values to the larger example with Doe and ET). A new ordered pair is always added (or "consed") to the head of the list. So adding the pair from our larger example to the **History List** just given would result in **(($1045.24 995) ($547.23 990) ($225.00 755))**. The change to **Deferred Interest's History List** is illustrated in Figure 24.

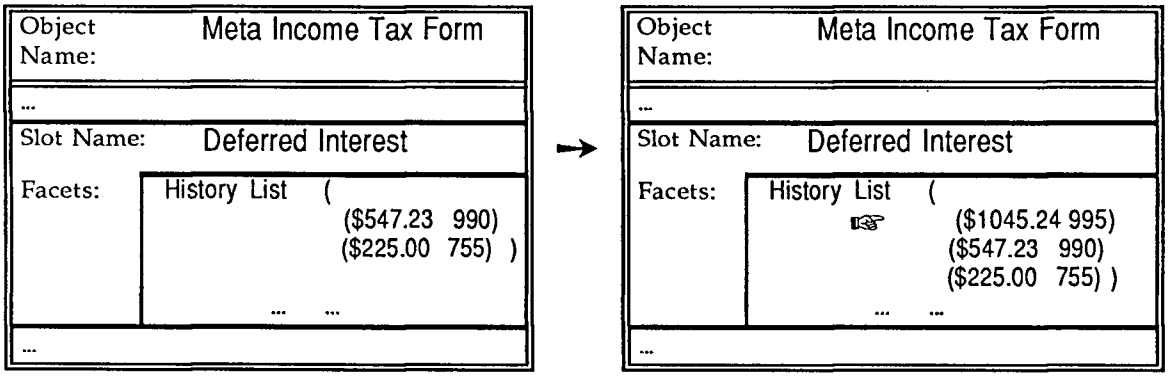

Figure 24: Inserting an Ordered Pair into the **History List** Facet for Slot **Deferred Interest**

Now let us return to the present, with Doe, anxiously awaiting her explanation. We will continue to look behind the scenes and see exactly how the explanation is generated.

Doe does not know it, but she interacts with the **Interface Manager** object during a consultation (① in Figure 25). It is the **Interface Manager's HOW? Method** that responds to Doe's request for an explanation. A HOW? explanation is always in reference to some slot and the strategy of the **Interface Manager's HOW? Method** is to explain the last slot-changing agent which set the

value of that slot. In our example, the last slot-changing agent that set **Deferred Interest**'s value was the **ADI Rule**.

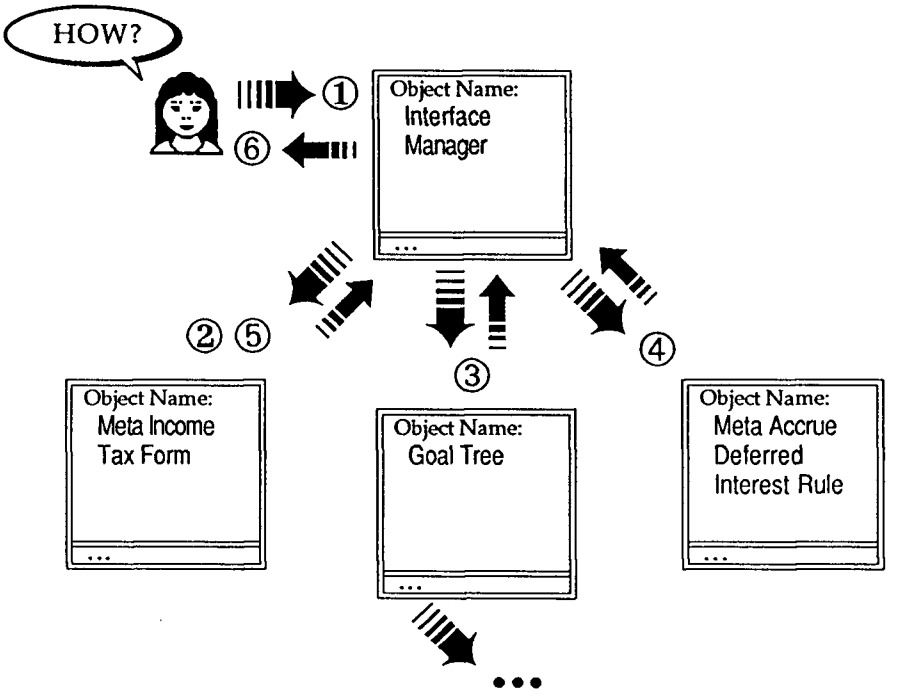

Figure 25: HOW? Explanation for a Rule

The **Interface Manager** has a template for each type of explanation it can generate, including HOW? explanations. The template for HOW? explanations for our example can be as simple as the one shown in Figure 26. The **Interface Manager** knows the slot name that the HOW? query refers to (**Deferred Interest**). Thus it need only find **Deferred Interest**'s translation, discover that the **ADI Rule** is the slot-changing agent, and fill out **Meta ADI Rule**'s explanation templates in order to fill out its HOW? template and produce its explanation.

Object Name:	Interface Manager
Inheritance:	Is Instance Of Interface Manager Class Parent
Slot Name:	HOW? Explanation Template
Facets:	Value (... ((as-translation <the slot name>) "resulted from the application of the" <the slot-changing agent's name> "which computed it" <slot-changing agent's filled-in Explanation Template for Consequent> (agent's 'Connector Template' with 'HOW? argument) <slot-changing agent's filled-in Explanation Template for Premise>) ...) ... ...
...	

Figure 26: The **Interface Manager's HOW? Explanation Template** for Slot-Changing Agents

The **Interface Manager** sends one message to find the slot's translation and its relevant **Goal Tree Node**, and then three more messages to access and fill the metaagent's explanation templates. The four messages, are sent, in the sequence shown in Figure 25, to:

② the **Meta Income Tax Form** metaobject. The **Meta Income Tax Form** method which processes the message locates and returns to the **Interface Manager** two pieces of information. It returns the ordered pair from the head of **Deferred Interest's History List** facet, i. e. the pair **($1045.24 995)**. It also returns a translated version of **Deferred Interest**'s name and current value, in our example "The value of **Deferred Interest**, **($1045.24)**,". So the list returned to the **Interface Manager** contains a pair and a string: **(($1045.24 995)** "The value of **Deferred Interest**, **($1045.24)**,").

③ the **Goal Tree** object. The method[5] which processes the message sends a message to the node object with **Node Index 995** and returns to the **Interface Manager** the template of information sent to the **Goal Tree** by **Meta ADI Rule**, i.e. the values of the slots **Agent, Goal Slot, Trigger Slots,** and **Input Slots.**

④ the **Meta Accrue Deferred Interest Rule** object. The method which processes the message returns to the **Interface Manager** the explanation templates for the **ADI Rule.** Examples of explanation templates are shown in Figure 27.

⑤ To the **Meta Income Tax Form** metaobject, again. The method which processes the message returns to the **Interface Manager** the values of slots **PRP Coverage, Savings Account Interest, Savings Bonds Interest,** and **Personal Retirement Plan (PRP) Interest** *at the time that the* **ADI Rule** *used those values to compute* **Deferred Interest***'s new value.* The final illustration in this section will look at how **Meta Income Tax Form** determines those time-dependent slot bindings.

The first message returns the slot's translation and relevant **Goal Tree Node,** the second message returns the rule, the third returns the rule's explanation templates, and the fourth fills out the values required for the rule's explanation templates. Combining the components appropriately, the **Interface Manager** is able to produce the response Doe sees at ⑥.

At ② we see that **Meta Income Tax Form**'s **Translation** facet for **Deferred Interest** is simple-minded: ("The value of <slot-name>, (<slot-value>)". For some slots, a more linguistic translation is preferred, e.g. "You are covered by a Personal Retirement Plan (PRP)" rather than a poor explanation like "**PRP Coverage** is **true**". The **Translation** facet should provide grammatical explanations depending on **PRP Coverage**'s value for simple cases, such as slots with Boolean values. For example, **Translation** could easily supply "you are covered by ..." for a true value of **PRP Coverage** and "you are not covered by ..." for a false value.

[5]By now it should be obvious which object's method, namely the object that the message was sent to.

Object Name:	Meta Accrue Deferred Interest Rule
...	
Slot Name:	Explanation Template for Consequent
Facets:	Value (('Deferred Interest" (" as the sum of interest from three sources:" (as-name 'Savings Account Interest') (get-slot-value 'Savings Account Interest') (as-name 'Savings Bonds Interest') (get-slot-value 'Savings Bonds Interest') "and" (as-name 'Personal Retirement Plan (PRP) Interest') (get-slot-value 'Personal Retirement Plan (PRP) Interest')))) ... ...
Slot Name:	Connector Template
Facets:	Value (('Deferred Interest" ((if (<HOW? explanation>) ("The computation was done because the following was true: ")) ...))) ... ...
Slot Name:	Explanation Template for Premise
Facets:	Value (('Deferred Interest" (as-translation 'PRP Coverage'))) ... ...
...	

Figure 27: Explanation Templates for **ADI Rule** for Goal Slot **Deferred Interest**

To complete the discussion of message ④'s behavior, we look next at the explanation templates for the **ADI Rule**. Figure 27 displays simple templates (**Explanation Template for Consequent, Connector Template,** and **Explanation Template for Premise**) that suffice for our example. When the **Meta ADI Rule** returns its

explanation templates to the **Interface Manager**, the latter interprets them, and sends the second message to **Meta Income Tax Form** (⑤ in Figure 25), to get the appropriate values.

The explanation templates are lists indexed by slot name into a nested list structure. The indexing provides a simple-minded solution to a complicating factor that is not apparent from our current example: a rule (or a method, for that matter) can solve more than one goal. For example, the **Consequent** of a rule may set more than one slot's value. In such cases the rule would exhibit the behavior in Figure 20 *for each goal it solved.* And the **Object Compiler** would have built a **Goal Tree Template** that contained a template like that shown in Figure 22 for each goal as well. For our illustrations, this complication is hidden, and we view the templates only from the index of the goal we are interested in explaining.

Finally, one last detail to explore. The description above of message ⑤ in Figure 25 alluded to time-dependent slot bindings for the four values in **ADI Rule**'s explanation templates. The mechanism that permits time-dependent retrieval of values from a metaobject's **History List** is the requirement that **Goal Tree Node** index values be assigned in strictly ascending order. The index values therefore provide an explicit temporal ordering of **Goal Tree** events—and, by extension, they order all events in the knowledge base.

To find the value of a slot at the time of creation of a particular **Goal Tree Node** , **Meta Income Tax Form's** method examine the slot's **History List** facet. The method searches the facet's value (a list of ordered pairs) for the pair whose **Node Index** is the greatest of all those in the list whose **Node Indexes** are less than the reference point **Node Index.** For example, assume **Deferred Interest's History List** facet were **(($1045.24 995) ($225.00 755) ($0.00 1)).** If **Meta Income Tax Form** needed to determine the value of **Deferred Interest** at the time of insertion of an arbitrary

Goal Tree Node, say **Node Index 824**, it would use the search technique to discover the value to have been **$225.00**—since **Node Index 755** is the greatest **Node Index** in the list that is less than **824**. In our example with Doe's HOW? query, the **Meta Income Tax Form** metaobject uses the same search technique to locate the values, at the time **Goal Tree Node 995** was created, for the four slots in the **ADI Rule**'s explanation templates.

Thus we have seen in Figures 25 through 27 how the **Interface Manager** produces the response Doe sees on her screen. Obviously, this is by no means a complete description of the rule-based processing. It does not mention how rule-based inference (forward, backward, and mixed chaining) is implemented. Indeed the entire inference engine is being (and for these examples will continue to be) viewed as a "black box". It does, however, illustrate how important aspects of Mycin's goal tree mechanism can be realized in a shell designed to the specification of chapter 4. The aspects illustrated here are important because they are amenable to generalization.

This section has illustrated the objectification of Mycin's HOW? explanations. The next section will show how the Mycin-style response to a WHY? question would be objectified.

5.2 WHY? EXPLANATION FOR A RULE

Having endured so much detail to see how HOW? explanations are produced for rules, the reader may feel some relief that just half the effort is required in order to see behind the scenes of WHY? explanations.

