1987

Teraphim: a domain-independent framework for constructing blackboard-controlled, blackboard-based expert systems in Prolog

Bruce Lyon

Rochester Institutc of Technology
School of Computer Science and Technology

Teraphim:
A Domain-independent Framework for Constructing Blackboard-controlled, Blackboard-based Expert Systems in Prolog

by
Bruce Lyon

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

Approved by: _____

Professor Peter Anderson

_____

Professor Al Biles

_____

Professor Peter Lutz

April 29, 1987

Teraphim:
A Domain-independent Framework for Constructing Blackboard-controlled, Blackboard-based Expert Systems in Prolog

by
Bruce Lyon

I, Bruce Lyon, hereby grant permission to the Wallace Memorial Library of R.I.T. to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or for profit.

Signed: _____

Date: 5/7/87

ABSTRACT

The blackboard architecture, in which a set of independent knowledge sources communicate by means of a global data base known as a blackboard, has been suggested as a generally useful design for knowledge-based systems. Teraphim is a domain-independent framework for writing blackboard-based expert systems in Prolog. It implements concepts common to a range of previous blackboard architecture programs, such as HEARSAY-III and BB1. Teraphim includes as its basic elements a partitioned blackboard, a simple blackboard-controlled scheduler, a set of general-purpose scheduling heuristics to control the scheduler, a generic knowledge source with the ability to ask the user questions about incomplete data, modifiable methods of reasoning about uncertain data, and a simple explanation facility that traces the origins of terms on the problem blackboard. Trials of the system indicate that it can be used to implement expert systems to solve either synthesis or analysis problems. The blackboard architecture of Teraphim lends itself to experimentation with the kinds of knowledge representation and control knowledge needed to solve problems. Prolog proved to be a convenient language for writing blackboard-based systems.

KEY WORDS AND PHRASES

Blackboard architecture, Expert systems, Prolog

COMPUTING REVIEW SUBJECT CODES

I.2.5 [Artificial Intelligence]: Programming Languages and Software -

Expert system tools and techniques

I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods and Search -

Heuristic methods

I.2.4 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods -

Representations (procedural and rule-based)

I.2.3 [Artificial Intelligence]: Deduction and Theorem Proving -

Nonmonotonic reasoning and belief revision

TABLE OF CONTENTS

1. INTRODUCTION

Teraphim is a domain-independent framework for writing blackboard controlled blackboard architecture expert systems in Prolog. Teraphim provides the user with a simple scheduler, some generic control knowledge sources, control, agenda, and problem blackboards, a modifiable method of representing uncertainty, an explanation program, and a special knowledge source for asking questions about incomplete data at run time. The user of the system must provide a set of domain knowledge sources which post blackboard entries to the problem blackboard in response to other entries on the problem blackboard. The user may also impose some structure on the problem blackboard. The Teraphim scheduler then controls the application of the problem knowledge sources. The generic control knowledge sources control the actions of the scheduler by activating or deactivating scheduling heuristics that affect the way the scheduler chooses which action to perform next. These control knowledge sources communicate by means of the control blackboard, and the pending actions of the system are recorded on the agenda blackboard.

The generic control knowledge sources provided by Teraphim are adequate to control the execution of simple expert systems, such as might be implemented by a set of production rules. If more complex solution strategies are required, the user can write domain-specific control knowledge sources. Teraphim provides some mechanisms by which user-written control knowledge sources can change the behavior of the scheduler during execution of the program. Thus, a Teraphim program may change its behavior as the search for a solution to its problem passes through several stages.

Teraphim is similar to some other generic blackboard architecture expert system building tools, such as HEARSAY-III (BALZ80, ERMA81), BB1 (HAYE84, HAYE85b), and AGE (NII79, NII80). Teraphim shares with these systems a background of concepts that have developed as research on the construction of expert systems has proceeded.

Two example expert systems were written with Teraphim. The first system, Identifier, is an analysis system with knowledge sources like production rules. The generic control knowledge sources provided by Teraphim proved sufficient for this system. The Identifier program also demonstrated Teraphim's ability to ask for additional data at run time, and the use of Teraphim's explanation program.

The second example program, Murder_plot, is a planning system that solves a problem similar to those solved by the STRIPS program. For this program, a more complex control strategy requiring user-supplied control knowledge sources was used. This program demonstrated the ability of Teraphim to change its control behavior during the solution of a problem.

Teraphim was found to be versatile and fairly easy to program. However, the two example programs ran slowly, even though they were simple. Most of the execution effort was spent on monitoring the certainties of partial solutions and pending actions. If Teraphim were executed in a more efficient version of Prolog this performance could be improved somewhat. Like all blackboard architecture systems, Teraphim has potential for parallel execution. Parallel execution could greatly improve Teraphim's performance. As the system exists, it would be most useful for rapid prototyping of expert systems in poorly understood domains and for experimentation with different control strategies.

Section 2 of this paper presents a discussion of the development and features of the blackboard architecture. The blackboard architecture developed as an extension of the production systems used in the first successful expert systems. Blackboard-based systems share a set of distinctive components; domain-independent expert system building tools such as HEARSAY-III, BB1, and AGE providing these components have been written. The implementations of these three systems are compared.

Section 3 documents the design of Teraphim. Each of the major constituents of Teraphim is discussed. Examples of blackboard entries, knowledge sources, and scheduling

heuristics are presented.

Section 4 details the two example Teraphim programs, Identifier and Murder_plot. Discussions of program traces illustrate how Teraphim works.

Section 5 provides a discussion of the conclusions about Teraphim, Prolog, and the blackboard model that may be drawn from consideration of the example programs.

Appendix A includes the Prolog code of Teraphim's main control loop. Appendix B contains the knowledge sources of the example program Identifier, and Appendix C presents the knowledge sources of the Murder-plot example program.

2. BACKGROUND

The blackboard architecture for knowledge-based systems first appeared in the HEARSAY-II speech-understanding system (BARR81). Although the HEARSAY-II system was only moderately successful at its task of understanding connected speech, the blackboard architecture it employed was recognized as a powerful framework for problem solving (WATE83). Since HEARSAY-II other blackboard architecture systems have been written to solve problems in a variety of domains. More recently, some domain-independent blackboard architecture systems such as HEARSAY-III (BALZ80, ERMA81), BB1 (HAYE84, HAYE85b), and AGE (NII79, NII80) have appeared. Blackboard concepts have also been used in commercial expert system building tools such as Inference Corporation's ART. Although no two of these systems have been exactly the same, they share a set of basic blackboard architecture concepts (HAYE83a).

2.1. Development of the Blackboard Architecture

Conventional programs generally apply algorithms to reach solutions, whereas problems for which no algorithmic procedures are known must be solved through the application of artificial intelligence techniques. Consequently, artificial intelligence systems, and especially knowledge-based systems, often have different organizations from conventional programs. In conventional programs, execution is controlled by explicit procedure calls. In contrast, many artificial intelligence programs have data-driven organizations, in which the data indirectly control execution. One method of data-driven programming, pattern-directed programming, has frequently been used in writing expert systems.

In pattern-directed programming, patterns occurring in the data cause appropriate actions to be taken. Rule-based systems are a further development of pattern-directed inference systems. In a rule-based system, knowledge about what deductions should be made from the data is encoded in a set of antecedent-consequent rules in which the consequent portion of a rule expresses the conclusions that can be drawn from the presence of data

matching the antecedent portion (BARR81).

Production systems can be considered an elaboration of rule-based systems (BARR81, WATE78). Most production systems consist of three main components:

1) a data base or context, which contains the problem-specific data;

2) a rule base, which is a collection of independent condition-action production rules;

3) an interpreter, which matches the data-base patterns with the conditions of the production rules (thereby determining which rules could potentially have their action portions executed) and controls the execution of the action portions of the appropriate production rules.

The interpreter of a production system acts in cycles, each of which has three phases:

1) a matching phase, when those production rules with conditions that can be satisfied by the current contents of the data base are triggered;

2) a conflict resolution phase, when one of the (possibly many) triggered production rules is selected for execution of its action portion;

3) an action phase, when the selected action is carried out.

The actions of production rules typically result in modifications to the data base that allow other production rules to be triggered. Usually a production system cycles through these three phases until either the problem is solved or no triggered rules remain.

Production systems gradually build a solution in the data base as new terms are added, cycle by cycle. The rules are applied in an opportunistic rather than a strategic fashion: that is, the order in which the rules will be used depends on the history of the system and is not explicitly determined beforehand. The individual rules are independent of each other to a greater degree than are the procedures in a conventional program, and this allows the builder of a production system to add or delete rules easily. These key advantages of the production system architecture are inherited by the blackboard architecture.

The blackboard architecture that originated with the HEARSAY-II system is a development of the production system organization. The blackboard itself is a special implementation of the data base of the production system organization; systems with a blackboard architecture generally restrict the way information posted on the blackboard can be recognized or altered. In addition, the blackboard generally is organized so the "position" of information on the blackboard is significant in the effect that information can have on the developing solution to the problem. The knowledge sources of the blackboard architecture usually retain the basic condition-action structure of production rules, but typically the knowledge sources are more complex than are production rules. Many blackboard systems have extended the range of constructs that can appear as conditions or actions of a knowledge source, so that often these can be arbitrarily complex. The simple interpreter of the production system organization is also extended to permit more sophisticated conflict resolution, sometimes by providing the interpreter or scheduler with its own blackboard and knowledge sources devoted to the resolution of the problem of which action to perform next.

The generality and flexibility of the blackboard architecture are the major points in its favor (HAYE83a, HAYE83b, HAYE85a, WATE83). The independence of the knowledge sources means that blackboard-based systems can draw on diverse forms of knowledge. The modularity of the knowledge sources allows blackboard-based systems to grow incrementally - knowledge sources can easily be added or removed as the system is built. In contrast to the constrained structure of production rules, the knowledge sources can encode more diverse kinds of knowledge. Treating the scheduling problem as a problem that can itself be solved by a blackboard-based system permits systems to have some degree of self-knowledge. This separation of the scheduling problem from the domain problem illustrates how a blackboard architecture can make decomposition of a complex problem into manageable sub-problems easier. These assets of the blackboard architecture have suggested to some that the blackboard architecture may be applicable to many kinds of problem solving (HAYE83a, WATE83).

2.2. Blackboard-based Systems

Of special interest are the generic or domain-independent blackboard-architecture systems that have been written since HEARSAY-II, including HEARSAY-III, BB1, and AGE. Although these three systems were developed independently, they have much in common (HEARSAY-III and BB1 are more similar to each other than either is to AGE; AGE can also be used to construct non-blackboard expert systems, but only the blackboard uses of AGE are of interest here). It is possible to discuss the blackboard architecture in general terms by focusing on the common features of these programs. Differences in terminology between these systems sometimes obscure their similarities, so in the following discussion a single eclectic terminology will be used to describe the components of all three.

Knowledge-based systems constructed on the blackboard model include four distinctive components (HAYE83a):

1) a blackboard, which is a global data base that organizes entries posted to it by the knowledge sources, and that mediates the interactions of those knowledge sources;

2) the blackboard entries, which are essentially intermediate results or partial solutions to the problem the system is trying to solve;

3) a set of independent knowledge sources, which respond to changes in the entries posted to the blackboard, most often by making further alterations to the blackboard;

4) a control mechanism or scheduler which determines the action that the system will take next by choosing from the set of possible actions recommended by the various triggered knowledge sources.

Each of these blackboard architecture components is discussed below.

2.2.1. Execution cycle of blackboard-based systems

Although the independence of the knowledge sources that is characteristic of blackboard-based systems makes such systems good candidates for the use of parallel processing, to date almost all blackboard-based systems have been implemented for sequential

processing, so that only one action at a time is possible for the system (HAYE83a, NII80, NII85). Therefore, the execution of a blackboard-based system usually must proceed through cycles like those of a production system.

For example, the following phases of each cycle of BB1 may be distinguished (HAYE84):

1) Decide which knowledge sources should be triggered by the current contents of the blackboard. Each triggered knowledge source (KS) is represented on a portion of the blackboard called the agenda (or To-Do-Set) by a knowledge source activation record (KSAR).

2) Select or schedule one of the KSARs from the agenda to be fired (i.e., to have the action portion of its KS executed) based on some set of scheduling rules, which may be subject to change.

3) Execute the scheduled KSAR, thereby probably altering the contents of the blackboard and thus allowing other KSs to be triggered.

The system will repeat this cycle until the problem is solved or there are no more KSARs to choose from. The execution cycles of HEARSAY-III and AGE could be expressed in similar terms.

2.2.2. The Blackboard

Communication between the various components of a blackboard-based system is accomplished by means of entries posted on the blackboard, which serves as a globally-accessible data-base.

In HEARSAY-III and BB1 the blackboard is divided into control and problem partitions, since in these systems two different problems must be solved in parallel - the domain problem, and the control or scheduling problem, which is the problem of how to choose which KSAR should be executed next. AGE does not use the same blackboard techniques for solving the control problem as it does for the domain problem.

In HEARSAY-III, BB1, and AGE the blackboards have a hierarchical internal structure. Entries exist at a given level of abstraction within their blackboard, and are connected by special links to other entries at higher or lower levels.

For example, in the PROTEAN protein-structure analysis system built with BB1 the problem blackboard has the following levels (HAYE84):

LEVEL	CONTENTS
Secondary-Anchor	Alpha-helices, etc. used to anchor partial solutions
Secondary	Portions of secondary structure incorporated into partial solutions
Blob	Peptides or side-chains
Atomic	Atoms

Note that entries at one level represent abstractions of several entries at the next lower level. This blackboard is also structured along the horizontal dimension, with entries that represent knowledge about adjacent parts of the molecule being analyzed posted close to each other. HEARSAY-III's object-oriented database language also leads to a hierarchy of entries, and AGE also uses a hierarchy of analysis levels to represent the developing solution to its domain task. Hayes-Roth has asserted that this hierarchy of levels of abstraction is an essential feature of the blackboard architecture (HAYE83a).

2.2.3. Blackboard Entries

In each program the syntax of the entries posted to the blackboard is different. In HEARSAY-III the blackboard is accessed through an object-oriented database language called AP3, which is itself built on INTERLISP (BALZ80, ERMA81). AGE and BB1 have their own ways of representing the blackboard entries, based on LISP. The different blackboards in these systems have different kinds of entries posted on them.

For example, in the PROTEAN application of BB1, a problem blackboard entry at the "Secondary-Anchor" level might be as follows (HAYE83a):

Name	Secondary-Anchor2
Type	Alpha helix
Number	1
Sequence	(ASN5 TRP6 LEU8 ILE9 TRP10)
Number-AA	6
Percent-AA	3
Internal-Constraints	Nil
External-Constraints	5
Constraints-to-Other-Structures	((Secondary2 2 (5 6)) (Secondary3 0 Nil)) (Secondary4 2 (3 4)) (Secondary5 1 (2)))
Parameters	((Origin 0 0 0) (Reference 4 0 0) (Tip 0 0 14.0))

Note that the Constraints-to-Other-Structures portion of this entry expresses the entry's relationship to four other entries at a lower level of the blackboard.

HEARSAY-III depends heavily on its use of the AP3 database language (BALZ80, ERMA81, WATE83). The blackboard entries in HEARSAY-III are AP3 units. The units are members of a hierarchy of types in which units of each type inherit the properties of their supertypes. Units may have complex structures; for example, a unit can represent a choice set, which is a collection of mutually exclusive alternative hypotheses. The units are connected to each other by roles that represent their interrelationships. A HEARSAY-III blackboard is thus a collection of nodes connected by labeled arcs.

AGE represents its partial solutions in a similar way (NII79). In AGE, each blackboard entry is part of a semantic net linked to other entries by "expectation" links from higher-level entries and "reduction" links from lower-level entries.

The entries on the control blackboards of HEARSAY-III and BB1 have specialized formats of their own, which do not differ in principle from the formats of the problem blackboard entries.

In HEARSAY-III, BB1, and AGE, KSARs represent a distinct class of blackboard entry with its own format. Each KSAR must record information about the knowledge source that produced it and the context in which it was created. For example, in HEARSAY-III each KSAR stores the triggered knowledge source's name, the AP3 context that matched the knowledge source's triggering pattern, and the values of any variables instantiated by the match (ERMA81). HEARSAY-III also can associate various kinds of status information with a KSAR that affects the scheduling of the KSAR. Both AGE and BB1 have similar methods of recording information about knowledge source activations.

2.2.4. Knowledge Sources

In a blackboard-based expert system, knowledge about how to solve problems is expressed as a number of independent knowledge sources. HEARSAY-III, BB1 and AGE each have their own method of defining and applying knowledge sources. Although there are differences between the three systems, all three demonstrate the influence of the design of the knowledge sources in HEARSAY-II.

In HEARSAY-III, each knowledge source is considered to be a large-grained production rule. Knowledge sources have three parts (ERMA81, WATE83):

1) a triggering pattern, which is an AP3 pattern that must be matched by data on the blackboard for the knowledge source to be triggered;

2) some immediate code, which is executed when the knowledge source is triggered, and that may act to bind useful information to the KSAR produced;

3) a body, which is executed when the KSAR is selected to be fired.

Knowledge sources in BB1 are similar (HAYE84, HAYE85b). Each knowledge source has two main parts, a condition and an action. The condition consists of a trigger and a precondition, both of which must be true for the knowledge source to be triggered. The action consists of one or more rules which are executed when the KSAR from the knowledge source is fired. Each rule has a left-hand side and a right-hand side, and is essentially a

production rule. In addition, each knowledge source has other information attached to it, such as a description in English of the knowledge source's behavior, estimates of the knowledge source's efficiency and reliability, and descriptions of which blackboards it reads from or writes to.

An AGE knowledge source consists of a set of rules that are grouped together because they are related in some way (NII79, NII80, WATE83). Each knowledge source has a list of preconditions that must be met for it to be triggered. Because the knowledge sources in AGE are of indeterminate size, it would be possible for an expert system written with AGE to have only one knowledge source, although that would defeat the special properties of the blackboard architecture. As in BB1, each AGE knowledge source has associated with it terms that describe its action to the system.

2.2.5. Control of the Scheduler

In HEARSAY-II, all of the knowledge sources were specialized for solving the problem of speech understanding. The problem of choosing which KSAR should be executed for each cycle was solved by the system's scheduler, which embodied a domain-specific strategy of problem solving. Given that the blackboard architecture is useful for solving problems, it is an obvious step to make a blackboard-based system that treats the problem of resolving which KSAR should be executed as a problem to be solved with its own blackboard and set of knowledge sources. HEARSAY-III and BB1 adopt this technique, and thus might be termed blackboard-controlled blackboard-based systems (BALZ80, ERMA81, HAYE84, HAYE85b). AGE does not use the same methods for solving the control problem as it does for solving the domain problem. Instead, AGE allows the user to select one of several possible predefined control strategies (NII79, NII85, WATE83).

In a HEARSAY-III system, knowledge about solving the control problem does not reside in the scheduler, which is no more sophisticated than the interpreter of a production system (BALZ80, ERMA81, WATE83). Instead, HEARSAY-III uses control knowledge

sources that can be triggered by the contents of the control blackboard as well as by the contents of the problem blackboard. Control knowledge sources may change the priorities of KSARs or post other information that affects the operation of the scheduler. Thus, knowledge about the possibility of actions that contribute to solving the domain problem (competence knowledge) is contained in the problem knowledge sources, whereas knowledge about the relative desirability of these actions (performance knowledge) is embodied in the control knowledge sources.

BB1 distinguishes between control and problem knowledge sources in essentially the same way (HAYE84, HAYE85b). BB1 especially emphasizes making explicit decisions about which control strategy to use, and about changing that strategy during solution of the problem if necessary.

2.2.6. Other Features

Expert systems must often work with errorful or contradictory data. Often rules for reasoning with uncertain data must be provided. No particular method for dealing with uncertainty is associated with the blackboard architecture, and AGE, for example, has no way to represent uncertainty (NII79, WATE83). But blackboard-architecture systems can easily accommodate reasoning about uncertainty. In BB1 (HAYE84), each knowledge source has a measure of reliability associated with it. The reliability of the results of the execution of a knowledge source is a function of the reliability of the knowledge source and the reliability of the data that triggered the knowledge source. Competing hypotheses may have different degrees of reliability associated with them, allowing the system to decide which one to believe. This decentralized method of representing the relative credibility of the system's knowledge is more in keeping with the blackboard architecture's structure than is the use by HEARSAY-II of a special knowledge source that rates the credibility of all blackboard hypotheses, as this one knowledge source requires knowledge about all aspects of the problem being solved (LESS77).

Expert systems that can explain their problem-solving behavior inspire greater confidence in their results than systems that cannot do so. Neither AGE nor HEARSAY-III has an explanation facility (WATE83). BB1 pauses during each cycle after selecting a KSAR for execution but before actually executing it; the user can ask the system to explain the basis of its decision at this point or can choose a different KSAR to be scheduled. The explanations consist of statements about what selection rules were operative and the rationales behind them (provided by the authors of the rules involved) (HAYE84). However, the user can appreciate the process by which any final solution is reached only by examining a trace of the system's execution. Since the opportunistic approach to problem solving used by blackboard-architecture systems is different from the depth-first strategy used in most human problem solving, the program trace may be difficult for the user to understand.

One characteristic of human experts that is not shared by most machine experts is the ability to learn. Although it would be difficult for most expert systems to learn anything about the problem domain, since they cannot have access to the world where domain facts can be verified, a blackboard-control system can access all existing information about the limited domain of its own scheduling knowledge sources. In BB1 provision is made for the system to create new scheduling knowledge based on the success of its current set of scheduling knowledge sources (HAYE85a). BB1 also can ask an expert monitoring its execution to explain why the expert over-rules any of BB1's scheduling decisions; the expert's answers can cause new scheduling knowledge sources to be constructed. This learning ability of BB1 is outside the scope of the present project.

AGE and BB1 both have facilities to let users construct system components in an interactive dialogue (HAYE84, NII79). While such abilities are useful (especially to novice users - that is, anyone besides the system designer), they will not be considered further in this paper.

3. IMPLEMENTATION OF TERAPHIM

Teraphim is a domain-independent framework for creating blackboard-based expert systems. Teraphim is written in C-Prolog, since pattern-directed programming is easy in Prolog. Teraphim includes the basics of a blackboard, a blackboard-controlled scheduler, the ability to ask the user for additional data, and a simple explanation facility based on tracing the origins of entries on the blackboard. Teraphim thus provides a tool for experimentation with the knowledge representation schemes and the problem-solving strategies required to solve problems in whatever domain is of interest, as well as serving as an experiment in the suitability of Prolog for writing such systems. Teraphim is a generic system, and as such is not particularly efficient. Rather, Teraphim should be regarded as a prototyping system. Although it resembles other generic blackboard architecture systems like HEARSAY-III, AGE, and BB1, Teraphim is not an attempt to directly imitate any other system; instead, it is a particular implementation of the concepts of the blackboard architecture common to all such systems.