Let's look in again on Doe, actually a few moments before the previous example. ET has just asked her for information:

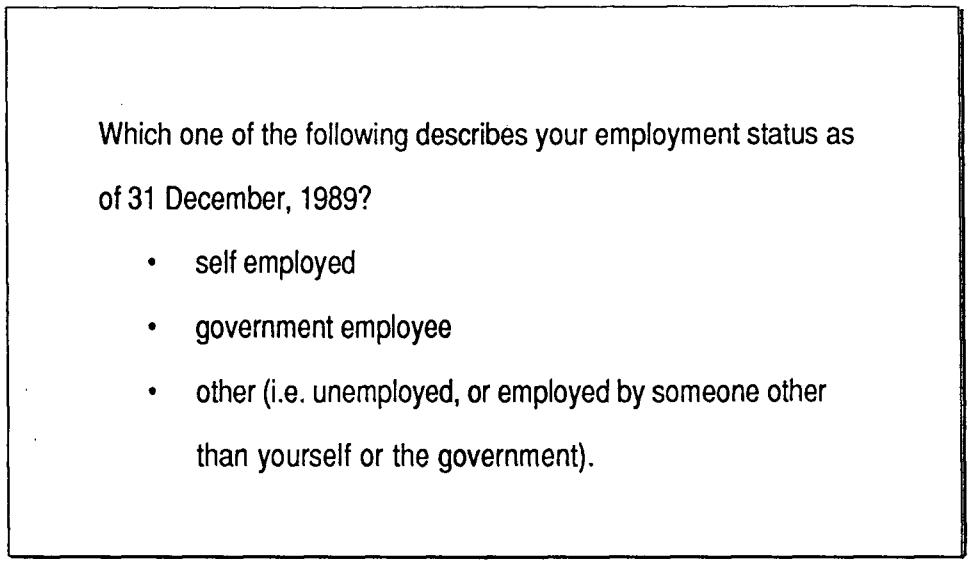

Figure 28: A User Query

Doe does not see why ET should need to know this. (Doe is forgetting that her accountant has known her much longer than ET, and has been intimately aware of her financial situation, including her employment status, for several years.) In any case, rather than giving an answer, Doe indicates that she would like to know WHY? she is being asked. ET responds:

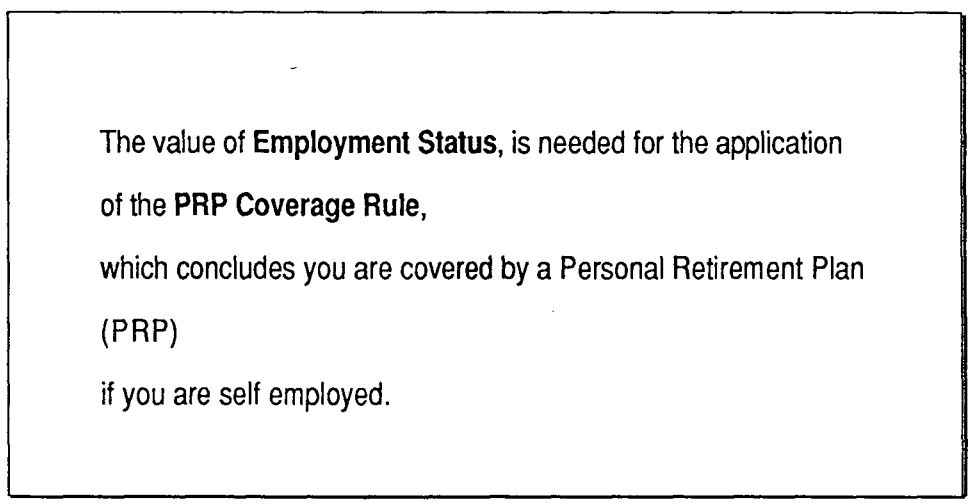

Figure 29: Sample WHY? Explanation for a Rule

Seeing how ET produces this WHY? explanation is easy if we ignore details unrelated to explanation. At the moment the WHY? explanation refers to, the rule-based portion of the **Inference Engine** is in the driver's seat: it is attempting to fire the **PRP Coverage Rule**, shown in Figure 30. If the rule does fire, **Meta PRP Coverage Rule** will add a new node to the **Goal Tree**. To bound the discussion, we abstract away the **Inference Engine**, and view it only through the behavior of the **Goal Tree** and the **Interface Manager**.

Object Name:	PRP Coverage Rule
...	
Slot Name:	Premise
Facets:	Value (= 'Employment Status' "self employed") ... ...
Slot Name:	Consequent
Facets:	Value (set-slot 'PRP Coverage' ; TO THE VALUE true) ... ...
...	

Figure 30: The **PRP Coverage Rule**

We can trace in Figures 31 and 32 the exchanges of information which elicit, and then produce, the WHY? explanation. The two figures could have been combined, but they are drawn separately to emphasize a subjective difference of what object is more "in control" at the moment. During normal processing the overall inference engine, embodied in Figure 31 as the **Goal Tree**, controls the reasoning, calling on the **Interface Manager** when necessary. When an explanation has been triggered (Figure 32), we can view the **Interface Manager** as "in control," at least in a local sense; then the **Interface Manager**

switches roles with the **Goal Tree** and calls on it to supply information to satisfy the explanation request.

In Figure 31, the **Goal Tree** (representing the inference engine) is deciding whether to fire the **PRP Coverage Rule** (①). **Goal Tree** sends a message, ②, to the **Interface Manager**, asking it to pose its query for **Employment Status**. The **Interface Manager** does so and Doe asks WHY? at ③.

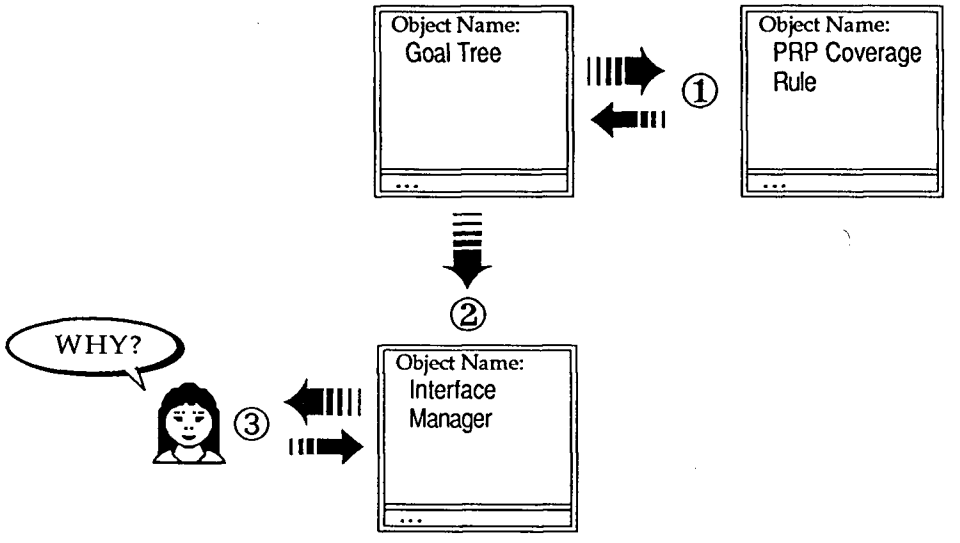

Figure 31: Context of a Specific WHY? Explanation

The **Interface Manager's WHY? Method** assumes control at this point, as illustrated in Figure 32. A WHY? explanation is always in response to a series of events like the one just described. The inference engine decides it needs to determine the value of a slot, decides that asking the user is a valid way to find it out, and sends a message to **Interface Manager** asking it to pose the query and return the answer. What stimulates the inference engine to decide that it needs information is coded in the knowledge base in a slot-changing agent. Thus the **WHY? Method's** strategy is to explain the slot-changing agent which is

generating this particular request for information from the user. In our example, that agent is the **PRP Coverage Rule**.

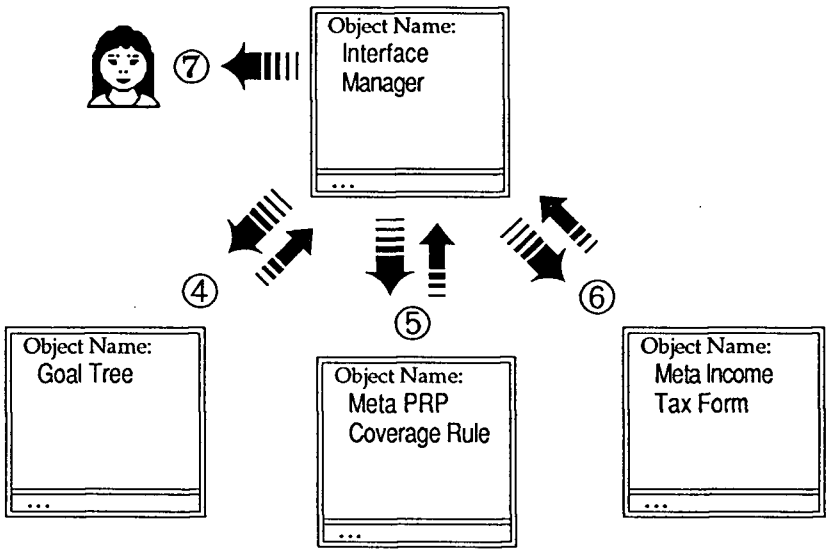

Figure 32: WHY? Explanation for a Rule

The **Interface Manager** fills in its **WHY? Explanation Template** to produce the WHY? explanation, just as for HOW? explanations. A simple, but satisfactory, template is shown in Figure 33. The **Interface Manager** must discover that the **PRP Coverage Rule** is the agent responsible for the query, and then fill out the rule's explanation templates in order to fill out its WHY? template and produce its explanation. (The **Interface Manager** already knows the translation for the slot **Employment Status**, since it just asked the user for the slot's value—a detail not shown on Figure 31.)

Object Name:	Interface Manager
...	
Slot Name:	WHY? Explanation Template
Facets:	Value (((as-translation <the slot name>) "is needed for the application of the" <the slot-changing agent's name> "which concludes" <slot-changing agent's filled-in Explanation Template for Consequent> <agent's 'Connector Template' with 'WHY? argument> <slot-changing agent's filled-in Explanation Template for Premise>))
...	

Figure 33: The **Interface Manager's WHY? Explanation Template**

The **Interface Manager** sends a message to find the agent's identity, and two more messages to fill in the agent's explanation templates. The three messages are sent, as shown in Figure 32, to:

④ the **Goal Tree** object. The method which processes the message returns to the **Interface Manager** the name of the slot-changing agent which stimulated the user query, **PRP Coverage Rule.**

⑤ the **Meta PRP Coverage Rule** metaagent object. The method which processes the message returns to the **Interface Manager** the explanation templates for the **PRP Coverage Rule.** Those templates are illustrated in Figure 34.

⑥ the **Meta Income Tax Form** metaobject. The method which processes the message returns to the **Interface Manager** the translations of the the slots required to fill out **Meta PRP Coverage Rule's** explanation templates, namely **PRP Coverage** and **Employment Status.**

Combining the responses from its messages appropriately, the **Interface Manager** produces the response Doe see at ⑦ in Figure 32.

Object Name:	Meta PRP Coverage Rule
...	
Slot Name:	Explanation Template for Consequent
Facets:	Value (('PRP Coverage' (as-translation 'PRP Coverage'))) ... ...
Slot Name:	Connector Template
Facets:	Value (('PRP Coverage' ((if (<WHY? explanation>) ("if ")) ...))) ... ...
Slot Name:	Explanation Template for Premise
Facets:	Value (('PRP Coverage' (as-translation 'Employment Status'))) ... ...
...	

Figure 34: Explanation Templates for **PRP Coverage Rule** for Goal Slot **PRP Coverage**

This section and the last illustrate just the objectification of Mycin's explanation functionality, with no enhancement to that functionality. The next three sections extend that functionality to explain the entire range of knowledge representation in the hybrid expert system shell specification.

5.3 HOW? EXPLANATIONS FOR METHODS (AND DEMONS)

This section demonstrates the possibility of generalizing Mycin's techniques for rule-based explanation, and thereby extending their explanatory range. The section's examples illustrate how Mycin-style HOW? explanations can be generated for methods, both ordinary ones and demons.

Pursuing our example, let's look in on Doe and Exper-Tax (ET) again. Doe asks HOW? the line on her form for Total Tax was computed. ET's explanation is:

> The value of **Total Tax**, **$14,250.11**, resulted from the
> application of the **Compute Total Tax Method**,
> which computed it as the formula (A-B)*C, where:
> A is **Base Tax ($18,000.12)**,
> B is **Citizen Deduction ($3000.00)**, and
> C is **Legislative Factor (0.95)**.