3.1. Execution Cycle

Although blackboard-based systems have potential for parallel execution, Teraphim is a serial system. Teraphim's execution cycle is similar to that of other serial blackboard systems:

1) Matching phase: The triggering patterns of the knowledge sources are compared with the current state of the blackboard to determine which knowledge sources should be triggered; a knowledge source activation record (KSAR) is created for each context in which a knowledge source can be triggered and the KSARs are posted on the agenda blackboard, joining the KSARs posted there in previous cycles.

2) Conflict resolution phase: One of the KSARs on the agenda is selected for execution; the selection process is determined by the contents of the control blackboard, where decisions about the selection criteria are posted. If several KSARs are equally rated by the scheduling heuristics, one is chosen at random, adding an element of non-determinism to

Teraphim's execution. If no suitable KSAR can be found, execution terminates.

3) Action phase: The knowledge source that produced the selected KSAR is executed in the triggering context; this usually results in changes to the control or problem blackboards. The KSAR is removed from the agenda. If the all of the problems presented to the program are solved, cycling stops. Otherwise, the system returns to step 1.

The Prolog code of Teraphim's main control loop is included in Appendix A.

1.1. Blackboard

The distinguishing feature of the blackboard-based architecture is a central communication medium known as a blackboard. The various independent parts of a system with a blackboard architecture communicate with each other by reading information from and posting information to this blackboard.

Teraphim uses the Prolog database as its blackboard. The database of a Prolog program is not normally structured, but such structure can be simulated. Teraphim divides the blackboard into three sections by dividing the knowledge sources and other active components of the system into groups and restricting access to various blackboard entry types to the members of one of the groups. The four groups of active components are:

1) The scheduler is the executive of the Teraphim system, responsible for interpreting and executing the knowledge sources. In each cycle the scheduler selects one of the knowledge source activation records from the agenda and executes it. The scheduler halts the program if either there is no suitable KSAR to be executed or all of the problems presented to the system are solved. Teraphim's domain-independent scheduler would not ordinarily be modified by the user to deal with a specific application.

2) The scheduling heuristics respond to the decisions posted on the control blackboard and apply these decisions to the KSARs on the agenda, thereby jointly determining which KSAR will be chosen for execution by the scheduler. Teraphim provides a number of general-purpose scheduling heuristics, which might be supplemented by the user with

application-specific heuristics, although the generic ones are intended to be sufficient for most problems.

3) The control knowledge sources respond to the contents of the problem and control blackboards by posting new decisions about the desirability of various possible actions on the control blackboard. Teraphim includes a number of generic control knowledge sources, which are sufficient to solve simple problems, but which need to be supplemented by the user if complex control strategies are required.

4) The problem knowledge sources contain most of the application-specific knowledge. It is the problem knowledge sources that react to the entries on the problem blackboard by posting new entries to the blackboard, thereby building up the solution to the problem. Teraphim provides one special problem knowledge source, *find_out*, which allows the system to ask the user for additional data when the information at hand is insufficient for a solution to the problem.

The three partitions of the Teraphim blackboard are:

1) The agenda blackboard: Reasoning about what actions are possible takes place here. Knowledge source activation records and associated entries are posted to the agenda blackboard. Control knowledge sources, scheduling heuristics, and the scheduler can read the entries on this blackboard. The scheduling heuristics and the scheduler may change the contents of the agenda.

2) The control blackboard: Decisions about how KSARs should be selected from the agenda for execution are posted here. Most of the entries on the control blackboard are posted by the control knowledge sources, although some may be specified by the user before execution starts. This blackboard is examined by the scheduling heuristics and by the control knowledge sources.

3) The problem blackboard: The solution to the problem is constructed by the problem knowledge sources here. This is also where any data provided by the user will be posted. Both problem and control knowledge sources monitor the problem blackboard.

Teraphim does not impose any internal organization on any of the three blackboards, although the user may specify a structure for the problem blackboard if the application demands it. For example, in the Identifier test program considered below, the blackboard is assumed to have entries at different "levels of abstraction" that have significance in determining the importance of a term to the developing solution of the problem. This corresponds with the "phoneme to sentence" dimension of the Hearsay-II system. But in the Murder_plot example, instead of using the structure of the blackboard to reflect how close entries are to a solution for the problem, an evaluation function specific to the problem is used, and no structure is imposed on the blackboard. Thus, in Teraphim, internal structure of the blackboard is an option, and different applications might use different degrees of structure for the problem blackboard, including no structure at all.

3.3. Blackboard Entries

If the blackboard is the medium of communication for the knowledge sources of Teraphim, then the entries on the blackboard are the messages. Since Teraphim is written in Prolog, the entries that are posted on the blackboard are Prolog terms.

The agenda blackboard contains the KSARs and associated entries. An example KSAR produced during the execution of the Identifier program is:

```
ksar(tiger,7,7)
rating(7,31).
reliability(7,some).
context(tiger,order(charlie,carnivore),7,7).
context(tiger,color(charlie,tawny),7,7).
context(tiger,pattern(charlie,striped),7,7).
precursors(7,[order(charlie,carnivore),class(charlie,mammal),
        skin(charlie,hair),empty_q,problem(charlie),
        diet(charlie,meat),color(charlie,tawny),
        pattern(charlie,striped)]).
```

The knowledge source *tiger* was triggered on cycle 7, producing KSAR number 7. KSAR number 7 has a rating of 31, determined by the scheduling heuristics currently active, and a reliability computed from the reliabilities of the blackboard entries that produced the KSAR and the reliability of the knowledge source *tiger*. The *context* entries for KSAR 7 preserve the variable bindings that resulted when the triggering pattern of *tiger* was instantiated from the problem blackboard. The *context* entries also record *tiger's* use of this combination of entries, assuring that *tiger* will not be triggered by these same entries on subsequent execution cycles. The *precursors* entry for KSAR 7 collects all of the problem entries that contributed to the KSAR's generation. Because Teraphim's certainty mechanism is non-monotonic, it is possible that the certainties of some of the blackboard entries that determine the reliability of KSAR 7 will be changed between the cycle the KSAR was produced and the cycle on which KSAR 7 is chosen for execution. By comparing the *precursors* entries for the existing KSARs with the problem blackboard entries produced on each cycle, it is possible for Teraphim to determine whether it might be necessary to recalculate the reliability of each KSAR.

KSARs are posted to the agenda blackboard by the scheduler using the *make_ksar* procedure. KSARs are removed from the agenda after execution by the scheduler using the procedure *remove_ksar*. Poisoned KSARs (see below) are also removed using *remove_ksar*, which removes all of the information related to the KSAR from the agenda (except for the *context* entries, which are needed to prevent knowledge sources from triggering again on entries they have already 'seen').

The control blackboard contains a wider variety of entries. The principal types of control blackboard entries and their meanings are as follows:

problem(Name) means that the name of one of the problems that must be solved is *Name*. Entries on the blackboard that refer to the solution of this problem should all contain *Name* as one of their elements so that their context can be recognized by the system. Usually

this entry would be posted by the user, although in more complex programs it might be asserted by one of the knowledge sources.

solved(Name) means that a solution to problem *Name* has been found. If the certainty of this entry exceeds a user-determined threshold, Teraphim will stop working on that problem. Usually this term is posted by an implementation-specific control knowledge source.

plan(Strategy) means that the scheduling heuristics associated with the plan *Strategy* should be made active by a control knowledge source. Plans that are part of Teraphim include *default*, *depth*, and *evaluate*, which are recognized by different control knowledge sources and which cause different scheduling heuristics to be activated. *Plan* entries may be provided by the user or may be posted by control knowledge sources.

active(Method) means that the scheduling heuristic *Method* should be used to evaluate the relative merits of the KSARs on the agenda blackboard. Scheduling heuristics provided by Teraphim include *prefer_recent*, *prefer_easy*, *prefer_deeper*, *prefer_focus*, and *use_evaluation*. *Active* entries are produced by control knowledge sources in response to the presence of *plan* entries on the control blackboard.

focus(Term) means that the scheduling heuristic *prefer_focus* should increase the ratings of those KSARs that would produce new blackboard entries matching *Term* if executed. *Focus* entries are generated by problem-specific control knowledge sources. Unlike *plan* and *active* entries, which usually remain on the control blackboard permanently once posted, *focus* entries generally refer to only one stage of the solution of a problem, and *focus* entries are removed from the control blackboard by control knowledge sources that recognize the satisfaction of the focus's goal.

inhibit(KS,Term) is an instruction to the scheduler that KSARs produced by the knowledge source *KS* and triggered by an entry matching *Term* are not to be considered for execution unless their reliability exceeds the certainty of the *inhibit* term, which derives from the control knowledge source that posted it. The inhibited KSARs are not removed from the

agenda, since the non-monotonic nature of Teraphim's certainty mechanisms means that the reliability of a blocked KSAR may later increase. The *KS* portion of the *inhibit* term may be filled by a Prolog variable, in which case all KSARs using the inhibited term will be blocked.

poison(KS,Term) is similar to *inhibit* except that poisoned KSARs are removed from the agenda permanently. Since evaluating the reliability of pending KSARs is the most time-consuming step of Teraphim's execution cycle, poisoning KSARs instead of just inhibiting them can greatly speed up the execution of the system. Of course, the poisoning of the KSARs cannot be undone, so poisoning is not as safe as inhibiting.

Other terms might also be considered part of the control blackboard - for example, each knowledge source has an *importance* term associated with it, which provides the initial rating of KSARs produced by that knowledge source before the scheduling heuristics are applied to it. It might be useful to have control knowledge sources that change the *importance* of other knowledge sources in response to the progress of the system towards a solution to the problem.

The syntax of the entries that can appear on the problem blackboard is not restricted by Teraphim, except in a few special cases. The choice of a knowledge representation scheme for the partial solutions that are built on the problem blackboard is up to the user. An object-oriented extension to Prolog could be used for most problem blackboard entries, for example. The only restriction is on terms that might result from questions put to the user by Teraphim. Teraphim includes a generic problem knowledge source, which can ask the user for more information if the program cannot generate a satisfactory solution to the problem with the facts at hand. The format of the terms created by the system from the user's replies to these questions is restricted. These terms must have the format *Slot(Object,Value)* - for example, the term *locomotion(clyde,walks)* would be produced to represent the idea that the value in Clyde's "locomotion" slot is "walks". If more flexibility is required, the user must provide his own question-asking knowledge sources.

One special entry that appears on the problem blackboard is the *supports* entry: *supports(Term,KS,List)*, meaning that the problem blackboard entry *Term* was posted by the knowledge source *KS* on the basis of the terms in *List*. The entries on the problem blackboard are tied to one another by the *supports* terms so that it is possible to trace the origin of any term on the blackboard by following the *supports* links. If a problem blackboard entry can be produced in several ways, it will have a number of different *supports* terms connected to it. By following the *supports* links to the terms supplied by the user and using the user's evaluation of the reliability of those original terms, the certainty of the derived terms can be calculated.

Usually problem blackboard entries, once made, are not removed, although the certainty of an entry may be changed during the solution of the problem. Terms with certainties below a user-specified threshold will produce KSARs that are never chosen to be executed, and thus these terms with low certainties will not affect the solution.

3.4. Knowledge Sources

Solving a problem with an expert system generally involves the creation of new information based on information already known. In Teraphim, procedural knowledge about how to produce new information is embodied in problem and control knowledge sources.

Problem knowledge sources communicate with the problem blackboard. It is the problem knowledge sources that respond to the information on the problem blackboard by gradually building a solution to the problem on the problem blackboard. Teraphim provides a single generic problem knowledge source, *find_out*, which can ask the user questions. Otherwise, a new set of problem knowledge sources must be written for each new problem domain.

Control knowledge sources respond to information posted on the problem and control blackboards by modifying the contents of the control blackboard. This affects the scheduler's choice of a KSAR to be executed because the scheduling heuristics evaluate the relative worth of the pending KSARs in response to the decisions posted on the control

blackboard. Teraphim includes generic control knowledge sources and scheduling heuristics sufficient for solving simple problems, but complex solution processes require more specific knowledge sources that must be written by the user.

A knowledge source in Teraphim comprises several elements. Below is a representative problem knowledge source *search_move* from the Murder_plan example program to be presented in Section 4.2. This knowledge source guides the movements through a house of a robot searching the rooms for a weapon:

```
knowledge_source(search_move) :->
        [
         /* Triggering pattern */
         [scheme(Problem,Weapon,Room,Plan,Cost,search,no),
         non(location(Problem,Weapon,Room)),
         adjacent(Room,NewRoom,MoveCost),
         allowable(NewRoom,Plan)],

         /* Immediate action */
         [],

         /* Firing action */
         [NewCost is Cost + MoveCost,
         add_to(Plan,NewRoom,NewPlan)],

         /* Productions */
         [scheme(Problem,Weapon,NewRoom,NewPlan,NewCost,search,no)]
        ].

importance(search_move,5).
type(search_move,problem).
reason(search_move,['Look for weapon in unvisited adjacent room']).
```

Each knowledge source has a body which consists of four lists of terms:

1) A triggering pattern, which is a conjunction of Prolog terms that must all be true (i.e., that can all be instantiated) for the knowledge source to be triggered. Usually, most of these terms will be blackboard entries, but they can also be calls to Prolog procedures that express relations between entries on the blackboard. *Search_move* refers to the problem blackboard

entries *scheme*, *location*, and *adjacent*. *Allowable* is a Prolog procedure that checks to make sure that a proposed new move does not violate any constraints. The special term *non(location(Problem,Weapon,Room))* should be noted; no *non* term is ever posted to any of the blackboards. Instead, a *non* term is always true, but with a certainty that is the complement of the certainty of the inner term (in this case, *location(Problem,Weapon,Room)*), so that if the inner term has a high certainty, the *non* term has a low certainty. If the inner term cannot be matched, the *non* term has the maximum possible certainty.

In this example, *search_move* is triggered when there is a partial search plan which has not reached the room containing the weapon the robot is searching for, and there is an adjacent room that the robot could enter. *Search_move* will propose that the robot enter the new room next.

Every time that Teraphim begins a cycle, it checks all the knowledge sources that share triggering terms with the blackboard entries produced on the last cycle to see if any of the triggering patterns is true. Thus, knowledge sources are triggered by changes to the blackboard. A different knowledge source activation record (or KSAR) is created for each possible instantiation of the triggering pattern. The KSARs and the *context* terms that preserve the instantiated variables are posted on the agenda blackboard.

2) An immediate action, which is a list of Prolog procedure calls that are executed during the matching phase immediately after the knowledge source is triggered. All variables that were instantiated by the triggering pattern will have the same values when the immediate action is carried out. Usually this section of the knowledge source is used to write a message to the user indicating that the knowledge source has been triggered, since this information is useful during program development. Most knowledge sources will have no immediate action once the program is running correctly. *Search_move* does not have an immediate action.

3) A firing action, which is a list of Prolog terms to be matched when the KSAR from

this knowledge source is chosen for execution (or firing) by the scheduler. It is likely that this will occur several cycles after the KSAR was posted, since more than one KSAR is usually produced each cycle. All the variables that appeared in the triggering section of the knowledge source will have the same instantiations as before. Not all knowledge sources will have anything in their firing action sections; for example, a knowledge source that corresponds to a production rule will have only a triggering pattern and a set of productions. The example knowledge source has a series of procedure calls in its firing action. Variables that are instantiated by the firing action transmit their values to the next section of the knowledge source, so the firing action is generally used to build up a new entry to be added to one of the blackboards. For example, *search_move's* firing action computes the cost of the new plan and adds the new room to the robot's path.

4) A production section, which is a list of Prolog terms that will be posted on the blackboard. When each term is posted, the knowledge source and triggering blackboard entries that produced the new entry are recorded in a *supports* entry for each new term. The *supports* term is used by the explanation facility to explain the genesis of the new terms to the user if requested, and is also used to determine the confidence or certainty that the system may place in the truth of the new assertions. Not all knowledge sources will have a production section. For example, some knowledge sources may cause a message to be written to the user without changing any blackboard. *Search_move's* production section causes a new partial plan to be posted to the problem blackboard. This new partial plan might cause *search_move* to be triggered again on the next execution cycle.

There may also be other terms describing a knowledge source. Each knowledge source may have a *certainty* associated with it. This *certainty* reflects the accuracy of the knowledge source and is a measure of the certainty that the system has in the results of the knowledge source's action. If no *certainty* is provided for a knowledge source, the default assumption is that the knowledge source does not introduce any additional error into the system. Each

knowledge source also may have an *importance* attached to it; the greater the *importance* of a knowledge source, the more likely it is that KSAR's produced by it will be chosen for execution by the scheduler. Again, the system will make default assumptions about the *importance*of knowledge sources that were not rated by the user. The *type* of a knowledge source is used by some of the scheduling heuristics; most scheduling heuristics will not change the ratings of KSARs belonging to control knowledge sources. If a knowledge source has no *type* term, Teraphim will assume that it is a problem knowledge source. Each knowledge source may have a *reason* term, which is an English-language explanation or justification of its action that can be used during explanation of the solution.

3.5. Scheduling Heuristics

The control knowledge sources do not affect the KSARs posted on the agenda blackboard directly; instead, they change the contents of the control blackboard, which changes the behavior of the scheduling heuristics. Scheduling heuristics that are active (because some control knowledge source has posted an appropriate *active* term on the control blackboard) change the ratings of each KSAR on the agenda blackboard. The text of *use_focus*, one of Teraphim's generic scheduling heuristics, is:

```
heuristic(use_focus) :->
        [
         /* Trigger */
         ksar(KS,C,Id),

         [
          /* Body */
          focus(Term),
          not type(KS,plan),
          get_used(KS,List,C,Id),
          knowledge_source(KS) :->
                [Cond|[Imm|[Act|[Res|_]]]],
          Cond = List,
          member(Term,Res),
          rating(Id,N),
          weight(use_focus,W),
          New is N + W,
          rerate(Id,New)
         ]
        ].

weight(use_focus,40).
```

The body of a scheduling heuristic is a list with two elements:

1) a trigger, which is matched in turn with each pending KSAR that has not yet been rated by the heuristic, and

2) a body, which is a list of terms to be called. The action of the list of terms changes the ratings of appropriate KSARs by adding some factor to the previous ratings. Because addition is always used to change ratings, it does not matter in what order the various active heuristics re-rate a KSAR. In this example, the scheduling heuristic *use_focus* increases the ratings of KSARs produced by knowledge sources that will post terms matching any current focus. It identifies these knowledge sources by looking at their production sections. Teraphim includes several other generic scheduling heuristics, such as *use_evaluation*, which increases the ratings of KSARs according to a domain-specific evaluation function provided by the user, and *prefer_recent*, which increases the ratings of recently produced KSARs and

thus causes a degree of depth-first behavior.

By activating different sets of scheduling heuristics, the control knowledge sources can affect which KSARs are preferred in the selection process. The set of active scheduling heuristics determines the overall course of the problem-solving process.

In addition to the explicit scheduling heuristics that can be activated (or de-activated) by the control knowledge sources, Teraphim has an implicit scheduling heuristic that always prefers the KSARs with the highest degree of certainty. This preference cannot be turned off by any control knowledge source. However, this built-in preference has a *weight* like those of the explicit heuristics, and by changing this *weight* to zero it is possible to prevent the reliabilities of the KSARs from affecting their ratings. This will still leave any thresholds for KSAR execution or solution acceptance intact.

3.6. Uncertainty

Expert systems frequently must reason with uncertain or contradictory information. Teraphim provides simple methods for dealing with uncertainty in the terms posted to the blackboards and in the operation of its knowledge sources. Teraphim's most primitive uncertain reasoning terms are separate from the rest of the system and may be modified by the user to fit various models of inexact logic.

Each blackboard entry provided by the user and each knowledge source can have a certainty associated with it. Entries or knowledge sources for which no certainty has been specified are assumed to have maximum certainty.

When a knowledge source posts a new entry derived from previously posted entries, the certainty of the new entry will be the conjunction of the certainties of all the triggering entries and the certainty of the posting knowledge source. If an entry could be produced in several different ways, each of which has a different degree of certainty associated with it, then the certainty of the entry will be the disjunction of the certainties of the different possi-

ble origins of that entry. Since the different ways of producing the same entry may be discovered at different times during the solution of the problem, the certainty of an entry may change after the entry is posted.

The special triggering pattern *non(Term)* has been mentioned above. This pattern can always be matched. Its certainty is the complement of the certainty of *Term*. If *Term* is not posted on the blackboard, then the certainty of *non(Term)* is assumed to be the maximum possible.

There are thus three primitive inexact reasoning functions required: conjunction, disjunction, and complement. Teraphim adopts the approach of the Inexact Reasoning Module (IRM) by collecting these primitive operations into one set of procedures which is then used by all of the other components of the system to reason about uncertainty (LECO86). In the examples discussed below, a fuzzy set theory approach to these three functions is used, so conjunction corresponds to minimum, and disjunction is equivalent to maximum. In addition, Teraphim allows the user to choose whatever terms for the various certainty levels are convenient. In the example programs, the levels are: *none*, *poor*, *some*, *good*, and *total* certainty. But because the primitive operations for inexact reasoning are separate from the rest of the system, users can write other definitions of conjunction and disjunction, and can use other names for the levels of certainty or have a different number of levels of certainty.

Each level of certainty, whatever its name is, must have a numerical equivalent so that the scheduler can determine the effect the certainty of KSARs will have on their ratings.

An important use of certainty in the Teraphim system involves two user defined thresholds, the minimum certainty that a KSAR must have to be considered for execution, and the minimum certainty that a problem solution must have to satisfy the system and allow it to stop looking for solutions to that problem.

Although being able to deal with uncertain knowledge is useful, it is expensive. Experience shows that up to one-half of the processor time required to solve a problem with

Teraphim is consumed by calculations of the certainties of KSARs.

3.7. Asking the user questions

The user of an expert system might not know how much information the system will require to solve the problem. Although Teraphim normally reads in the initial facts from a file, it is possible that the user will not have provided enough information for the system to solve the problem. Therefore, Teraphim includes a generic problem knowledge source, *find_out*, which is capable of asking the user for more information if the problem cannot be solved. The designer of the problem knowledge sources may designate some of the blackboard terms as being askable; the term *askable(Functor)* means that the system is allowed to ask about the value of terms with the functor Each askable term must have the format *Slot(Problem,Value);* the designer must specify the range of legal values for *Value*. If the system runs out of KSARs without solving the problem, it will search for knowledge sources that could be fired if more of the *askable* terms were known. Then it asks the user about the value and certainty for each term. If the user does not know the correct value, the answer "unknown" will keep the system from asking further questions about that term but will not cause any knowledge sources to be triggered. An example of Teraphim asking questions is shown in the Identifier program discussed in Section 4.1.