Figure 35: Sample HOW? Explanation for a Method

Looking behind the scenes of this HOW? explanation is simplified by first considering the "big picture". The kernel idea of this thesis is the *generalization of the behavior for rules* explained in the last two sections *to work for all slot-changing agents*. That is, the same generalized mechanism can provide explanations for rules, methods, demons, user input, and external access.

If we look at the **Compute Total Tax Method** in Figure 36, we can verify that ET's explanation is accurate regarding **Total Tax**'s computation.

Object Name:	Compute Total Tax Method
Inheritance:	Is Instance Of Methods
Slot Name:	Source Code
Facets:	Value ((set-slot 'Base Tax' ; TO THE VALUE (* ('Taxable Income ' 'Tax Rate'))) (if 'Is Citizen' (set-slot 'Citizen Deduction' $3000.00) ; ELSE (set-slot 'Citizen Deduction' $0.00)) (set-slot 'Total Tax' (* (- 'Base Tax' 'Citizen Deduction') 'Legislative Factor'))) ... ...
...	

Figure 36: **Compute Total Tax Method**

It is simple to illustrate a generalization of HOW? explanation behavior: Figures 20 and 25 must be generalized by substituting generic instances of objects for specific ones, e.g. <slot-changing agent> instead of **ADI Rule**. Figures 37 and 39, duplicates of Figures 14 and 16 respectively, illustrate schemata that realize the generalization. (In these schema figures, generic instances are shaded.) Using the figures for illustration, let's now consider how the HOW? explanation for Doe's **Total Tax** was generated.

At the time the **Compute Total Tax Method** (call it **CTT Method**) fired, it generated *three* sets of messages like those in Figure 37. The **Goal Slots** of the resulting **Goal Tree Nodes** are: **Base Tax, Citizen Deduction,** and **Total Tax.** The notion of parsing a method into multiple goals was introduced in section 4.3. For the moment we are only concerned with the third set of messages, those associated with **Total Tax.**

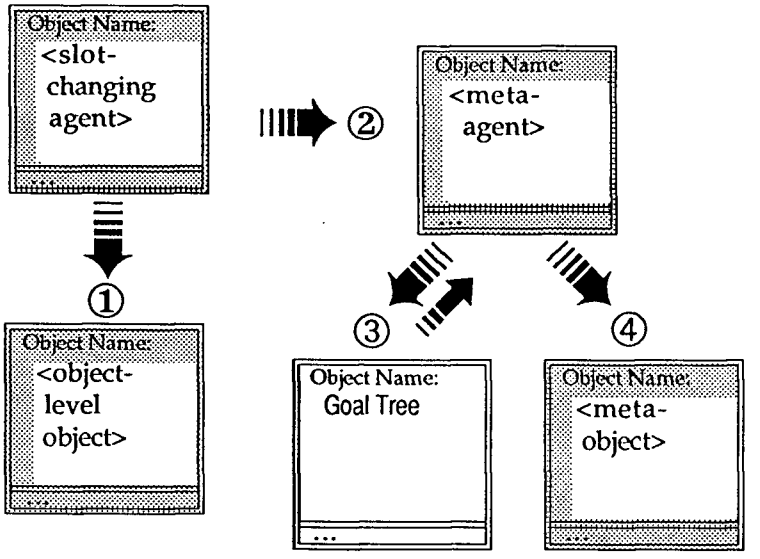

Figure 37: A Slot-Changing Agent Stores Information for HOW? Explanations

The four messages in Figure 37 occurred for **Total Tax** with the following substitutions of specific objects for generic ones: **CTT Method** for the slot-changing agent, **Meta CTT Method** for the metaagent, **Income Tax Form** for the object-level object, and **Meta Income Tax Form** for the metaobject. The behaviors associated with the calls are the same for **CTT Method** as they were for the **ADI Rule**. Figure 38 illustrates the template of information that **Meta CTT Method** passed on to **Goal Tree** for it to store in the new **Goal Tree Node**.

Object Name:	Goal Tree Node 2907
...	
Slot Name:	Agent
Facets:	Value Compute Total Tax Method ... ...
Slot Name:	Goal Slot
Facets:	Value Total Tax ... ...
Slot Name:	Trigger Slots
Facets:	Value nil ... ...
Slot Name:	Input Slots
Facets:	Value ('Base Tax' 'Citizen Deduction' 'Legislative Factor') ... ...
...	

Figure 38: **CTT Method**'s Template of Information Stored in a **Goal Tree Node**

Now let us return again to the present, with Doe asking for and getting her explanation. The general schema for HOW? explanations, generalized from Figure 25, is shown in Figure 39. Readers with an eye for detail may have noticed that Figure 39 has more objects on it than Figure 25. This is required for generality about what metaobjects the **Interface Manager** sends messages to, to get the slot values and translations that fill out its templates, ⑤ in the figure. The **Interface Manager** may or may not return for information to the same metaobject that the query is about. Likewise it may or may not access values and translations from other metaobjects. Thus ⑤ in Figure 39 represents any number (including zero) of messages to metaobjects.

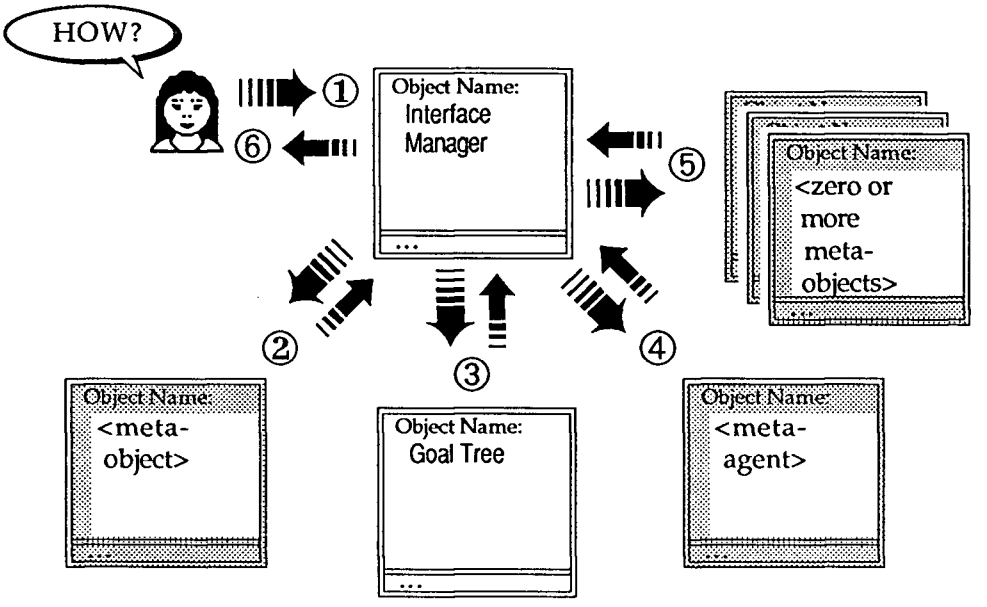

Figure 39: HOW? Explanation Schema for Slot-Changing Agents

The **Interface Manager** sends four messages to produce its explanation of **Total Tax** for Doe. The messages fit the schema of Figure 39, with the appropriate substitutions of specific objects for generic ones from the discussion of Figure 37 and the substitution of the **Meta Income Tax Form** metaobject for the collection of generic metaobjects at ⑤. Again the behaviors are the same for **CTT Method** as for **ADI Rule**. The explanation templates in **Meta CTT Method** shown in Figure 40 permit the **Interface Manager** to fill out its **HOW? Explanation Template** and produce the answer Doe sees at ⑥.

Object Name:	Meta Compute Total Tax Method
...	
Slot Name:	Explanation Template for Consequent
Facets:	Value (('Total Tax' (" as the formula (A-B)*C, where: A is " (as-name 'Base Tax') (get-slot-value 'Base Tax') ", B is" (as-name 'Citizen Deduction') (get-slot-value 'Citizen Deduction') ", and C is" (as-name 'Legislative Factor') (get-slot-value 'Legislative Factor')))) ... ...
Slot Name:	Connector Template
Facets:	Value (('Total Tax' nil) ...) ... ...
Slot Name:	Explanation Template for Premise
Facets:	Value (('Total Tax' nil) ...) ... ...
...	

Figure 40: Explanation Templates for **CTT Method** for Goal Slot **Total Tax**

The first part of this section has demonstrated how the generalized behavior schemata, Figures 37 and 39, can be applied to produce explanations for methods. The remainder of the section demonstrates their application to demons, a subclass of methods. Ordinary methods that we have talked about up to this point must be explicitly invoked, by a rule or method. As described in section 2.4, demons are augmented methods: they have a slot named **Invocation Condition**, and the inference engine watches globally and fires a demon whenever its **Invocation Condition** is satisfied.

Object Name:	Low Taxable Income Demon
Inheritance:	Is Instance Of Demons
Slot Name:	Source Code
Facets:	Value ((if (≤ 'Taxable Income' $10000.00) (set-slot 'Total Tax' $0.00))) ... ...
Slot Name:	Invocation Condition
Facets:	Value (after-set 'Taxable Income') ... ...
...	

Figure 41: **Low Taxable Income Demon**

Figure 41 illustrates a demon in Ex-Tax, named **Low Taxable Income Demon (LTI Demon)**. Unbeknownst to Doe, this demon fires during her consultation with Ex-Tax, whenever some slot-changing agent sets a value for slot **Taxable Income**. Were Doe's **Taxable Income** less than $10,000.00, the demon would immediately set her **Total Tax** to zero. This does not happen, since Doe's **Taxable Income** is considerably more than $10,000.00.

Object Name:	Goal Tree Node 102	
...		
Slot Name:	Agent	
Facets:	Value	Low Taxable Income Demon
	...	...
Slot Name:	Goal Slot	
Facets:	Value	Total Tax
	...	...
Slot Name:	Trigger Slots	
Facets:	Value	('Taxable Income')
	...	...
Slot Name:	Input Slots	
Facets:	Value	nil
	...	...
...		

Figure 42: **LTI Demon**'s Template of Information Stored in a **Goal Tree Node**

When Doe later uses ET for her son John's taxes, the **LTI Demon** does fire and this event provides the explanation for our example. Three figures illustrate the key information needed to produce a HOW? explanation for a demon. Figure 42 illustrates the **LTI Demon**'s template of information, stored in its corresponding **Goal Tree Node**. Figure 43 shows the demon's explanation templates. Finally, Figure 44 is the HOW? explanation that Doe sees.

Object Name:	Meta Low Taxable Income Demon
...	
Slot Name:	Explanation Template for Consequent
Facets:	Value (('Total Tax' (" as zero "))) ... ...
Slot Name:	Connector Template
Facets:	Value (('PRP Coverage' ((if (<WHY? explanation>) ("if ")) ...))) ... ...
Slot Name:	Explanation Template for Premise
Facets:	Value (('Total Tax' ("(1)" (as-translation '(after-set 'Taxable Income')) "(2)" (as-translation ' (≤ 'Taxable Income' $10000.00)))))) ... ...
...	

Figure 43: Explanation Templates for **LTI Demon**

Demons are explained exactly the same way as ordinary methods. Reflecting their distinct invocation mechanism, several slots in a demon's explanation process always contain values appropriate for the demon's **Invocation Condition.** Examples are the **Trigger Slots** in the **Goal Tree Node** for the demon (Figure 42) and the metaagent's **Connector Template** and **Explanation Template for Premise** (Figure 43).

The value of **Total Tax**, **$0.00**, resulted from the application of
the **Low Taxable Income Demon**,
which computed it as zero.
The computation was done because the following were true:
(1) the value of **Taxable Income** was calculated, and
(2) the value of **Taxable Income**, **($2,755.88)**, was ≤ $10,000

Figure 44: Sample HOW? Explanation for a Demon

This section introduced the generalization of rule-based HOW? explanations, producing the schemata in Figures 37 and 39. It then showed that the generalization enables explanations of both ordinary methods and demons. In the next section we see that the WHY? explanation mechanism also generalizes.

5.4 WHY? EXPLANATIONS FOR METHODS (AND DEMONS)

This section generalizes the rule-based WHY? mechanism and presents examples, again, for methods and demons. For ordinary methods, and then for demons, we look at samples of **Source Code**, explanation templates, and a resulting explanation. Textual explanations are very brief in this and the next few sections, mainly setting context for the figures.