Teraphim can be run in three modes: *normal*, *verbose*, or *crawl*. In *verbose* mode, the agenda is displayed every cycle, and the KSAR that will be executed is noted. Terms posted to the blackboard are echoed to screen. In *crawl* mode, the system pauses after choosing a KSAR but before executing it. The user can interrupt execution at this point and display the blackboard entries, change the contents of the blackboards, or select a different KSAR to be executed.

3.8. Explaining the results

Expert systems that can explain their results are easier to debug and inspire greater confidence in the correctness of their solutions. Teraphim has a simple explanation facility that allows users to trace the development of its problem solutions. After the system halts, the user can inquire about the origin and certainty of any of the terms on the problem blackboard. Teraphim permits the designer of the problem knowledge sources to specify English translations of the problem blackboard terms. A sample dialogue with the explanation facility is shown in Section 4.1. with the Identifier program example.

4. EXAMPLE USES OF TERAPHIM

Two example expert systems were written to illustrate Teraphim's capabilities. Since the problems solved by expert systems are frequently classified as analysis problems or synthesis problems, one expert system of each type was constructed.

4.1. The Identifier Program

The generic control knowledge sources of Teraphim are sufficient to control expert systems with knowledge sources that are similar to production rules. The first example program was written as the Teraphim version of a toy rule-based system called Identifier described in (WINS85). The problem is to provide a way for Robbie the robot to identify an animal from its description. Winston wrote a set of 15 if-then rules for this problem. In Appendix B are Winston's rules recast as Teraphim knowledge sources. For example, the knowledge source *penguin* represents Winston's rule:

I14	If	the animal is a bird it does not fly it swims it is black and white
	then	it is a penguin

The 16 knowledge sources required to translate this knowledge for Teraphim may be summarized as follows:

Knowledge Source	Summary
mammal_1	An animal with hair is a mammal
mammal_2	An animal that gives milk is a mammal
bird_1	An animal with feathers is a bird

bird_2	An animal that flies and lays eggs is a bird
carniv_1	A mammal that eats meat is a carnivore
carniv_2	A mammal with pointed teeth, claws, and forward-facing eyes is a carnivore
ungulate_1	A mammal with hoofs is an ungulate
ungulate_2	A mammal that chews a cud is an even-toed ungulate
cheetah	A carnivore with a tawny colored spotted coat is a cheetah
tiger	A carnivore with a tawny colored striped coat is a tiger
giraffe	An ungulate with a long neck and long legs and a tawny coat with spots is a giraffe
zebra	An ungulate with black and white stripes is a zebra
ostrich	A bird that does not fly and is black and white is an ostrich
penguin	A bird that swims and does not fly and is black and white is a penguin
default_bird	A bird that flies is an albatross
solution	An animal is identified if its species is known

In this example, all of the knowledge sources are assumed to be perfectly reliable. The only uncertainty in the solution comes from uncertainty in the observations presented to the

system. A simple range of named certainty factors (*none, poor, some, good,* and *total* certainty) was used, with fuzzy-set versions of conjunction and disjunction. A solution is required to have at least *good* certainty before it is considered adequate, and KSARs are required to have at least *some* certainty to be chosen for execution.

The problem blackboard for the Identifier program is structured, with four levels of increasing priority as follows:

LEVEL	MEANING
observation	observable facts about animals
class	the class of the animal
order	the order of the animal
species	the species of the animal

KSARs that produce blackboard entries on lower levels of the blackboard will be preferred. Note that the user provides Identifier with terms on the observation level only; the other entries are produced by the system.

In one run of this system, the starting observations presented to the system were:

```
charlie gives milk
certainty = poor
charlie is tawny
certainty = total
charlie has a striped pattern
certainty = total
charlie eats meat
certainty = some
charlie walks
certainty = total
charlie is a problem
certainty = total
```

This is not enough information about the animal (a tiger) for the system to identify it. Instead, the system must ask for more information. The program is allowed to ask questions about any of the directly observable properties of the animal - e.g. its color, if it has feathers or not, or if it eats meat. Of course, the blackboard entries are actually Prolog terms and not phrases in English. Not included in Appendix B are the terms that specify how translation is to be done. An example of such a term is:

```
english(color(Animal,Shade),[Animal,is,Shade]).
```

Other terms provide the system with information about how to phrase the questions it can ask the user:

```
question(color,Animal,[what,color,Animal,is]).
```

Although in this example only one problem (identifying Charlie) is presented to the system, it would also be possible to provide data about several different animals to be identified.

Note that no special control knowledge sources were written for Identifier. The knowledge source *solution* is required to stop the program after an acceptable identification is made. A similar knowledge source must be written for each Teraphim program.

When the Identifier program is run with the observations about the tiger, the first cycle of the system is as follows:

```
Cycle 1
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      10      poor
ksar(depth,1,2)         100     total
ksar(default,1,3)       100     total
Choosing best ksar
Choosing ksar 2
Execute best ksar
active(prefer_deeper)
```

The KSARs from the generic control knowledge sources have high ratings (because their parent knowledge sources have high importance ratings) and thus are chosen first. These generic knowledge sources specify the control heuristics that will affect the selection of KSARs from the agenda. For example, in this cycle the KSAR from the knowledge source *depth* was chosen for execution. This produced the new blackboard entry *active(prefer_deeper)*, which causes the system to use the scheduling heuristic *prefer_deeper* to rate the KSARs on the agenda. This heuristic causes KSARs posting entries to the lower levels of the problem blackboard to be preferred. This preference gives the system a semblance of depth-first behavior since partially-solved problems will have higher priorities than others. Note that KSARs 2 and 3 have the same rating and certainty; the choice between them was made randomly.

After all of the control knowledge sources have been executed, the only KSAR remaining on the agenda is the one produced by *mammal_2*, which was generated in response to the observation that the animal gives milk. However, the certainty of this KSAR is only *poor*, so it cannot be chosen. The lack of any worthwhile KSARs causes the generic knowledge source *find_out* to trigger for each "interesting" question that could be put to the user at this point. A question is interesting if the answer to the question, in combination with known blackboard entries or the answers to other questions, will allow one of the unused knowledge sources to fire. Since the class of the animal must be known before any of the other knowledge sources can be used, questions that can determine the class of the animal are generated. Then Identifier asks the question:

```
Cycle 4
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      20      poor
ksar(find_out,4,4)      -2      total
Choosing best ksar
Choosing ksar 4
Execute best ksar
In order to use mammal_1
I need to know what kind of skin charlie has
hair
feathers
or unknown
Which is it?
|: hair.
Certainties range from none to total
Which is it?
|: total.
skin(charlie,hair)

Cycle 5
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      20      poor
ksar(mammal_1,5,5)      20      total
Choosing best ksar
Choosing ksar 5
Execute best ksar
class(charlie,mammal)
```

The program will repeat its question if the user does not respond with one of the choices presented. Note that the question identifies the knowledge source for which the question is trying to find triggering information.

The information that Charlie has hair is enough for Identifier to determine that Charlie is a mammal. Since the class of the animal is now known with *good* or better certainty, no further effort is expended on deducing Charlie's class.

The program now proceeds to identify Charlie as a tiger:

```
Cycle 6
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      20      poor
ksar(carniv_1,6,6)      26      some
Choosing best ksar
Choosing ksar 6
Execute best ksar
order(charlie,carnivore)

Cycle 7
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      20      poor
ksar(tiger,7,7)         31      some
Choosing best ksar
Choosing ksar 7
Execute best ksar
species(charlie,tiger)

Cycle 8
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      20      poor
ksar(solution,8,8)      150     some
Choosing best ksar
Choosing ksar 8
Execute best ksar
I have determined that charlie is a tiger
solved(charlie)
```

However, the problem cannot be considered solved at this point because the certainty of the solution is only *some*. The problem is that the order of Charlie (carnivore) was determined from the observation that Charlie eats meat, which had a low certainty. Identifier now uses *find_out* to see if a more certain order identification can be made. There are two possibilities: Charlie could be shown to be a carnivore using the knowledge source *carniv_2*, or Charlie might be an ungulate. The system asks some questions to sort out the problem:

Cycle 10
Create ksars
Rating ksars

ksar	rating	certainty
ksar(mammal_2,1,1)	20	poor
ksar(find_out,10,9)	-1	total
ksar(find_out,10,10)	-1	total
ksar(find_out,10,11)	-1	total
ksar(find_out,10,12)	-1	total

Choosing best ksar
Choosing ksar 11
Execute best ksar
In order to use carniv_2
I need to know which way the eyes of charlie face
forward
sideways
or unknown
Which is it?
|: forward.
Certainties range from none to total
Which is it?
|: good.
eyes(charlie,forward)

Cycle 11
Create ksars
Rating ksars

ksar	rating	certainty
ksar(mammal_2,1,1)	20	poor
ksar(find_out,10,9)	-1	total
ksar(find_out,10,10)	-1	total
ksar(find_out,10,12)	-1	total

Choosing best ksar
Choosing ksar 9
Execute best ksar
In order to use carniv_2
I need to know what kind of teeth charlie has
pointed
blunt
or unknown
Which is it?
|: pointed.
Certainties range from none to total
Which is it?
|: good.
teeth(charlie,pointed)

```
Cycle 12
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      20      poor
ksar(find_out,10,10)    -1      total
ksar(find_out,10,12)    -1      total
Choosing best ksar
Choosing ksar 12
Execute best ksar
In order to use ungulate_2
I need to know if charlie chews a cud
yes
no
or unknown
Which is it?
|: unknown.
Certainties range from none to total
Which is it?
|: total.
cud(charlie,unknown)

Cycle 13
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      20      poor
ksar(find_out,10,10)    -1      total
Choosing best ksar
Choosing ksar 10
Execute best ksar
In order to use carniv_2
I need to know what kind of feet charlie has
claws
hoofs
or unknown
Which is it?
|: claws.
Certainties range from none to total
Which is it?
|: total.
feet(charlie,claws)
```

```
Cycle 14
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mammal_2,1,1)      20      poor
ksar(carniv_2,14,13)    26      good
Choosing best ksar
Choosing ksar 13
Execute best ksar
order(charlie,carnivore)

All problems solved!
```

Note the user response "unknown" to the question about cud chewing. A blackboard entry that does not have a value useful to any of the knowledge sources will still prevent the system from asking more questions about this property of the animal, since the user cannot be asked two questions about the same observation.

No new blackboard entry for Charlie's species is produced, but the new *order* term has a certainty of *good*, and this increases the certainty of the species identification to *good*. Identifier considers the problem solved and halts. The user can now ask to see all of the problem blackboard entries.

After Identifier stops, the user can ask for a discussion of the reasoning used by the system to deduce each of the blackboard entries:

```
| ?- discuss.
|: charlie is a tiger.
charlie is a tiger deduced by tiger
from -
charlie is a carnivore
charlie is tawny
charlie has a striped pattern
with certainty good
|: charlie is a carnivore.
charlie is a carnivore deduced by carniv_2
from -
charlie is a mammal
charlie has pointed teeth
charlie has claws
charlie has forward-facing eyes
with certainty good
|: carniv_2.
Any mammal with pointed teeth, claws, and
forward-facing eyes is a carnivore
With certainty total
|: charlie is a mammal.
charlie is a mammal deduced by mammal_1
from -
charlie has hairy skin
with certainty total
|: charlie has hairy skin.
charlie has hairy skin deduced by user
with certainty total
|: user.
This knowledge source is a human!
with certainty total
|: done.
done
```

Notice that the identification of Charlie as a carnivore first was made because of the observation that Charlie eats meat. But the questions asked during the run established a better reason for this deduction, so the new justification appears in the discussion.