Figure 45 is the schema for WHY? explanations (duplicating Figure 17). It generalizes Figure 32, and also includes the context of asking WHY? (from Figure 31). In Figure 45, the **Interface Manager** uses the explanation templates

from the metaagent to fill out its **WHY? Explanation Template** (Figure 33) and produce the explanation at ⑤.

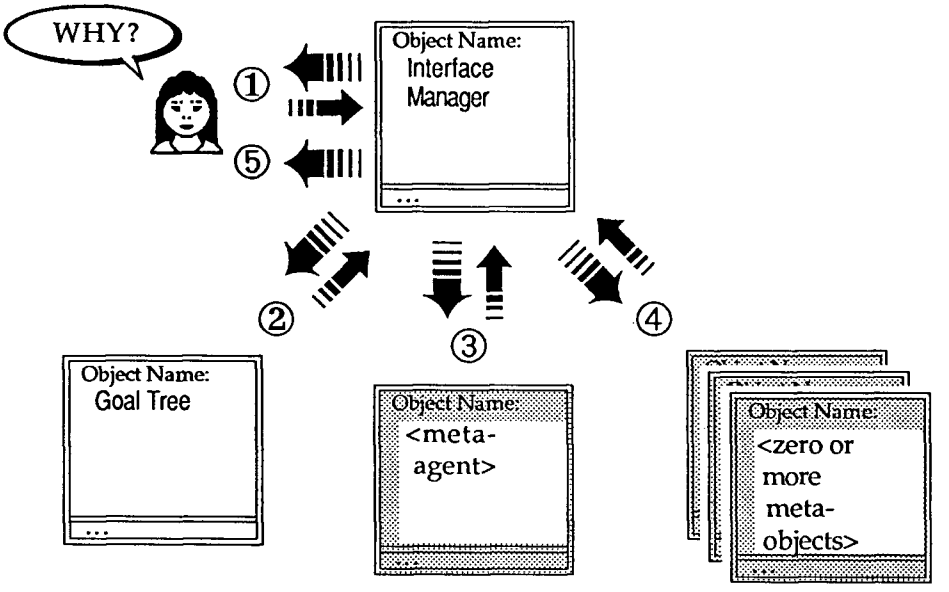

Figure 45: WHY? Explanation Schema

Object Name:	Compute Total Tax Method
...	
Slot Name:	Source Code
Facets:	Value ((set-slot 'Base Tax' ; TO THE VALUE (* ('Taxable Income ' 'Tax Rate'))) (ask-user-if-unknown 'Is Citizen')) (if 'Is Citizen' (set-slot 'Citizen Deduction' $3000.00) ; ELSE (set-slot 'Citizen Deduction' $0.00) (set-slot 'Total Tax' (* (- 'Base Tax' 'Citizen Deduction') 'Legislative Factor'))) ... ...
...	

Figure 46: **Compute Total Tax Method,** Augmented with User Query

The version of **CTT Method** shown in Figure 46 is more complete than the version shown in Figure 36, in that it seeks the value of **Is Citizen** if that value is unknown. This user query is effected by a message to the **Interface Manager**'s **Ask User If Unknown Method** (**"Ask User ... Method"** for short). The method compiler has strict rules about when the **Ask User ... Method** may be used, to give it explanatory capability, as described in section 4.3. When the **Ask User ... Method** is used in a method or demon, the **Object Compiler** creates a corresponding goal in the templates of the method's or demon's metaagent. Figure 47 illustrates the explanation templates in our example for goal slot **Citizen Deduction**. Combining its **WHY? Explanation Template** (Figure 33) with the templates in Figure 47, the **Interface Manager** produces the explanation in Figure 48.

Object Name:	Meta Compute Total Tax Method
...	
Slot Name:	Explanation Template for Consequent
Facets:	Value (('Is Citizen' (as-translation 'Citizen Deduction') ...))) ... ...
Slot Name:	Connector Template
Facets:	Value (('Is Citizen' ((if (<WHY? explanation>) ("based on")) ...))) ... ...
Slot Name:	Explanation Template for Premise
Facets:	Value (('Is Citizen' (as-translation 'Is Citizen')) ...))) ... ...
...	

Figure 47: Explanation Templates for **CTT Method** for Goal Slot **Is Citizen**

Your citizenship is needed for the application of the **Compute Total Tax Method,**
which concludes the value of your **Citizen Deduction**
based on your citizenship.

Figure 48: Sample WHY? Explanation for a Method

Object Name:	Low Gross Income Demon
Inheritance:	Is Instance Of Demons
Slot Name:	Source Code
Facets:	Value ((if (≤ 'Gross Income' $20000.00) ; THEN (set-slot 'Total Tax' $0.00)) ; ELSE (if (≤ 'Gross Income' $25000.00) ; THEN ((ask-user-if-unknown 'Is Citizen') (if 'Is Citizen' ; THEN (set-slot 'Total Tax' $0.00)))))) ... ...
Slot Name:	Invocation Condition
Facets:	Value (after-set 'Gross Income') ... ...
...	

Figure 49: **Low Gross Income Demon**

The **Low Gross Income Demon (LGI Demon,** shown in Figure 49) provides the example for a demon. It is similar to the **Low Taxable Income Demon** seen earlier. It applies when **Gross Income** has been computed and may set a taxpayer's **Total Tax** to zero, if the taxpayer's **Gross Income** is low enough. "Low enough" depends on whether the taxpayer is a citizen or not, and **LGI Demon** uses **Ask User ... Method** to get the value of **Is Citizen.** Figure 50 shows the explanation templates for **LGI Demon** for goal slot **Is Citizen,** and Figure 51 is the resulting explanation.

Object Name:	Meta Low Gross Income Demon
...	
Slot Name:	Explanation Template for Consequent
Facets:	Value (('Is Citizen' (as-translation 'Total Tax') ...))) ... ...
Slot Name:	Connector Template
Facets:	Value (('Is Citizen' ((if (<WHY? explanation>) ("based on")) ...))) ... ...
Slot Name:	Explanation Template for Premise
Facets:	Value (('Is Citizen' (as-translation 'Is Citizen')) ...))) ... ...
...	

Figure 50: Explanation Templates for **Low Gross Income Demon** for Goal Slot **Is Citizen**

Your citizenship is needed for the application of the **Low Gross Income Demon**,
which concludes the value of your **Total Tax**
based on your citizenship.

Figure 51: Sample WHY? Explanation for a Demon

This section has brought the explanatory capability of methods and demons up to par with rules in a hybrid expert system shell. Given the parsing of methods and demons into rule-like goals, equivalent to rules' goals in the **Goal Tree**, the Mycin explanation mechanism has been successfully generalized to explain methods and demons too.

5.5 EXPLANATIONS FOR USER INPUT, EXTERNAL ACCESS, INHERITANCE, AND INITIALIZATION VALUES

The previous section completes the treatment of rules, methods, and demons as distinct from other slot-changing agents for explanatory purposes. This section illustrates that the remaining slot-changing agents—user query and external access—also work in the generalized explanation behavior schemata. The section concludes with a suggestion for providing HOW? explanations for two more important ways slots' values are determined: values at initialization time and inheritance.

The first new slot-changing agent we consider is user input. The discussion of WHY? explanations has already considered user queries, from the perspective when the user has been queried, but he or she asked WHY?

rather than answering. As we have seen, a WHY? question is always in response to a user query, which is itself in response to a request for a slot's value from some slot-changing agent. The answer to WHY? the user is being queried is closely related to the explanation of HOW? the slot-changing agent uses the slot being queried for.

User input is fundamentally different from the other slot-changing agents—rules, methods, demons, and external access—in that user input is *never* the slot-changing agent that is ultimately behind a user query[6]. Thus there is no need to discuss WHY? explanations for user input. All the WHY? explanations that occur get answered by reference to a rule, method, demon, or external access.

A HOW? query to many Mycin-based shells elicits a simple "You told me" response for user input. The context is past tense. That is, HOW? explanations of user input rely on the perspective of user input after the user has supplied an answer.

User input fits the behavior schemata for storing HOW? explanation information (Figure 37) and producing HOW? explanations (Figure 39). The **Ask User Method**, provided by the expert system shell, is substituted for the generic slot-changing agent in both schemata.

The **Ask User Method** facilitates uniform explanation of all user queries, regardless of the slot-changing agent that stimulates the query. This facility is achieved by the **Object Compiler**'s requirement that all slot-changing agents interface to the user via **Ask User Method**. For example the rule inference engine sends messages to **Ask User Method** for any user input. Methods and demons

[6]If one tried to concoct a scenario in which one user input directly stimulated another, my reply would be that it must have been rules or methods producing the behavior, else it is outside the class of hybrid shells this thesis addresses.

implementing user input are restricted to using either **Ask User Method** itself or methods that ultimate use **Ask User Method**, such as the **Ask User If Unknown Method** introduced above. Any facility that an external access class object has for communicating with the user is coded in rules, methods, and/or demons and hence also restricted to using **Ask User Method**.

Since **Ask User Method** is a method, the explanation mechanisms already described for methods apply. Returning to our example, let's observe Doe as she asks HOW? the value of **Is Citizen** was set (she is just fooling around with Ex-Tax at this point.)

Looking back to Figure 46 reminds us of the context in which Doe was asked about her citizenship Figure 52 illustrates the distinguished **Ask User Method** being accorded distinguished, i.e. ad hoc, treatment in the **Interface Manager's HOW? Explanation Template**. Figure 53 illustrates the **Goal Tree Node** created with **Meta Ask User Method**'s template of information. For this simple answer, the information in the **Goal Tree Node** is enough for the **Interface Manager** to fill out its template and produce the explanation at Figure 54. That is, the **Meta Ask User Method** does not have explanation templates because its explanations are handled on an ad hoc basis in the **Interface Manager**.

Object Name:	Interface Manager
...	
Slot Name:	HOW? Explanation Template
Facets:	Value ((if (= <the slot-changing agent> 'Ask User Method') ;THEN ((as-translation <the slot name>) "was supplied by you" ; ELSE ...)) ... ...
...	

Figure 52: **Interface Manager's HOW? Explanation Template** for User Input

Object Name:	Goal Tree Node 1007
...	
Slot Name:	Agent
Facets:	Value Ask User Method ... ...
Slot Name:	Goal Slot
Facets:	Value Is Citizen ... ...
Slot Name:	Trigger Slots
Facets:	Value nil ... ...
Slot Name:	Input Slots
Facets:	Value nil ... ...
...	

Figure 53: **Ask User Method's** Template of Information Stored in a **Goal Tree Node**

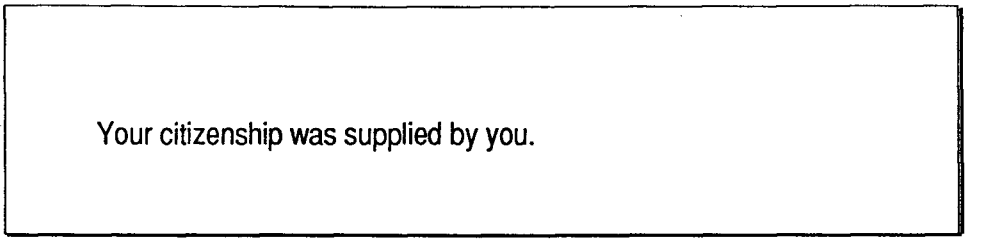

Figure 54: Sample HOW? Explanation for User Input

Now let us turn our attention from user input to objects which set slot values based on access to external objects. Our example considers a database access via modem over a telecommunications network. In Doe's mythical nation, taxpayers' statements of wages and withholdings are on-line. Thus early on in the consultation Ex-Tax dials out to the national Tax Database. Doe can ask both HOW? and WHY? questions about the activities of the slot-changing agent objectified as the **Tax DB External Access** object.

HOW? explanations are handled for external access just as shown in schemata Figures 37 and 39. Of course the **Tax DB External Access** object is substituted for the generic slot-changing agent in the schemata. Figures 55 and 56 show the **Goal Tree Node** information and **Tax DB External Access** object's explanation templates. The **Interface Manager** combines them to produce the explanation shown at Figure 57.

Object Name:	Goal Tree Node 16
...	
Slot Name:	Agent
Facets:	Value Tax DB External Access ... ...
Slot Name:	Goal Slot
Facets:	Value Gross Income ... ...
Slot Name:	Trigger Slots
Facets:	Value nil ... ...
Slot Name:	Input Slots
Facets:	Value nil ... ...
...	