This example problem took 59 CPU seconds to solve, running in C-Prolog on a multi-user computer. Execution was slowed since the program was in *verbose* mode; in *terse* mode, which does not print out the lists of KSARs or the blackboard entries, the same example took 51 seconds of CPU time.

4.2. The Murder_plot Program

Teraphim is not limited to imitating production systems or to solving only analysis problems. Another program, called Murder_plot, was written to demonstrate the use of Teraphim to solve a synthesis problem (in this case, a planning problem).

The problem is a variant of the sort of planning problems typically solved by the STRIPS program: Robbie the robot lives in a house with his owner. One day, Robbie becomes tired of constantly solving blocks-world problems and decides to murder his owner and dispose of the body. This problem can be decomposed into three subproblems:

1) Find a weapon to kill the owner with;

2) Find the owner and kill him with the weapon;

3) Hide the owner's body somewhere.

There are two weapons in the house: a knife in the kitchen and a gun in the den. Each has a different certainty of being able to kill the owner. There are two possible places to dispose of the corpse: the cellar or the lawn. Each has a different reliability as a hiding place. The robot and the owner each start in different rooms of the house, which is represented by a network of linked nodes (the rooms). Robbie is to find the shortest path through the house that will accomplish his goal and have an acceptable certainty of success. Omitting the house, the starting facts are:

```
the robot is in the family_room
certainty = total
the gun is in the den
certainty = total
the knife is in the kitchen
certainty = total
the owner is in the entry
certainty = total
gun is lethal
certainty = some
knife is lethal
certainty = good
could hide body in lawn
certainty = good
could hide body in cellar
certainty = total
```

The 22 knowledge sources needed to solve this problem are listed in Appendix C, and may be briefly summarized as follows:

Knowledge Source	Summary
searchfocus	Establishes a focus on the search for a weapon
end_search	Removes the search focus when the weapon is found
stalkfocus	Establishes a focus on the search for a path from the weapon to the owner
end_stalk	Removes the stalk focus when the owner is found
hidefocus	Establishes a focus on the search for a place to hide the body
end_hide	Removes the hide focus when the body has been hidden

mergefocus	Establishes a focus on merging the partial plans into a solution
end_merge	Removes the focus on merging plans
beg_search	Begins the search for a weapon at the robot's initial location
beg_stalk	Begins the search for the owner at the place where the weapon was found
beg_hide	Begins the search for a place to hide the body at the room where the owner was killed
search	Moves the robot to a room which has not yet been searched for a weapon
find_tool	Moves the robot into the room with the weapon when the robot is adjacent to it
get_tool	Makes the robot pick up the weapon when the robot finds it
stalk	Moves the robot into unsearched rooms looking for the owner
find_owner	Moves the robot into the room with the owner when the robot is adjacent to it
kill_owner	Kills the owner when the robot is in the same room
hide	Moves the robot into unsearched rooms looking for a place to hide the body
find_place	Moves the robot into a room where the body could be hidden when the robot is adjacent to it

hide_body	Hides the body when the robot has reached a suitable place
mergeplans	Combines the three partial plans into one solution
solution	Recognizes a solution

Note that the design of this system is motivated by an attempt to illustrate the features of Teraphim, and not by the desire to attain an efficient program.

This system differs from the Identifier system in several ways:

1) The knowledge sources are not like production rules and usually call several specialized procedures from their action sections.

2) The problem blackboard on which the partial solutions are posted is not structured. Instead, a special evaluation function is supplied, which evaluates the cost of each partial plan based on the number of rooms Robbie must traverse.

3) Specialized control knowledge sources are provided, which divide the search for a solution into stages, as described above.

Robbie's path through the house will have three legs: from his starting position to the weapon, from the weapon to the owner, and from the owner to the hiding place for the corpse. The first two legs depend on which weapon is chosen, but the last leg depends only on which hiding place is selected. Therefore, the four possible solutions to the problem can be generated by determining the two paths which find the weapons and combining them with the two possible trips from the owner to a hiding place. Since the two parts of each solution are independent of each other, the system can work on several parts of the problem at the same time.

The paths through the house are generated by a simple breadth-first search procedure, with only one room added to the search path each execution cycle. Obviously, it would be

more efficient to write a single knowledge source to determine the shortest path between two rooms in one cycle.

Control over what part of the solution the system should work on is provided by some specialized control knowledge sources that direct the operation of the *prefer_focus* scheduling heuristic. For example, the knowledge source *searchfocus* creates a focus on the searches for the two weapons. This causes the system to favor KSARs involved with the search. The knowledge source *end_search* removes the *search* focus when the search has succeeded. It also *poisons* the search for the weapon that has been found, causing all KSARs pertaining to the search to be removed from the agenda. When the weapon has been found, the knowledge source changes the focus to prefer KSARs that belong to the search for the owner. The construction of the plan for hiding the body continues at the same time, under a different set of focuses. When plans for both doing the murder and hiding the body are available, the plans are merged to produce a plan to solve the entire problem.

When this problem is run, the first cycles are used for executing the control knowledge sources. For example:

```
Cycle 3
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(searchfocus,1,1)   100     some
ksar(searchfocus,1,2)   100     good
ksar(hidefocus,1,3)     100     good
ksar(hidefocus,1,4)     100     total
ksar(beg_search,1,5)    5       some
ksar(beg_search,1,6)    5       good
ksar(beg_stalk,1,7)     5       some
ksar(beg_stalk,1,8)     5       good
ksar(beg_hide,1,9)      5       good
ksar(beg_hide,1,10)     5       total
ksar(default,1,12)      100     total
Choosing best ksar
Choosing ksar 4
Execute best ksar
focus(scheme(frankenstein,cellar,_9530,_9531,_9532,hide,_9533))
```

```
Cycle 4
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(searchfocus,1,1)   100     some
ksar(searchfocus,1,2)   100     good
ksar(hidefocus,1,3)     100     good
ksar(beg_search,1,5)    5       some
ksar(beg_search,1,6)    5       good
ksar(beg_stalk,1,7)     5       some
ksar(beg_stalk,1,8)     5       good
ksar(beg_hide,1,9)      5       good
ksar(beg_hide,1,10)     45      total
ksar(default,1,12)      100     total
Choosing best ksar
Choosing ksar 12
Execute best ksar
active(prefer_recent)
active(prefer_easy)
```

Notice how the establishment of a focus affects the ratings of KSARs. The rating of KSAR 10 jumps from 5 to 45 when the focus is activated in cycle 3.

As the program continues, the *search* and *hide* subproblems generate KSARs. Eventually, the program finds a path from the owner's location to the cellar, which is the most reliable place to hide the body:

Cycle 14
Create ksars
Rating ksars

ksar	rating	certainty
ksar(beg_search,1,5)	45	some
ksar(beg_search,1,6)	45	good
ksar(beg_stalk,1,7)	5	some
ksar(beg_stalk,1,8)	5	good
ksar(beg_hide,1,9)	45	good
ksar(hide,10,18)	29	total
ksar(hide,10,19)	29	total
ksar(hide,11,20)	29	total
ksar(hide,12,21)	29	total
ksar(hide,13,22)	29	total
ksar(hide,13,23)	29	total
ksar(hide,13,24)	29	total
ksar(hide_body,14,26)	50	total

Choosing best ksar
Choosing ksar 26
Execute best ksar
Know how to get body to cellar
scheme(frankenstein,cellar,cellar,[body,family_room,cellar],
2,hide,yes)

Cycle 15
Create ksars
Rating ksars

ksar	rating	certainty
ksar(beg_search,1,5)	45	some
ksar(beg_search,1,6)	45	good
ksar(beg_stalk,1,7)	5	some
ksar(beg_stalk,1,8)	5	good
ksar(beg_hide,1,9)	45	good
ksar(hide,10,18)	29	total
ksar(hide,10,19)	29	total
ksar(hide,11,20)	29	total
ksar(hide,12,21)	29	total
ksar(hide,13,22)	29	total
ksar(hide,13,23)	29	total
ksar(hide,13,24)	29	total
ksar(end_hide,15,27)	100	total

Choosing best ksar
Choosing ksar 27
Execute best ksar
poison(_38305,scheme(frankenstein,cellar,_38306,
_38307,_38308,hide,no))
expired([hide,frankenstein,cellar])

```
Cycle 16
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(beg_search,1,5)    45      some
ksar(beg_search,1,6)    45      good
ksar(beg_stalk,1,7)     5       some
ksar(beg_stalk,1,8)     5       good
ksar(beg_hide,1,9)      45      good
Choosing best ksar
Choosing ksar 9
Execute best ksar
scheme(frankenstein,lawn,entry,[body],0,hide,no)
```

Notice that the solution to the subproblem involving the search for a path to the cellar causes the other KSARs related to this subproblem to be poisoned and removed from the agenda. This allows the program to run faster, since the reliabilities of the poisoned KSARs do not need to be recomputed.

On cycle 18, the program finds the way to hide the body in the lawn (easier but not as certain as hiding the body in the cellar). The gun is found in the den on cycle 26, and on cycle 32 a path from the den to the owner's location in the family room is determined. Now a complete solution to the problem can be assembled:

```
Cycle 36
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(beg_stalk,1,8)     5       good
ksar(search,22,39)      30      good
ksar(search,22,40)      30      good
ksar(search,22,41)      30      good
ksar(mergeplans,33,61)  43      some
ksar(mergeplans,33,62)  58      some
Choosing best ksar
Choosing ksar 62
Execute best ksar
scheme(frankenstein,gun,lawn,[family_room,den,gun,
family_room,entry, kill,body,lawn,hide],4,kill,yes)
```

```
Cycle 37
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(beg_stalk,1,8)     5       good
ksar(search,22,39)      30      good
ksar(search,22,40)      30      good
ksar(search,22,41)      30      good
ksar(mergeplans,33,61)  43      some
ksar(end_merge,37,63)   100     some
ksar(solution,37,64)    150     some
Choosing best ksar
Choosing ksar 64
Execute best ksar
Owner killed with gun
disposed of owner in lawn with gun after
family_room den gun family_room entry kill
body lawn hide costing 4
solved(frankenstein)
```

However, the gun has only *some* reliability as a murder weapon, so this solution is not definitive. The program continues to explore other possibilities. On cycle 40, Murder_plot assembles a plan to kill the owner with the gun and hide the body in the cellar, but this solution is no better than the previous one. On cycle 45, the program finds a path to the more reliable knife in the kitchen, and on cycle 52 it finds a path from the kitchen to the owner. Now the previously discovered methods of disposing of the body can be combined with this new plan for attacking the owner, and the optimum solution (the shortest path through the house which reliably kills and disposes of the owner) is determined on cycle 57. This involves using the knife as a weapon and hiding the body in the lawn:

```
Cycle 57
Create ksars
Rating ksars
ksar                    rating  certainty
ksar(mergeplans,53,86) 15       good
ksar(end_merge,57,88) 100       good
ksar(solution,57,89)    150     good
Choosing best ksar
Choosing ksar 89
Execute best ksar
Owner killed with knife
disposed of owner in lawn with knife after
family_room entry dining_room kitchen knife
dining_room entry kill body lawn hide
costing 6
solved(frankenstein)

All problems solved!
```

The program accepts this solution, which has *good* reliability, and stops. A different solution, using the knife on the owner and hiding the body in the cellar, has the same reliability but involves more steps, and so is not considered.