Figure 55: **Tax DB External Access** Object's Template of Information Stored in a **Goal Tree Node**

Object Name:	Meta Tax DB External Access
...	
Slot Name:	Explanation Template for Consequent
Facets:	Value (('Gross Income' ("by directly accessing the value in the national Tax Database"))) ... ...
Slot Name:	Connector Template
Facets:	Value nil ... ..
Slot Name:	Explanation Template for Premise
Facets:	Value nil ... ...
...	

Figure 56: Explanation Templates for **Tax DB External Access** for Goal Slot **Gross Income**

The value of **Gross Income, $72104.75**, resulted from the application of the **Tax DB External Access**, which computed it by directly accessing the value in the national **Tax Database.**

Figure 57: Sample HOW? Explanation for External Access

WHY? Explanations for external access operate, as expected, within the schema Figure 45. Two examples of external access querying the user are (1) external access to a database querying the user for password information and (2) an external knowledge base querying the user to accomplish the task it has been asked to do. These two examples run the gamut from trivial (to explain) to potentially very complex. Our example illustrates an explanation closer to the complex end of the continuum.

Doe has been using another expert system software product for years, Ex-Invest. Ex-Tax and Ex-Invest are marketed by the same company, and they are able to communicate, to make sure that the income tax decisions Ex-Tax makes are consistent with the user's investment plans, as she has defined them to Ex-Invest. It has been almost a year since Doe has consulted with Ex-Invest (EI). Because its information is stale, EI asks Doe questions she is not used to seeing, to be certain its picture of her financial and investment situation is still correct.

We look in on Doe, who is working with Ex-Tax, remember, when ET asks her to review the investment goals she has defined to EI. Doe is

surprised and asks WHY? Figure 58 illustrates the explanation templates used by the **Interface Manager** to create the explanation Doe sees, Figure 59.

Comparing the figures, we notice the **Ex-Invest KB External Access** object displaying a different public name (an external demon name and its external knowledge base name in Figure 59) from its private one (Figure 58). Similarly, **Meta Ex-Invest KB External Access** has dynamically instantiated the **Investment Goals** portion of its explanation templates in Figure 58, since **Investment Goals** was sought during the consultation.

Object Name:	Meta Ex-Invest KB External Access
...	
Slot Name:	Explanation Template for Consequent
Facets:	Value (('Investment Goals' ("it needs a review of your Investment Goals")) ...) ... ...
Slot Name:	Connector Template
Facets:	Value (('Investment Goals' ((if (<WHY? explanation>) ("whenever the following are true: ")) ...))) ... ...
Slot Name:	Explanation Template for Premise
Facets:	Value (('Investment Goals' ("(1) you are now initiating a consultation with Ex-Invest, (Ex-Tax has done this for you) and (2) it has been more than 6 months since you last consulted with Ex-Invest")) ...) ... ...
...	

Figure 58: Explanation Templates for **Ex-Invest KB External Access** for Goal Slot **Investment Goals**

A review of your **Investment Goals** is needed for the application of the **Update Stale Information Demon**, in external KB **Ex-Invest**
which concludes it needs a review of your **Investment Goals** whenever the following are true:

(1) you are now initiating a consultation with **Ex-Invest**, (**Ex-Tax** has done this for you)

and

(2) it has been more than 6 months since you last consulted with **Ex-Invest**.

Figure 59: Sample WHY? Explanation for External Access

Slots attain values in two important ways we have not yet considered in this chapter. First, some slots, representing static information, have predefined values at initialization time that never change. Second, slots' values may be inherited if not locally specified. Although initialization values and inheritance are not considered slot-changing agents, as defined in this thesis, both mechanisms commonly occur. HOW? questions may occur in these contexts, but WHY? questions do not, since neither mechanism stimulates user queries[7].

[7]It is conceivable that a slot in the inheritance hierarchy could be content to remain unknown until its value were needed to be inherited by some descendant, at which time the user would be queried. As before, if such behavior were not implemented with rules or methods, it is outside the class of shells addressed here.

The remainder of this section illustrates a way to provide HOW? explanations for slots whose values obtain due to initialization time values or inheritance. The illustrations rely on the **Interface Manager**, both to detect the situation and to produce an appropriate explanation.

The **Interface Manager** checks for an initialization value by examining the **History List** facet returned by the metaobject for the slot (see Figure 60). The **Goal Tree Node** with **Node Index 1** is the root node of the **Goal Tree**, so if the slot's **History List** facet is a list containing just the ordered pair (<slot value> **1**), the slot's value has not changed since initialization. When the **Interface Manager** senses the condition, it does not send the other messages shown in the HOW? explanation schema for slot-changing agents (Figure 39); the simpler schema in Figure 60 suffices. The **Interface Manager** simply uses a specialized template, as in Figure 61, to produce an answer like the one in Figure 62.

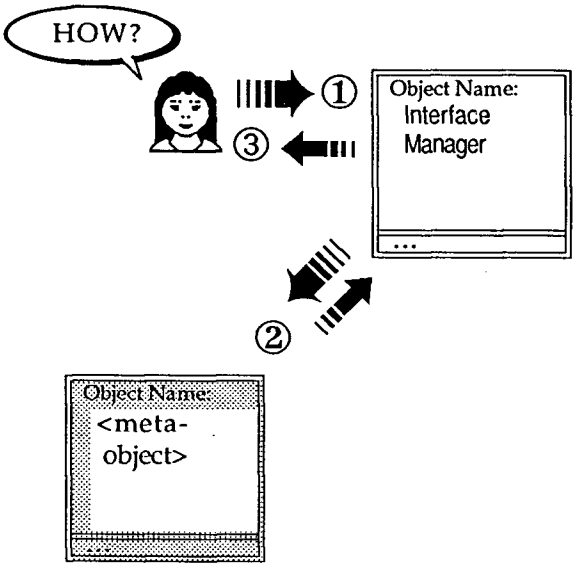

Figure 60: HOW? Explanation Schema for Initialization Time Value

Object Name:	Interface Manager
...	
Slot Name:	HOW? Explanation Template
Facets:	Value ((if (= <the slot's History List> ((<slot value> 1))) ;THEN ((as-translation <the slot name>) "was predefined at the beginning of the consultation and did not change") ; ELSE ...)) ... ...
...	

Figure 61: **Interface Manager's HOW? Explanation Template** for Initialization Values

The value of **Current Tax Year, 1989**, was predefined at the beginning of the consultation and did not change.

Figure 62: Sample HOW? Explanation for Initialization Time Value

Detecting inherited values requires the **Interface Manager** to send three messages, as illustrated in the schema in Figure 63. The **Interface Manager** begins as usual for HOW? questions, by sending a message to the metaobject for the slot (②). If the slot has no locally specified value, the metaobject returns **nil** for the head of the slot's **History List**. The **Interface Manager** must then look at the object-level object to see if the slot is inheriting a value (③).. The frame system makes inherited values obvious via the object-level slot's **Value Inherited From** facet. When a slot's value is inherited, the facet specifies the superclass which it is inherited from, as illustrated in Figure 64. The **Interface Manager**

must then access the superclass' metaobject to retrieve its **Translation** template (④) and produce its HOW? explanation (⑤).

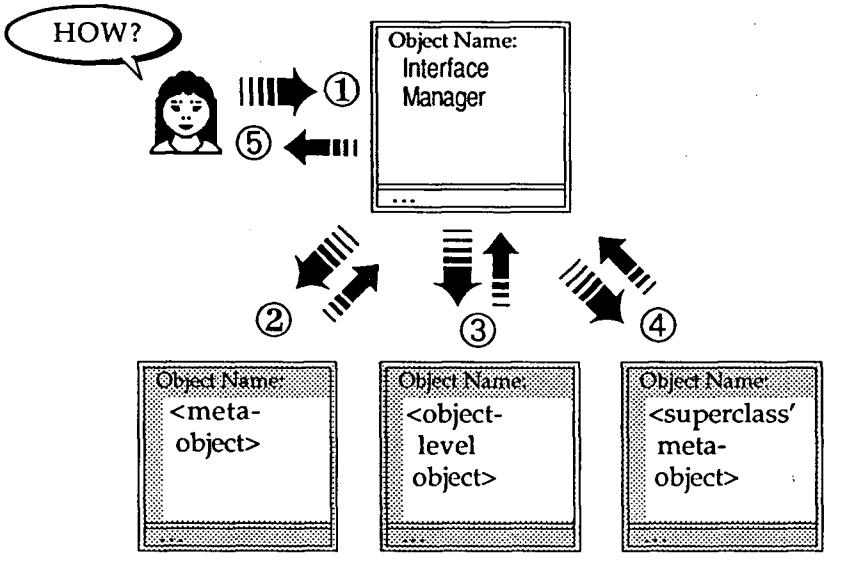

Figure 63: HOW? Explanation Schema for Inherited Value

Object Name:	Income Tax Form	
Inheritance:	Is Instance Of	Complex Tax Forms
Slot Name:	Tax Form Number	
Facets:	Value	Q4010
	Value Inherited From	Complex Tax Forms
	...	...
...		

Figure 64: Inherited Slot Value

Explaining inherited values can be quite simplistic. For our example, the **Interface Manager** applies the template in Figure 65 to produce the explanation in Figure 66. Explanations based on more sophisticated templates could

clarify the nature of inheritance, e.g. whether it is direct (one IS-A link) or chained, or whether the link is a class-instance or class-subclass link.

Object Name:	Interface Manager
...	
Slot Name:	HOW? Explanation Template
Facets:	Value ((if (<the slot's value is inherited>) ;THEN ((as-translation <the slot name>) "because" (as-translation <the object name>) "is a" (as-translation <the superclass name>)) ; ELSE ...)) ... ...
...	

Figure 65: **Interface Manager's HOW? Explanation Template** for Inherited Values

Your **Tax Form Number** is **Q4010**

because your **Income Tax Form** is a complex tax form, due to employment status and/or investment activities.

Figure 66: Sample HOW? Explanation for Inherited Value

5.5 EXPLANATIONS FOR USER INPUT, EXTERNAL ACCESS, INHERITANCE, AND INITIALIZATION VALUES

The preceding sections have presented examples that illustrate the manner in which HOW? and WHY? questions may be answered for all ways that a slot's value can be determined. Queries about slots whose values are set by slot-changing agents are uniformly treated by the **Interface Manger**. The **Interface Manger** combines its general template for the question (HOW? or WHY?) with the template specific to the agent to produce the user's explanation. For user input, the **Interface Manger** answers HOW? queries with no recourse to an agent-specific template.

We have also seen examples of HOW? explanations for slots whose values are determined by means other than agents, i.e. determined by inheritance, or unchanged since initialization time. Queries about such slots are answered by the **Interface Manger** using specific templates for each circumstance.

This concludes the examples of HOW? and WHY? queries. The Mycin style of explanation has been objectified and generalized so as to apply to all the ways a slot's value can be determined in a hybrid expert system designed to the specification in chapter 4.

5.6 WHAT-IS-IT? EXPLANATION

The mechanisms in the preceding examples have illustrated, first, the objectification of Mycin's rule explanation capability and, second, the extension of that functionality so that it applies to all slots. The final section in this chapter illustrates how easily the specification accommodates an additional kind of explanation, which we call WHAT-IS-IT? queries.

Rubinoff suggests a user may be at times confused, but not about WHY? something is being asked nor about HOW? something was determined; the

confusion may instead be about *what* something *is* ([Rubinoff85]). The WHAT-IS-IT? explanations presented here are a simplified version of Rubinoff's explanations of "concepts". Whereas Rubinoff is primarily concerned with concepts in rules, his ideas are generalized here so as to apply to slots whose values are set by any slot-changing agent.

Let's return to Doe and Ex-Tax again for one last example. To make the scenario plausible, assume that Doe became a naturalized citizen during the current tax year, and that we observe her using ET on her daughter Marie's taxes. When ET determines that Marie's **Gross Income** is $21,005.05, the program asks Doe about Marie's citizenship (because of the **LGI Demon** in Figure 49). Doe is not sure how fine-grained ET's information about citizenship is, so she asks WHAT-IS-IT? The explanation she gets is:

Is Citizen (your citizenship)

can have values true or false.