A program with this degree of complexity takes a long time to run: in *verbose* mode this solution was found after 497 seconds of CPU time. In *terse* mode, it required 420 seconds of CPU time. This is not unusual for complex programs written in C-Prolog. A more specialized program dedicated to solving this problem would have been much faster, however. Teraphim is useful primarily as a rapid prototyping tool for exploring poorly understood problems for which no simple solution is known.

5. CONCLUSIONS

5.1. Prolog as a Implementation Language

Prolog is naturally suited for pattern-directed programming, and is therefore a good choice for writing blackboard-based systems like Teraphim. Two main operations are required by the blackboard architecture: storing new blackboard entries, and matching blackboard entries with the triggering patterns of knowledge sources. Both are easily accomplished in Prolog. Prolog is backward-chaining whereas blackboard-based systems are essentially forward chaining, but this difference did not cause any difficulty.

Not all of Prolog's features were equally convenient. The primitive nature of Prolog's facilities for input and output of text was an obstacle. Also, since all terms in the Prolog database are global, the restrictions on access to the various blackboards that is imposed by Teraphim cannot be enforced, but must remain the responsibility of the programmer. Applications programmers would also need to be careful not to redefine any of the system's special terms.

Teraphim runs slowly. A different implementation of Prolog might have resulted in somewhat better performance, but many features of Teraphim itself tend to make it an expensive program to run, and a different serial language probably could not help it much.

5.2. Generic Control Knowledge

Because Teraphim is a domain-independent program, it cannot contain any special knowledge about the problems it will be used to solve. An expert system written with Teraphim cannot even learn much about itself by self-examination before starting to work on a solution. This puts limits on the amount of control knowledge the system can bring to the solution of any particular problem. Teraphim has three scheduling heuristics that can apply knowledge about a domain problem supplied by the application programmer: *prefer_deeper* can rate KSARs by their effect on the blackboard if the domain programmer provides a

structure for the blackboard, *use_evaluation* can rate KSARs according to a domain-specific evaluation function (which must be named *evaluate),* and *use_focus* responds to focuses placed by domain-specific control knowledge sources. Otherwise, only syntactic, not semantic, information is available to the system. For example, the scheduling heuristic *prefer_easy* increases the rating of KSARs produced by shorter knowledge sources since they will be easier to execute. Although Teraphim's generic control knowledge is adequate to guide the execution of simple programs not far removed from production systems, problems requiring complex solution strategies require domain-specific control knowledge that only the domain programmer can supply. It is difficult to see how this problem can be avoided.

5.3. Dealing With Uncertainty

The need to deal with uncertain information was the most important factor in Teraphim's long execution time. Because Teraphim is non-monotonic, the certainties of already established "facts" are subject to revision at any time. A list of the functors of all of the ancestors of each blackboard entry was kept, and the certainty of a term was checked whenever a new term with the same functor as one of the ancestors was added. For a program like Murder_plot, in which the majority of the blackboard entries have the same functor, this involved a great amount of computation. The certainty of a blackboard entry can only be found by recursive reference to the certainties of the term's antecedents. This need for recursion means that the standard allocations for global stack and heap storage may be exceeded by complex programs like the Murder_plot example.

5.4. Using Teraphim

As a blackboard-based system, Teraphim has features that make it suitable for writing prototype expert systems to solve problems in poorly understood domains. The system is robust, and it is usually possible to solve a problem in several different ways, although not all of the solutions are likely to be equally efficient. The independence of the knowledge

sources permits programs to be built incrementally and also encourages experimentation with different kinds of domain knowledge.

The Identifier program demonstrates the ease with which sets of rules can be transcribed into Teraphim programs. Far more complex problems can also be solved. The ability of knowledge sources to call procedures from their action sections permits the construction of arbitrarily powerful knowledge sources. The control knowledge sources of Teraphim could thus be used to manage the execution of any set of Prolog programs. The domain knowledge sources of a Teraphim program could include other artificial intelligence tools, such as a version of GPS or STRIPS.

It seems that complex control schemes, such as that used in the Murder_plot program, are very expensive in execution time. In general, Teraphim knowledge sources should probably be written to encode larger "chunks" of knowledge than did the knowledge sources in that program.

5.5. Parallel Execution

Blackboard architectures lend themselves to parallel processing. For example, in a massively parallel system, each knowledge source could be assigned its own processor. Then the processors could simultaneously determine which knowledge sources were triggered. While the scheduler was determining which KSAR to fire, all of the possible firing actions and productions could be precalculated. It seems that parallel processing of a blackboard-based system could speed up execution in direct proportion to the number of processors available, as long as there were more knowledge sources than processors - a very likely situation, since any truly useful program might have hundreds to thousands of knowledge sources.

LITERATURE CITED

BALZ80 Balzer, R., L.D. Erman, P. London, and C. Williams. 1980. HEARSAY-III: A domain-independent framework for expert systems. In *AAAI* 1, pp. 108-110

BARR81 Barr, A., and E.A. Feigenbaum, eds. 1981. *The Handbook of Artificial Intelligence* volume 1. Los Altos, California: William Kaufman

ERMA81 Erman, L.D., P.E. London, and S.F. Fickas. 1981. The design and an example use of HEARSAY-III. In *IJCAI* 7, pp. 409-415

HAYE83a Hayes-Roth, B. 1983a. The blackboard architecture: A general framework for problem solving? Technical Report HPP-83-30, Stanford, California, Stanford University

HAYE83b Hayes-Roth, B. 1983b. A blackboard model of control. Technical Report HPP-83-38, Stanford, California, Stanford University

HAYE84 Hayes-Roth, B. 1984. BB1: An architecture for blackboard systems that control, explain, and learn about their own behavior. Technical Report HPP-84-16, Stanford, California, Stanford University.

HAYE85a Hayes-Roth, B. 1985. A blackboard architecture for control. *Artificial Intelligence* 26, pp 251-321

HAYE85b Hayes-Roth, B., and M. Hewett. 1985. Learning control heuristics in BB1. Technical Report HPP-85-2, Stanford, California, Stanford University

LECO86 Lecot, K., and D.S. Parker. 1986. Control over inexact reasoning. *AI Expert* Premier issue, pp 32-43

LESS77 Lesser, V.R. and L.D. Erman. 1977. A retrospective view of the HEARSAY-II architecture. *IJCAI* 5, pp 790-800

NII79 Nii, H.P., and N. Aiello. 1979. AGE(Attempt To Generalize): A knowledge-based program for building knowledge-based programs. *IJCAI* 6, pp 645-655

NII80 Nii, H.P. 1980. An introduction to knowledge engineering, blackboard model, and AGE. Technical Report HPP-80-29, Stanford, California, Stanford University

NII85 Nii, H.P. 1985. Research on blackboard architectures at the Heuristic Programming Project. Technical Report KSL-85-24, Stanford, California, Stanford University

WATE78 Waterman, D.A., and F. Hayes-Roth. 1978. An overview of pattern-directed inference systems. In Waterman, D.A., and F. Hayes-Roths, eds., *Pattern-Directed Inference Systems* New York: Academic Press pp 3-24

WATE83 Waterman, D.A., and F. Hayes-Roth. 1983. An investigation of tools for building expert systems. In Hayes-Roth, F., D.A. Waterman, and D.B. Lenat, eds., *Building Expert Systems* Reading, Massachusetts: Addison-Wesley

WINS84 Winston, P.H., 1984 *Artificial Intelligence* second edition Reading, Massachusetts: Addison-Wesley

APPENDIX A: MAIN LOOP

```
/* This is a condensed version of the main control loop
of the Teraphim program. Some of the tracing statements
used by the verbose and crawl modes have been omitted in
the interest of clarity. */

run :-                                          /* This is the command
                                                issued by the user */
        compile,                                /* Create proper
                                                supports terms for
                                                initial input */

        continue,!.


continue :-
        cycle(N),                               /* What cycle is this */
        write_sent(['Cycle',N]),                /* Write it out */
        try_out,                                /* Trigger the
                                                knowledge sources,
                                                creating KSARs */
        poison_ksars,                           /* Remove all poisoned
                                                KSARs from agenda */
        apply,                                  /* Apply scheduling
                                                heuristics to the
                                                agenda of KSARs */
        abolish(temp_cert,2),                   /* Clean up from
                                                previous work */
        find_best(P),                           /* Select KSAR
                                                with best rating
                                                to execute */
        schedule(P),                            /* Execute the
                                                chosen knowledge
                                                source */
        next_cycle,                             /* Count cycle */
        ((not exist_unsolved,                   /* If all done,
        terminate);                             then stop */
        next).                                  /* otherwise,
                                                continue */
```

```
next :-
        quit,                                   /* If quit, then stop
                                                   without solution */
        write_sent(['No more worthwhile ksars']),
        findall(P,(problem(P),not solved(P)),List),
        (List=[];
        (Nlist=['I could not solve',List],
        flatten(Nlist,Flat),
        write_sent(Flat))),
        dump,
        write_sent(['Teraphim ends']).

next :-
        continue.                               /* Keep looping */
```

APPENDIX B: IDENTIFIER

```
knowledge_source(mammal_1) :->
        [[skin(Animal,hair)],
        [],
        [],
        [class(Animal,mammal)]].
reason(mammal_1,['Every animal with hairy skin is a mammal']).

knowledge_source(mammal_2) :->
        [[milk(Animal,yes)],
        [],
        [],
        [class(Animal,mammal)]].
reason(mammal_2,['Every animal that gives milk is a mammal']).

knowledge_source(bird_1) :->
        [[skin(Animal,feathers)],
        [],
        [],
        [class(Animal,bird)]].
reason(bird_1,['Every animal with feathers is a bird']).

knowledge_source(bird_2) :->
        [[locomotion(Animal,flies),
        eggs(Animal,yes)],
        [],
        [],
        [class(Animal,bird)]].
reason(bird_2,['Any animal that flies and lays eggs is a bird']).

knowledge_source(carniv_1) :->
        [[class(Animal,mammal),
        diet(Animal,meat)],
        [],
        [],
        [order(Animal,carnivore)]].
reason(carniv_1,['Any mammal that eats meat is a carnivore']).
```

```
knowledge_source(carniv_2) :->
        [[class(Animal,mammal),
        teeth(Animal,pointed),
        feet(Animal,claws),
        eyes(Animal,forward)],
        [],
        [],
        [order(Animal,carnivore)]].
reason(carniv_2,['Any mammal with pointed teeth, claws, and
forward-facing eyes is a carnivore']).

knowledge_source(ungulate_1) :->
        [[class(Animal,mammal),
        feet(Animal,hoofs)],
        [],
        [],
        [order(Animal,ungulate)]].
reason(ungulate_1,['Any mammal with hoofs is an ungulate']).

knowledge_source(ungulate_2) :->
        [[class(Animal,mammal),
        cud(Animal,yes)],
        [],
        [],
        [order(Animal,ungulate),
        toes(Animal,even_toes)]].
reason(ungulate_2,['Any mammal that chews a cud is an even-toed
ungulate']).

knowledge_source(cheetah) :->
        [[order(Animal,carnivore),
        color(Animal,tawny),
        pattern(Animal,spotted)],
        [],
        [],
        [species(Animal,cheetah)]].
reason(cheetah,['A carnivore with a tawny coat with spots is a
cheetah']).

knowledge_source(tiger) :->
        [[order(Animal,carnivore),
        color(Animal,tawny),
        pattern(Animal,striped)],
        [],
        [],
        [species(Animal,tiger)]].
reason(tiger,['A carnivore with a tawny coat with stripes is a tiger']).
```

```
knowledge_source(giraffe) :->
        [[order(Animal,ungulate),
        legs(Animal,long),
        neck(Animal,long),
        color(Animal,tawny),
        pattern(Animal,spotted)],
        [],
        [],
        [species(Animal,giraffe)]].
reason(giraffe,['A tawny ungulate with long legs and neck and spots
is a giraffe']).


knowledge_source(zebra) :->
        [[order(Animal,ungulate),
        color(Animal,black_and_white),
        pattern(Animal,striped)],
        [],
        [],
        [species(Animal,zebra)]].
reason(zebra,['A black_and_white striped ungulate is a zebra']).


knowledge_source(ostrich) :->
        [[class(Animal,bird),
        non(locomotion(Animal,flies)),
        neck(Animal,long),
        color(Animal,black_and_white)],
        [],
        [],
        [species(Animal,ostrich)]].
reason(ostrich,['A non-flying black_and_white bird with a long neck
is an ostrich']).


knowledge_source(penguin) :->
        [[class(Animal,bird),
        non(locomotion(Animal,flies)),
        locomotion(Animal,swims),
        color(Animal,black_and_white)],
        [],
        [],
        [species(Animal,penguin)]].
reason(penguin,['A non-flying black_and_white bird that swims is
a penguin']).
```

```
knowledge_source(default_bird) :->
        [[class(Animal,bird),
        locomotion(Animal,flies)],
        [],
        [],
        [species(Animal,albatross)]].
reason(default_bird,['The only flying bird I know about is an
albatross']).


knowledge_source(solution) :->
        [[species(Animal,What)],
        [],
        [write_sent(['I have determined that',Animal,'is a',What])],
        [solved(Animal)]].
type(solution,plan).
importance(solution,150).
```