It is used in determining the following:

(1) **Total Tax**

(2) **Citizen Deduction**

Figure 67: Sample WHAT-IS-IT? Explanation

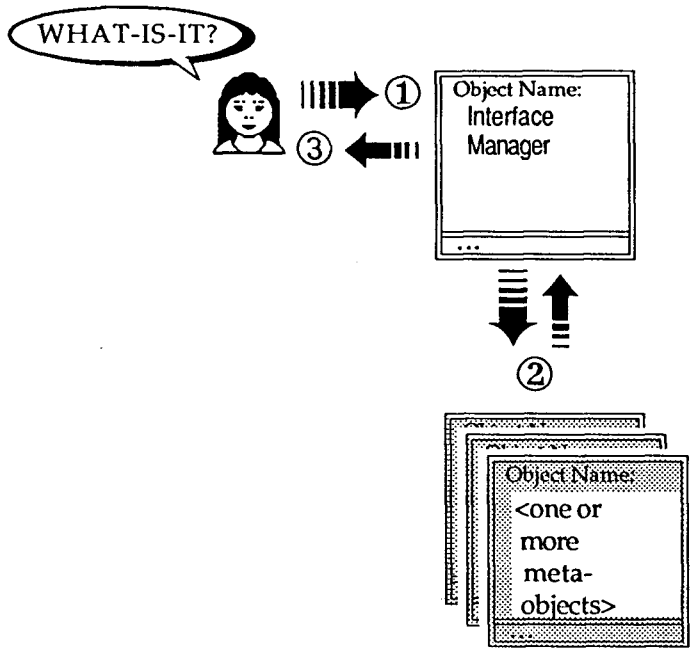

Figure 68: WHAT-IS-IT? Explanation Schema

The schema for producing WHAT-IS-IT? explanations, Figure 68 (a duplication of Figure 18), is simple. To fill out its **WHAT-IS-IT? Explanation Template**, Figure 69, the **Interface Manager** needs information from the metaobject of the slot being asked about and from the metaobjects of that slots' **Impacted Slots**. For our example, all the required information is found in **Meta Income Tax Form**. Its WHAT-IS-IT? information for slot **Is Citizen** is illustrated in Figure 70.

Object Name:	Interface Manager
...	
Slot Name:	WHAT-IS-IT? Explanation Template
Facets:	Value ((<the slot-name> (as-translation <the slot name>) "can have values" <the slot's Legal Values facet> "It is used in determining the following:" (numbered-list (mapcar (as-translation <the slot's Impacted Slots facet>))))) ... ...
...	

Figure 69: The **Interface Manager's WHAT-IS-IT? Explanation Template**

Object Name:	Meta Income Tax Form
Inheritance:	Is Instance Of Metaobjects
Slot Name:	Is Citizen
Facets:	Legal Values (true false) Translation "your citizenship" Impacted Slots ('Total Tax' 'Citizen Deduction') ... ...
...	

Figure 70: Metaobject's Information for WHAT-IS-IT? Explanations

The information in a metaobject for WHAT-IS-IT? explanations comes from two sources. The **Legal Values** facet is defined by the developer; it is required for the successful compilation of an object. The developer has the option to define the **Translation** facet, but metaobjects inherit a default translation if the user has declined, e.g. "the value of <slot-name>". The **Impacted**

Slots facet, in contrast to the first two, is inferred by the **Object Compiler**, incrementally, as slot-changing agents are compiled.

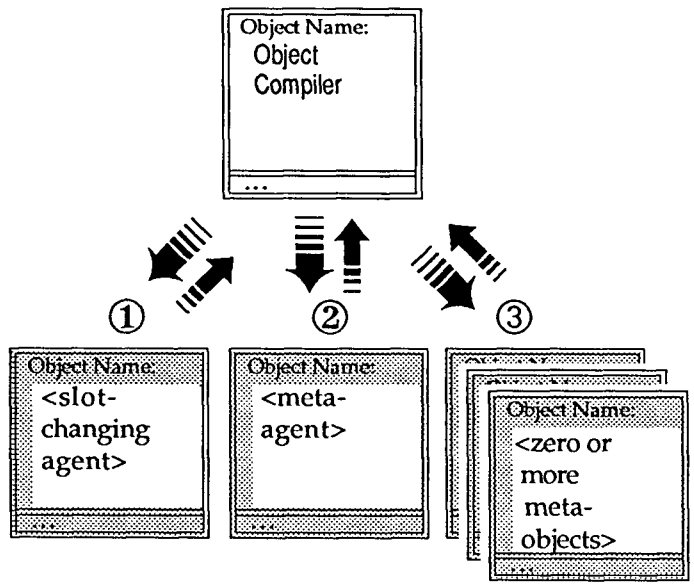

Figure 71: **Object Compiler** Stores Information for WHAT-IS-IT? Explanations

As shown in Figure 71 (which duplicates Figure 8), at the time of compilation of each slot-changing agent (①) the **Object Compiler** builds the **Goal Tree Templates** and explanation templates in the corresponding metaagent (②). In addition, for every object mentioned in the template's **Trigger Slots** and/or **Input Slots** slots, the **Object Compiler** sends a message to the corresponding metaobject (③). The method that responds to the message adds the template's **Goal Slot** to the **Impacted Slots** facet for each slot in the **Trigger Slots** and/or **Input Slots**. In our example the **Impacted Slots** facet for slot **Is Citizen** resulted from the compilation of the **CTT Method** (Figure 36) and the **LGI Demon** (Figure 49).

This chapter has provided numerous examples of explanations produced by an expert system built to specification in chapter 4. In the next and final

chapter, we consider research into better explanations, and speculate whether this framework would facilitate their creation.

6 ANALYSIS AND CONCLUSIONS

The structure and behavior proposed in chapter 4 seem capable of generating basic explanations—HOW?, WHY?, and WHAT-IS-IT?—for all aspects of hybrid expert systems' behavior, as evidenced by the examples in chapter 5. However the proposal is but a preliminary step toward providing hybrid expert system shells with truly robust explanation facilities. This final chapter considers the proposal within the larger context of what would constitute better explanation facilities for hybrid shells.

The first section in the chapter remains focused on the proposal itself. It reflects on the proposed solution, admits to some weaknesses, and makes tentative suggestions for improving them.

The following section examines the potential for adding other explanation types besides HOW?, WHY?, and WHAT-IS-IT? to the framework's repertoire. Some types of explanations seem quite amenable to the proposed framework—those that rely on metaknowledge about an expert system's behavior that can be inferred from object-level structure and behavior. The section also considers advanced proposals for building expert systems with "deeper" kinds of knowledge than most expert systems possess today.

The explanations produced by this proposal are extremely local in their scope and context. The third section examines what is required to produce better explanations, from knowledge representation and psycholinguistic points of view. Again, some enhancements seem feasible within the proposed framework, but advanced proposals require significant additions to the representations included in typical expert system shells.

The last section contains a final summary of the ideas in the proposal and recaps their origins. In light of the suggestions for enhancements to the explanation types and linguistic abilities of future expert systems, the usefulness of the thesis proposal is considered in the concluding remarks.

6.1 REFLECTING ON THE SOLUTION

This section considers three aspects of the proposed architecture for a hybrid shell's explanation facility. The first is whether the metastructures might be at a different level of abstraction than metaobjects, say metaslots or metafacets. The second topic is how one should relate metaknowledge for other purposes, say for control, to the metaknowledge for explanations presented here. The third topic raises the issue of performance concerns in an expert system supporting the extra processing to log all slot-changing activity in the **Goal Tree**.

Metaslots or Metafacets Rather Than Metaobjects?

This section cannot justify a metaobject-based architecture as the best possible architecture for explaining objects. But it does present researchers' comments that can be interpreted as arguments for strictly partitioning metaobjects from object-level objects.

In a discussion of general knowledge representation, Hayes notes the important trend toward explication of knowledge sources by physical separation, e.g. the separation of knowledge bases from the inference engine which acts on them (now the standard expert system architecture). Hayes continues that metadeductive information needs to be made explicit and

separated from factual information, "for reasons of semantic clarity, plasticity, and deductive power" ([Hayes85a]).

Sterling, in a more recent discussion focused on expert systems, argues explicitly for an architecture that consists of metalevel knowledge built around a core of expert knowledge, i.e. around the "first generation" expert system. He notes that "expressing a theory of explanation by decomposing it into metalevel programs can aid in understanding the application of the theory" ([Sterling88]).

For Maes, a crucial property for object-oriented, reflective architectures is a disciplined split between the object and reflective levels. She argues that such a split permits the creation of a standard message protocol between objects and metaobjects. A second motivation is that such a split permits changing metabehavior (e.g. turning a trace or explanations on and off) while leaving object-level objects unchanged ([Maes88]).

Relating Other Metaknowledge to Explanation Metaobjects

Consider the following scenario: a developer is building an expert system within a hybrid shell with explanation facilities that rely on metaobjects as described in chapter 4. A developer wanting to adding other kinds of metaknowledge, say for control of inference, would have a key decision to make in terms of architecture for the new metaknowledge. That metaknowledge for control could be added into existing explanation metaobjects; the result would be to endow metaobjects with two distinct functions. The alternative would be to create a separate body of metaobjects for control, distinct from the existing metaobjects; the result would be a proliferation of metaobjects, distinguished by distinctly different purposes.

Given the comments in the previous section, the best approach seems clearly to be the latter alternative. Keeping metaobjects for different purposes separate has several advantages. It would be easier to understand the individual behavior of the explanation and control metasystems if the two were not mixed. It would be easier to have the control metasystem apply to explanations, or alternatively to explain the control metabehavior if the two systems are cleanly distinct. And it would be possible to change, turn on and off, or even completely eliminate one of the metasystems while leaving the other alone if they are separate.

Performance Concerns

The processing that must be done to log all object-level activity in the **Goal Tree** is admittedly a cause for some concern about its effect on object-level performance. Jackson et al note that in some systems with explicit metalevels, the metaprocessing adds an additional order of magnitude to the object-level processing ([Jackson89]).

One possibility is to dynamically log only a subset of slots in the **Goal Tree**. This approach raises two questions: who selects the subset? Possible answers are the developer, the user, or perhaps the shell itself, based on some metalevel criteria. The second question is: how would the system respond if an explanation were requested for a slot outside the explainable subset?

A second possibility, perhaps only useful in combination with the previous one, is to log activity at a higher level of abstraction than each slot-changing agent. Possibilities are groups of rules and/or methods, which function together to compute a high level goal. Questions analogous to those in the previous paragraph immediately arise. Who defines the level of

abstraction and groupings of agents? And how would one get a lower level explanation when needed?

A promising technique for improving performance is partial evaluation, or "compiling in", which transforms a combination of object- and metalevel logic programs into more efficient code. It has been demonstrated that the technique can eliminate much of the metalevel overhead, when metaprocessing is used for control in logic-based systems ([Jackson89], [Sterling88], [Coscia88]). However its usefulness has not been explored in general for explanation nor within frame-based or hybrid architectures.

6.2 EXPLAINING OTHER QUERIES

HOW?, WHY?, and WHAT-IS-IT? are the only explanation types illustrated for the framework proposed in this thesis. It seems reasonable that other explanations could be added to this framework by adding appropriate structure and behavior in the fashion described in sections 4.1 through 4.5.

Chandrasekaran et al have analyzed explanations as belonging to one of three types. In their scheme, type 1 explanations explain object-level behavior, type 2 explanations justify that behavior, and type 3 explanations explain control strategy ([Chandrasekaran89]).

Obviously the three explanations illustrated in this thesis are all type 1. Wick and Slagle point out that these explanations are possible without supplementing the object-level knowledge coded in most current expert systems. In the next section, other type 1 explanations are described and the possibility of supporting them in the framework of this thesis is analyzed. The section after that considers type 2 and type 3 explanations.

Explaining Other Object-Level Knowledge

Wexelblat suggests several type 1 queries that users might ask. Some relate to providing the user with instructions, and are trivial to answer, e.g. "How do I do what you ask me to do?" or "What do I do next?" Other are subsumed by HOW? and WHY? queries, e.g. "Why do you ask me to do this task?" A third group are more open-ended, e.g. "What do you know about **X**?" ([Wexelblat89]). This last group of queries is simple to implement in the framework presented here. If **X** is an object-level object other than a slot-changing agent, the WHAT-IS-IT? explanation is an appropriate response. If **X** is a slot-changing agent, one must simply implement structure and behavior for agents similar to the **Impacted Slots** facet and compile time behavior described in chapter 4.

In [Wick89a] Wick and Slagle look at almost the identical problem as this thesis, i.e. what can be done to improve explanations with existing expert system shell technology and object-level knowledge sources. (The difference between this thesis' view and theirs is that they do not explicitly consider slots—which they call *phrases*—computed by agents other than rules.) Wick and Slagle's "Joe" explanation facility answers six questions—Who, What, Where, When, Why, and How—in three "tenses"[8].