APPENDIX C: MURDER _PLOT

```
knowledge_source(searchfocus) :->
        [[problem(P),
        lethal(Weapon)],
        [],
        [],
        [focus(scheme(P,Weapon,_,_,_,search,_))]].
importance(searchfocus,100).
type(searchfocus,plan).
reason(searchfocus,['Work on finding weapon']).


knowledge_source(end_search) :->
        [[scheme(P,Weapon,_,_,_,search,yes)],
        [],
        [delete(focus(scheme(P,Weapon,_,_,_,search,_)))],
        [poison(KS,scheme(P,Weapon,_,_,_,search,no)),
        expired([search,P,Weapon])]].
importance(end_search,100).
type(end_search,plan).
reason(end_search,['Stop searching when weapon found']).


knowledge_source(stalkfocus) :->
        [[expired([search,P,Weapon])],
        [],
        [],
        [focus(scheme(P,Weapon,_,_,_,stalk,_))]].
importance(stalkfocus,100).
type(stalkfocus,plan).
reason(stalkfocus,['Work on finding owner']).


knowledge_source(end_stalk) :->
        [[scheme(P,Weapon,_,_,_,stalk,yes)],
        [],
        [delete(focus(scheme(P,Weapon,_,_,_,stalk,_)))],
        [poison(KS,scheme(P,Weapon,_,_,_,stalk,no)),
        expired([stalk,P,Weapon])]].
importance(end_stalk,100).
type(end_stalk,plan).
reason(end_stalk,['Stop looking for owner when you find him']).
```

```
knowledge_source(hidefocus) :->
        [[hide_place(Goal),
        problem(P)],
        [],
        [],
        [focus(scheme(P,Goal,_,_,_,hide,_))]].
importance(hidefocus,100).
type(hidefocus,plan).
reason(hidefocus,['Work on finding a place to hide the body']).


knowledge_source(end_hide) :->
        [[scheme(P,Goal,_,_,_,hide,yes)],
        [],
        [delete(focus(scheme(P,Goal,_,_,_,hide,_)))],
        [poison(KS,scheme(P,Goal,_,_,_,hide,no)),
        expired([hide,P,Goal])]].
importance(end_hide,100).
type(end_hide,plan).
reason(end_hide,['Stop looking for hiding place once found']).


knowledge_source(mergefocus) :->
        [[scheme(P,Goal,_,_,_,hide,yes),
        scheme(P,Weapon,_,_,_,stalk,yes)],
        [],
        [],
        [focus(scheme(P,Weapon,Goal,_,_,kill,_))]].
importance(mergefocus,100).
type(mergefocus,plan).
reason(mergefocus,['Combine the plans if you can']).


knowledge_source(end_merge) :->
        [[scheme(P,Weapon,Goal,_,_,kill,yes)],
        [],
        [delete(focus(scheme(P,Weapon,Goal,_,_,kill,_)))],
        [poison(KS,scheme(P,Weapon,Goal,_,_,kill,_)),
        expired([merge,P,Weapon,Goal])]].
importance(end_merge,100).
type(end_merge,plan).
reason(end_merge,['If you"ve merged plans, stop working on them']).


knowledge_source(beg_search) :->
        [[location(Problem,robot,Room),
        lethal(Weapon)],
        [],
        [],
        [scheme(Problem,Weapon,Room,[Room],0,search,no)]].
importance(beg_search,5).
reason(beg_search,['Start search for weapon']).
```

```
knowledge_source(beg_stalk) :->
        [[lethal(Weapon),
        location(Problem,Weapon,Room)],
        [],
        [],
        [scheme(Problem,Weapon,Room,[Weapon],0,stalk,no)]].
importance(beg_stalk,5).
reason(beg_stalk,['Start search for owner']).


knowledge_source(beg_hide) :->
        [[location(Problem,owner,Room),
        hide_place(Goal)],
        [],
        [],
        [scheme(Problem,Goal,Room,[body],0,hide,no)]].
importance(beg_hide,5).
reason(beg_hide,['Start search for place to hide body']).


knowledge_source(search) :->
        [[scheme(Problem,Weapon,Room,Plan,Cost,search,no),
        non(location(Problem,Weapon,Room)),
        adjacent(Room,NewRoom,MoveCost),
        allowable(NewRoom,Plan)],
        [],
        [NewCost is Cost + MoveCost,
        add_to(Plan,NewRoom,NewPlan)],
        [scheme(Problem,Weapon,NewRoom,NewPlan,NewCost,search,no)]].
importance(search,5).
reason(search,['Haven"t looked here yet for weapon'].


knowledge_source(find_tool) :->
        [[scheme(Problem,Weapon,Room,Plan,Cost,search,no),
        adjacent(Room,NewRoom,MoveCost),
        location(Problem,Weapon,NewRoom),
        allowable(NewRoom,Plan)],
        [],
        [NewCost is Cost + MoveCost,
        add_to(Plan,NewRoom,NewPlan)],
        [scheme(Problem,Weapon,NewRoom,NewPlan,NewCost,search,no)]].
importance(find_tool,20).
reason(find_tool,['Weapon is in this room']).
```

```
knowledge_source(get_tool) :->
        [[scheme(Problem,Weapon,Room,Plan,Cost,search,no),
        location(Problem,Weapon,Room)],
        [],
        [write_sent(['Know how to find',Weapon,'in',Room])],
        [scheme(Problem,Weapon,Room,Plan,Cost,search,yes)]].
importance(get_tool,40).
reason(get_tool,['Take weapon if in room with it']).


knowledge_source(stalk) :->
        [[scheme(Problem,Weapon,Room,Plan,Cost,stalk,no),
        non(location(Problem,owner,Room)),
        adjacent(Room,NewRoom,MoveCost),
        allowable(NewRoom,Plan)],
        [],
        [NewCost is Cost + MoveCost,
        add_to(Plan,NewRoom,NewPlan)],
        [scheme(Problem,Weapon,NewRoom,NewPlan,NewCost,stalk,no)]].
importance(stalk,5).
reason(stalk,['Look for owner in unsearched rooms']).


knowledge_source(find_owner) :->
        [[scheme(Problem,Weapon,Room,Plan,Cost,stalk,no),
        adjacent(Room,NewRoom,MoveCost),
        location(Problem,owner,NewRoom),
        allowable(NewRoom,Plan)],
        [],
        [NewCost is Cost + MoveCost,
        add_to(Plan,NewRoom,NewPlan)],
        [scheme(Problem,Weapon,NewRoom,NewPlan,NewCost,stalk,no)]].
importance(find_owner,20).
reason(find_owner,['Go into room with owner in it to find him']).


knowledge_source(kill_owner) :->
        [[scheme(Problem,Weapon,Room,Plan,Cost,stalk,no),
        location(Problem,owner,Room)],
        [],
        [write_sent(['Know how to find owner with',Weapon])],
        [scheme(Problem,Weapon,Room,Plan,Cost,stalk,yes)]].
importance(kill_owner,40).
reason(kill_owner,['Kill owner if in room with him']).
```

```
knowledge_source(hide) :->
        [[scheme(Problem,Goal,Room,Plan,Cost,hide,no),
        adjacent(Room,NewRoom,MoveCost),
        allowable(NewRoom,Plan)],
        [],
        [NewCost is Cost + MoveCost,
        add_to(Plan,NewRoom,NewPlan)],
        [scheme(Problem,Goal,NewRoom,NewPlan,NewCost,hide,no)]].
importance(hide,5).
reason(hide,['Look in new room for place to hide body']).


knowledge_source(find_place) :->
        [[scheme(Problem,Goal,Room,Plan,Cost,hide,no),
        adjacent(Room,Goal,MoveCost),
        allowable(Goal,Plan)],
        [],
        [NewCost is Cost + MoveCost,
        add_to(Plan,Goal,NewPlan)],
        [scheme(Problem,Goal,Goal,NewPlan,NewCost,hide,no)]].
importance(find_place,20).
reason(find_place,['Go in room if it"s a good place to hide body']).


knowledge_source(hide_body) :->
        [[scheme(Problem,Room,Room,Plan,Cost,hide,no)],
        [],
        [write_sent(['Know how to get body to',Room])],
        [scheme(Problem,Room,Room,Plan,Cost,hide,yes)]].
importance(hide_body,40).
reason(hide_body,['Hide body if room is suitable']).


knowledge_source(mergeplans) :->
        [[scheme(Problem,Weapon,Room,Plan,Cost,search,yes),
        scheme(Problem,Weapon,Room2,Plan2,Cost2,stalk,yes),
        scheme(Problem,Room3,Room3,Plan3,Cost3,hide,yes)],
        [],
        [NewCost is Cost + Cost2 + Cost3,
        append(Plan,Plan2,Plan4),
        add_to(Plan4,kill,Plan5),
        add_to(Plan3,hide,Plan3a),
        append(Plan5,Plan3a,Plan6)],
        [scheme(Problem,Weapon,Room3,Plan6,NewCost,kill,yes)]].
importance(mergeplans,80).
reason(mergeplans,['Combine partial plans to solve problem']).
```

```
knowledge_source(solution) :->
        [[scheme(Problem,Weapon,Room,Plan,Cost,kill,yes)],
        [],
        [translate(scheme(Problem,Weapon,Room,Plan,Cost,kill,yes),List),
        write_sent(['Owner killed with',Weapon]),
        write_sent(List)],
        [solved(Problem)]].
importance(solution,150).
type(solution,plan).
reason(solution,['Have figured out a plan to get rid of owner']).
```