The queries the Joe facility explains are shown in Figure 72. To the left of each query is a denotation indicating which aspect of this thesis' solution handles the same query, or could do so with minor modification. For example, their queries denoted by HOW?, WHY?, or WHAT-IS-IT? may vary

[8]One could debate with Wick and Slagle over their interpretation of some of the journalistic question words. And their "future tense" is actually the subjunctive mood, reflecting what *may* happen, not what *will* happen.

from the standard queries by time perspective. Their queries denoted VALUE? are easily answered by searching the **History List** as discussed in section 5.1. Their queries denoted by USER? can be handled simply by asking the user's name and/or maintaining in **Ask User Method**'s metaagent an index of its use to solicit slot values. Finally, their queries denoted INDEX? simply require compile time indexing of slots to agents or slots to slots, similar to the indexing of the **Impacted Slots** facet in section 4.3.

Treatment Here	Joe Query
HOW?	Who entered phrase is value?
VALUE?	What was phrase at time?
INDEX?	Where was phrase is value used?
VALUE?	When was phrase equal to value?
WHY?	Why was phrase is value obtained?
HOW?	How was phrase is value obtained?
USER?	Who is being recorded as author?
VALUE?	What is phrase?
WHY?	Where is phrase being used?
VALUE?	When is the birth of phrase?
WHY?	Why do you want to know phrase?
HOW?	How is phrase being computed?
INDEX?	Who can supply phrase?
WHAT-IS-IT?	What can phrase be?
WHAT-IS-IT?	Where can phrase be used?
WHAT-IS-IT?	When can phrase be of interest?
USER?	Why would you ask for phrase?
INDEX?	How can phrase be obtained?

Figure 72: Joe's Explanation Queries

Another type 1 query that could be implemented in the framework proposed here is "Why wasn't this slot computed with this agent?" Implementing it would necessitate broadening the definition of agent

"firing". In the broader view, firing would mean an agent was considered; it need not set a slot value. Examples of circumstances when an agent is considered but does not set a value occur during backward chaining or during the execution of methods that contain conditional assignments. If behavior that fits this broader definition of agent firing was also logged into the Goal Tree, it would be possible to ascertain where the state of the knowledge base prevented an agent from setting a slot's value. Admittedly this is not a trivial extension, as it may require hypothetical reasoning about consequences to link up the **Goal Tree** to the agent the query is about. [Sterling88] and [Walker87] provide approaches to negative queries.

All the queries discussed so far have been about slot values, that is, the values of slots' **Value** facets. In some applications it may be desirable to explain other facets as well. Similarly, in shells that permit dynamic placement of objects in the IS-A hierarchy at run time, explanations of such placement may be important. These explanations would require behavior to sense methods that set other facets and methods that place objects in the hierarchy and treat those methods in a fashion analogous to the way assignment statements are treated in chapter 4.

Finally, slots have been the topic discussed in queries so far. However it may be desirable to explain "What is this object?" This query, although of type 1, may be more difficult that it appears. Explaining an object in general terms (or explaining relationships between objects) requires generalization of the details associated with the object's (or objects') slots. Generalization is addressed in section 6.3.

Justifications and Explaining Control Knowledge

Explanations of object-level behavior, even all the ones suggested above, may not be adequate for users of some expert systems. Chandrasekaran et al contrast the different types of explanations that are possible if knowledge deeper than the object-level is available:

Q1: Why is a tax cut appropriate?

A1: Because a tax cut's preconditions are high inflation and trade deficits, and current conditions include those factors.

Q2: Why is a tax cut a good idea for shrinking trade deficits?

A2: Tax cuts encourage savings, stimulate investment, and increase production, which decreases prices, increases exports, makes domestic goods attractive, and reduces trade deficits.

Q3: Why aren't you suggesting increased tariffs as a way of decreasing trade deficits?

A3: Because that plan involves political costs. *Politically easier plans are considered before those with political costs.*

The first query, a simple WHY? query, is type 1 in Chandrasekaran et al's scheme. The second query justifies the expert system's decision, type 2. The third query mixes type 1 explanation with an explanation of strategy (the italicized portion), which is type 3 ([Chandrasekaran89]).

Mycin was only capable of type 1 explanations, due in part to its representation of knowledge. The uniformity of its rule-based representation mixes control and domain knowledge, making it difficult to explicate one separately from the other. Strategic information (type 3) was implicit in the

ordering of rule clauses, and hence not available for explanation purposes ([Clancey84], [Hasling83], [Wallis84]). Knowledge required to provide justifications (type 2) was not represented at all, because it was not necessary for object-level behavior ([Wallis84], [Swartout83b]).

Several researchers have studied models of causality in expert systems, anticipating improved explanation facilities. Wallis and Shortliffe admit that causal knowledge is "useful but not sufficient for problem solving in most medical domains". Problems associated with reasoning from causal knowledge are well documented ([Wallis84], ([Clancey84)]). For example, Brown argues that incorporating causal models into diagnostic systems will improve their robustness, but also admits that "most of the currently formulated models are ... surprisingly 'brittle'" ([Brown83]).

Hasling describes an early effort to improve Mycin's explanations, by encoding type 2 metaknowledge about strategy in rules. The system she describes, Neomycin, represents strategic knowledge as domain-independent metalevel goals and rules. During a consultation, metalevel goals and rules are initially activated, and they in turn ultimately access object-level goals and rules. When the user asks WHY?, explanations of the lowest level metagoal are initially presented; subsequent WHY? queries ascend up the goal tree. Neomycin demonstrated that type 3 explanations are possible from a system using a goal tree and templates for explanations, by encoding metalevel strategic knowledge.

Swartout et al argue for strict separation of underlying domain knowledge (type 2) and strategic knowledge (type 3) from object-level constructs (type 1) in expert systems. Their framework would revolutionize the construction of expert systems. Rather than building the object-level expert system, the developer concentrates on building underlying models and

devising problem solving strategies for an application domain. The underlying model includes causal relations, type hierarchies, and other kinds of knowledge. An automated program writer, using a goal-subgoal refinement process, combines these (and other) knowledge sources to build the object-level expert system, in the process generating a development history. During a consultation, a goal tree is created. All the knowledge sources—underlying model, development history, goal tree, etc.—are used to produce explanations, permitting answers to a very broad range of questions ([Neches85]).

As Wick and Slagle note, these advanced approaches to representing deeper knowledge place increased burden on expert system developers to supply supplementary information for explanations. While application of these techniques in actual practice is limited today, it is reasonable to expect their wider use, in some form, in future generations of expert systems.

This concludes the discussion of improving the range of queries that an expert system explains, by broadening the knowledge represented in those systems. Next we consider improving the linguistic expression of explanations.

6.3 EXPRESSING EXPLANATIONS BETTER

Wick and Slagle note that explanations can vary along four dimensions: grain size of the queries, the goals of explanations, the target audience, and the interactiveness of the explanation facility ([Wick89a]). This section discusses solutions to producing better explanations that address those dimensions.

Some of the approaches to improving the grain size of explanations, i.e. making them address the user's question at the right level of detail, rely on a concept of generalization in knowledge representation. For example, Rubinoff's scheme for explaining slots (the inspiration for WHAT-IS-IT? explanations) includes two techniques for improving the grain size of explanations: one involves ranking rules that help explain a slot with a notion of relevancy, the second involves generalizing rules that relate to the same slot to avoid redundancy [Rubinoff85]).

Consideration of grain size problems with rule-based explanations leads Georgeff to suggest a higher level abstraction for some applications: procedures ([Georgeff86]). One way to view Georgeff's procedures is as collections of related rules and/or methods. In a similar vein, Neomycin's strategic view of its object-level tasks includes an assignment of a focus to each task. Neomycin moves up its goal stack in a way similar to Mycin in order to produce explanations, but it is more selective about what it says than Mycin. Neomycin omits mentioning some tasks based on metarules that govern explanation behavior ([Clancey86]). The Neomycin approach is similar to Chandrasekaran et al's generic task approach, which also couches its goal tree in task terms [Chandrasekaran89].

In responding to WHY? questions, another Mycin derivative, Teiresias, expands the grain size of a question to supply context, then proceeds to answer the expanded question. Teiresias reasons about Mycin's rules and reasoning to determine the appropriate expansion of a question ([Wexelblat89]).

Other efforts to produce better explanations focus less on representation issues, relying instead on linguistic solutions for planning natural language utterances. McKeown provides a summary of the linguistic issues that text

generation systems must address, which include appropriate vocabulary, pronoun use, sentence complexity, and sentence syntax ([McKeown85a]). Weiner discusses specific psycholinguistic principles aimed at producing explanations which are "natural". He presents techniques for constraining syntactic form, embedding explanations, and attending to focus ([Weiner80]).

It is well accepted that expert systems should tailor explanations to their users. Modeling users' expertise can influence both the amount of information presented as well as the content and form of explanations ([Wick89b]). Efforts at modeling users vary from the extremely simplistic to the more principled.

Some systems simply categorize users as either "users" or "developers". Hasling uses this simple technique to choose appropriate vocabulary in explanations ([Hasling83]). Swartout extends the concept to determine which steps in a method should be included in an explanation, assuming that steps related to implementation, rather than the problem domain, will not be of interest to a user ([Swartout83b]). Wallis and Shortliffe describe an alternative categorization of users by their level of expertise; each step presented in a final explanation is included by virtue of comparing a measure of its complexity with the user's expertise level ([Wallis84]).

Other systems construct a more dynamic model of the user. Weiner's Blah system models the information it assumes the domain expert knows, and omits that information from explanations ([Weiner80]). Drawing on earlier work by Allen, Carberry, and others, researchers such as van Beek, McKeown et al, and Neches et al map the user's queries into a hierarchy of the user's inferred plans and goals ([van Beek87], [McKeown85b], [Neches85]).

Some systems model discourse as well as users. Swartout's Xplain system maintains a memory of context in order to avoid explaining steps which the

system knows the user already knows about ([Swartout83b]). McKeown builds a network of goals from individual utterances in a discourse in order to infer a higher level "relevant" goal for the discourse as a whole ([McKeown85b]).

Interestingly, most approaches to providing better explanations assume that the knowledge to be explained will be represented with rules. Yet many of the structures to support better explanations themselves rely on frame-oriented representations, either semantic networks or hierarchies like the hybrid shells discussed in this thesis.

6.4 SUMMARY AND CONCLUSIONS

The topic addressed in this thesis, explaining all aspects of hybrid expert systems' behavior, has not been explicitly addressed elsewhere. However the ideas herein represent a synthesis more than sheer originality.

The issue of expanding expert systems' explanation capabilities, but relying on existing knowledge structures and sources, is echoed by Wick and Slagle in [Wick89a]. The structures and behavior that are objectified in the thesis' solution are those of the Mycin system, as described in [Buchanan84a] and elsewhere. The notion of compiling and parsing slot-changing agents is at once an extension and simplification of Rubinoff's mechanism to explain concepts in [Rubinoff85]. Principles for an object-oriented architecture were adopted from Maes' crucial properties of an object-oriented reflective architecture in [Maes87]. The glue that binds these concepts together, notably the parsing of methods' assignment statements, represents the unique contribution of this thesis.

Since the ideas suggested here have not been tested with an implementation, an alternative way of evaluating their validity and/or usefulness is proffered. These closing remarks argue that the synchrony of these ideas with two trends suggest the ideas have merit. The first trend was noted in the introduction: hybrid architectures are the state of the art in commercial expert system shells.

The second trend is the direction of research toward producing better explanations of expert systems' behavior. Frame-like structures crop up frequently in linguistically motivated explanation research; see McKeown's hierarchies of points of view and goals in [McKeown85b] and Clancey's

generalization of rules in [Clancey84]. A goal tree is essential to generating strategic explanations as evidenced by the work of Clancey, Swartout, Neches et al, and others. Wick claims his Joe facility is implementable with any shell that includes: inferential links; histories and time stamps, properties of slots; and execution traces of slot values—all of which these ideas support ([Wick89a]).

In defense of the low level detail this proposal's explanations supply, research indicates that such detail is the appropriate foundation for better explanations. As Chandrasekaran notes, a system first has to generate an internal explanation, then employ user models to shape the explanation ([Chandrasekaran86]). Reporting on a debate at the 1988 AAAI Workshop on Explanation, Wick notes the most hotly debate topic to have been whether explanations should reflect the knowledge base's actual constructs, or whether object-level behavior and explanation structures should be decoupled. Wick reports that the majority of workshop participants argued against decoupling explanations from object-level structures. Though a raw trace is clearly too detailed for users, most participants felt the "proper way to deal with such complexity is to provide explanation routines which can select appropriate pieces of the trace to present to the user" ([Wick89b]).

The proposal in this thesis is a first step toward producing explanations for hybrid expert systems. The ideas fill a gap in current research, explaining methods and other agents on a par with rules. And the ideas provide a basis on which advanced explanations could be built. The proposal seems worthy of testing with an implementation, which would answer questions about its usefulness and performance.

BIBLIOGRAPHY

[Aiello88] Luigia Aiello and Giorgio Levi, "The Uses of Metaknowledge in AI Systems", in *Meta-Level Architectures and Reflection*, Pattie Maes and Daniele Nardi, eds., North-Holland, Amsterdam, 1988.

[Bartlett32] F. C. Bartlett, *Remembering: A Study in Experimental and Social Psychology*, Cambridge University Press, 1932.

[Bobrow85] Daniel G. Bobrow, "If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, pp. 1401-1408, Nov., 1985.

[Brachman83] Ronald J. Brachman, "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks", *Computer*, Vol. 16, No. 10, pp. 30-36, Oct., 1983.

[Brachman85a] R. J. Brachman, "'I Lied about the Trees' Or, Defaults and Definitions in Knowledge Representation", *AI Magazine*, Vol. 6, No. 3, pp. 80-93, Fall 1985.

[Brachman85b] Ronald J. Brachman and Hector J. Levesque, *Readings in Knowledge Representation*, Morgan Kaufmann, San Mateo, Calif., 1985.

[Brown83] John Seely Brown, "Causal Reasoning", in "Special Report on 1982 Workshop on Automated Explanation Production", William R. Swartout, ed., *SIGART Newsletter*, p. 12, July, 1983.

[Buchanan84a] Bruce G. Buchanan and Edward H. Shortliffe, "Explanation as a Topic of AI Research", in *Rule-Based Expert Systems*, Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison-Wesley, Reading, Mass., pp. 331-337, 1984.

[Buchanan84b] Bruce G. Buchanan and Edward H. Shortliffe, eds., *Rule-Based Expert Systems*, Addison-Wesley, Reading, Mass., 1984.

[Chandrasekaran84] B. Chandrasekaran, "Expert Systems: Matching Techniques to Tasks", in *Artificial Intelligence Applications*, Walter Reitman, ed., Ablex, Norwood, N.J., pp. 41-64, 1984.

[Chandrasekaran86] B. Chandrasekaran, J. Josephson, & A. Keuneke, "Functional Representations as a Basis for Generating Explanations", in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pp. 726-731, 1986.

[Chandrasekaran89] B. Chandrasekaran, Michael C. Tanner, and John R. Josephson, "Explaining Control Strategies in Problem Solving", *IEEE Expert*, Vol. 4, No. 1, pp. 9-24, 1989.

[Clancey84] William J. Clancey, "Extensions to Rules for Explanation and Tutoring", in *Rule-Based Expert Systems*, Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison-Wesley, Reading, Mass., pp. 531-568, 1984.

[Clancey86] W. J. Clancey, "From GUIDON to NEOMYCIN and HERACLES in Twenty Short Lessons", in *Current Issues in Expert Systems*, A. van Lamsweerde and Pierre Dufour, eds., Academic Press, 1988 (also in AI Magazine, Vol. VII, No. 3, pp. 40-60, Aug. 1986).

[Cointe88] Pierre Cointe, "The ObjVlisp Kernel: A Reflective Lisp Architecture to Define a Uniform Object-Oriented System", in *Meta-Level Architectures and Reflection*, Pattie Maes and Daniele Nardi, eds., North-Holland, Amsterdam, 1988.

[Coscia88] Patrizia Coscia, et al, "Object Level Reflection of Inference Rules by Partial Evaluation", in *Meta-Level Architectures and Reflection*, Pattie Maes and Daniele Nardi, eds., North-Holland, Amsterdam, 1988.

[Davis80] R. Davis, "Meta-Rules: Reasoning about Control", *Artificial Intelligence*, Vol. 15, No. 3, pp. 179-222, Dec. 1980.

[Forgy82] C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Match Problem", *Artificial Intelligence*, Vol. 19, No. 1, Sept., 1982.

[Georgeff86] Michael Georgeff and Amy L. Lansky, *A System for Reasoning in Dynamic Domains: Fault Diagnosis on the Space Shuttle*, SRI International Technical Note 375, Menlo Park, Calif., 1986.

[Gevarter87] William B. Gevarter, "The Nature and Evaluation of Commercial ES Building Tools", *Computer*, Vol. 20, No. 5, pp. 24-41, May 1987.

[Harmon89a] Paul Harmon, "Tool Review: Goldworks II", *Expert Systems Strategies*, Vol. 5, No. 5, pp. 8-14, 1989.

[Harmon89b] Paul Harmon, "Mainframe Tools", *Expert Systems Strategies*, Vol. 5, No. 6, pp. 1-6, 1989.

[Harmon89c] Paul Harmon, "U.S. Expert Systems Building Tools", *Expert Systems Strategies*, Vol. 5, No. 8, pp. 1-14, 1989.

[Hasling83] Diane Warner Hasling, "Abstract Explanations of Strategy in a Diagnostic Consultation System", in *Proceedings of AAAI-83 Second National Conference on Artificial Intelligence*, pp. 157-161, 1983.

[Hayes85a] Patrick J. Hayes, "Some Problems and Non-Problems in Representation Theory", in *Readings in Knowledge Representation*, Ronald J. Brachman and Hector J. Levesque, eds., Morgan Kaufmann, San Mateo, Calif., 1985.

[Hayes85b] Patrick J. Hayes, "The Logic of Frames", in *Readings in Knowledge Representation*, Ronald J. Brachman and Hector J. Levesque, eds., Morgan Kaufmann, San Mateo, Calif., 1985.

[IntelliCorp87] IntelliCorp, *KEE 3.1 TellAndAsk Reference Manual*, 1987.

[Jackson89] Peter Jackson, Han Reichgelt, and Frank van Harmelen, eds., *Logic-Based Knowledge Representation*, The MIT Press, Cambridge, Mass., 1989.

[Levesque85] Hector J. Levesque and Ronald J. Brachman, "A Fundamental Tradeoff in Knowledge Representation and Reasoning (Revised Version)", in *Readings in Knowledge Representation*, Ronald J. Brachman and Hector J. Levesque, eds., Morgan Kaufmann, San Mateo, Calif., 1985.

[Maes87] Pattie Maes, "Concepts and Experiments in Computational Reflection", in *Object-Oriented Programming: Systems Programming: Systems, Languages, and Applications '87 Conference Proceedings*, pp. 147-155, 1987.

[Maes88] Pattie Maes, "Issues in Computational Reflection", in *Meta-Level Architectures and Reflection*, Pattie Maes and Daniele Nardi, eds., North-Holland, Amsterdam, 1988.

[McCarthy85] John McCarthy, "Epistemological Problems of Artificial Intelligence", in *Readings in Knowledge Representation*, Ronald J. Brachman and Hector J. Levesque, eds., Morgan Kaufmann, San Mateo, Calif., 1985.

[McKeown85a] Kathleen R. McKeown, "The Need for Text Generation", in *AFIPS Conference Proceedings*, Anthony S. Wojcik, ed., AFIPS Press, Reston, Va., pp. 87-92, 1985.

[McKeown85b] Kathleen R. McKeown, Myron Wish, and Kevin Matthews, "Tailoring Explanations for the User", in *IJCAI Conference Proceedings*, Vol. II, pp. 794-798, 1985.

[Minsky85] M. Minsky, "A Framework for Representing Knowledge", in *Readings in Knowledge Representation*, Ronald J. Brachman and Hector J. Levesque, eds., Morgan Kaufmann, San Mateo, Calif., 1985 (also in *The Psychology of Computer Vision*, P. H. Winston, ed., McGraw-Hill, 1975).

[Mylopoulos84] J. Mylopoulos, "An Overview of Knowledge Representation", in *On Conceptual Modelling*, M.L. Brodie, J. Mylopoulos, J.W. Schmidt, eds., pp. 3-18, Springer-Verlag, 1984.

[Neches85] Robert Neches, William R. Swartout, and Johanna D. Moore, "Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, pp. 1337-1351, Nov., 1985.

[Newell72] A. Newell and H. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Parsaye88] Kamran Parsaye and Mark Chignell, *Expert Systems for Experts*, John Wiley & Sons, New York, 1988.

[Rubinoff85] R. Rubinoff, "Explaining Concepts in Expert Systems: The CLEAR System", in *Proceedings of the Second Conference on Artificial Intelligence Applications*, IEEE Computer Society Press, pp. 416-421, 1985.

[Scott84] A. Carlisle Scott, et al, "Methods for Generating Explanations", in *Rule-Based Expert Systems*, Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison-Wesley, Reading, Mass., pp. 338-362, 1984.

[Smith85] Brian C. Smith, Prologue to "Reflection and Semantics in a Procedural Language", in *Readings in Knowledge Representation*, Ronald J. Brachman and Hector J. Levesque, eds., Morgan Kaufmann, San Mateo, Calif., 1985.

[Sterling86] Leon Sterling and Ehud Shapiro, *The Art of Prolog*, The MIT Press, Cambridge, Mass., 1986.

[Sterling88] Leon Sterling, "A Meta-Level Architecture for Expert Systems", in *Meta-Level Architectures and Reflection*, Pattie Maes and Daniele Nardi, eds., North-Holland, Amsterdam, 1988.

[Swartout83a] William R. Swartout, et al, "Special Report on 1982 Workshop on Automated Explanation Production", *SIGART Newsletter*, pp. 7-13, July, 1983.

[Swartout83b] William R. Swartout, "XPLAIN: a System for Creating and Explaining Expert Consulting Programs", *Artificial Intelligence*, Vol. 21, No. 3, pp. 285-325, Sept., 1983.

[Swartout85] William R. Swartout, "Knowledge Needed for Expert System Explanation", in *AFIPS Conference Proceedings*, Anthony S. Wojcik, ed., AFIPS Press, Reston, Va., pp. 93-98, 1985.

[Swartout87] William R. Swartout, "Explanation", in *Encyclopedia of Artificial Intelligence*, Vol. I, S. Shapiro, ed., John Wiley & Sons, New York, pp. 298-300, 1987.

[Thayse88] André Thayse, ed., *From Standard Logic to Logic Programming*, John Wiley & Sons, New York, 1988.

[Thuraisingham89] Bhavani Thuraisingham, "From Rules to Frames and Frames to Rules", *AI Expert*, Vol. 4, No. 10, pp. 31-39, Oct., 1989.

[van Beek87] Peter van Beek, "A Model for Generating Better Explanations', in *Proceedings of the Association for Computational Linguistics 25th Annual Meeting*, pp. 215-220, July 1987.

[Walker87] Adrian Walker, "Expert Systems in Prolog", in *Knowledge Systems and Prolog*, Adrian Walker (ed.), Michael McCord, John F. Sowa, and Walter G. Wilson, Addison-Wesley, Reading, Mass., pp. 219-289, 1987.

[Wallis84] Jerold W. Wallis and Edward H. Shortliffe, "Customized Explanations Using Causal Knowledge", in *Rule-Based Expert Systems*, Bruce G. Buchanan and Edward H. Shortliffe, eds., Addison-Wesley, Reading, Mass., pp. 371-388, 1984.

[Weiner80] J. L. Weiner, "BLAH, A System Which Explains its Reasoning", *Artificial Intelligence*, Vol. 15, No. 1, pp. 19-48, Jan., 1980.

[Wexelblat89] Richard L. Wexelblat, "On Interface Requirements for Expert Systems", *AI Magazine*, Vol. 10, No. 3, pp. 66-78, 1989.

[Wick89a] Michael R. Wick and James R. Slagle, "An Explanation Facility for Today's Expert Systems", *IEEE Expert*, Vol. 4, No. 1, pp. 26-36.

[Wick89b] Michael R. Wick, "The 1988 AAAI Workshop on Explanation", *AI Magazine*, Vol. 10, No. 3, pp. 22-26, 1989.

[Wilkerson89] Brian Wilkerson, "Designing Object-Oriented Applications", to appear in *Apple Systems Journal*, Vol. 1, No. 1, 1989.

[Woods83] William A. Woods, "What's Important About Knowledge Representation?", *Computer*, Vol. 16, No. 10, pp. 22-27, Oct., 1983.