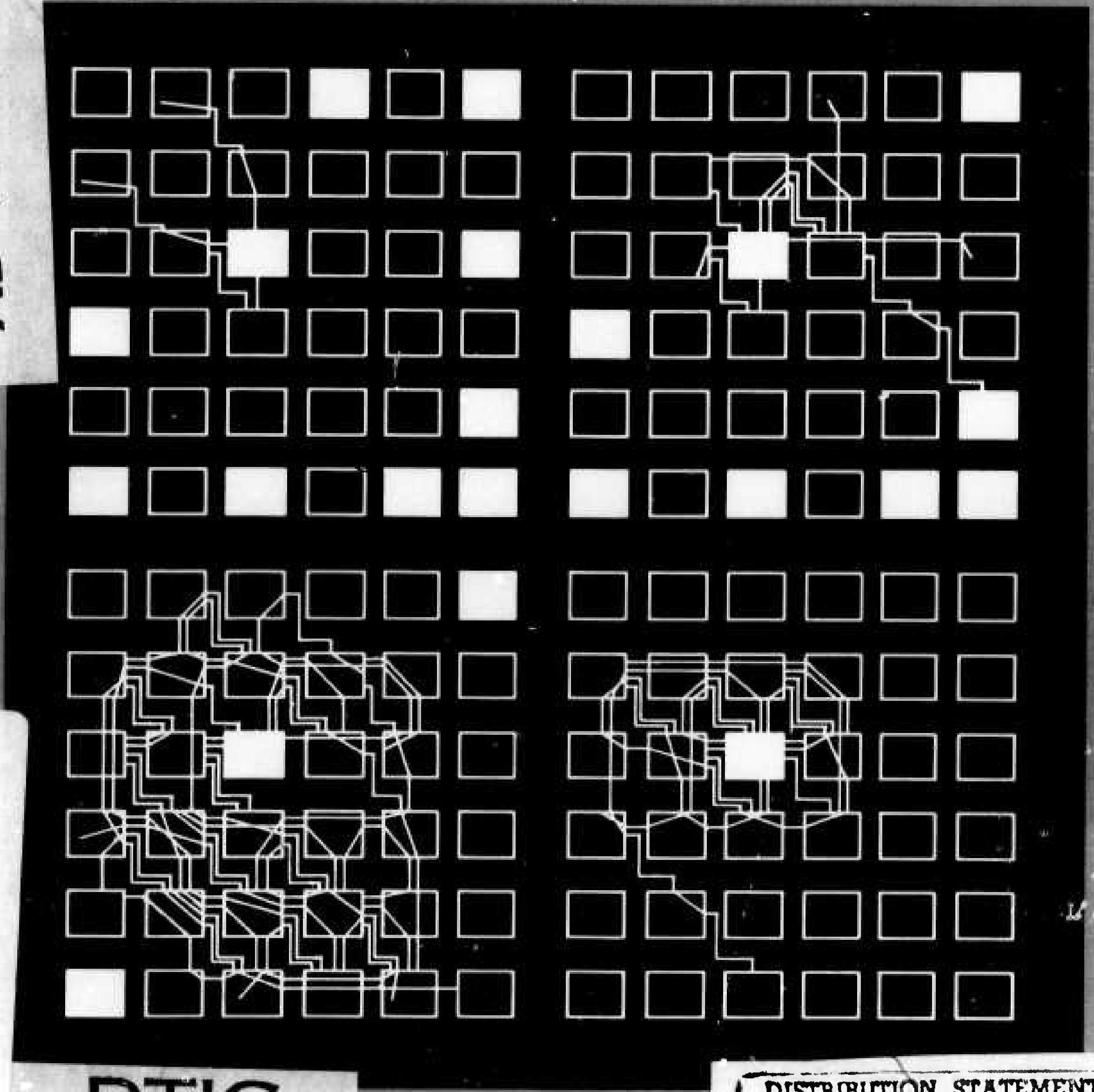
12

PROCEEDINGS

EXPERT SYSTEMS WORKSHOP

April 1986



Sponsored by:

DARPA

Defense Advanced Research Projects Agency
Information Processing Techniques Office

EXPERT SYSTEMS WORKSHOP

Proceedings of a Workshop
Held at
Asilomar Conference Center
Pacific Grove, California
April 16-18, 1986

Sponsored by the
Defense Advanced Research Projects Agency

Science Applications International Corporation
Report Number SAIC-86/1701
Lee S. Baumann
Workshop Organizer

This report was supported by
The Defense Advanced Research
Projects Agency under DARPA
Order No. 3456, Contract No. MDA903-84-C-0160
Monitored by the
Defense Supply Service, Washington, D.C.

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER
SAIC-86/1701

2. GOVT ACCESSION NO.
AD-A170 399

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
PROCEEDINGS
EXPERT SYSTEMS WORKSHOP
~~Proceedings of a workshop,~~ April 1986

5. TYPE OF REPORT & PERIOD COVERED
ANNUAL TECHNICAL
October 1985-April 1986

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
LEE S. BAUMANN (Ed.)

8. CONTRACT OR GRANT NUMBER(s)
MDA903-84-C-0160

9. PERFORMING ORGANIZATION NAME AND ADDRESS
SCIENCE APPLICATIONS INTERNATIONAL CORPORATION
1710 Goodridge Drive, 10th Floor
McLean, Virginia 22102

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
ARPA ORDER No. 3456

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

12. REPORT DATE
April 1986

13. NUMBER OF PAGES
197

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Expert Systems, Artificial Intelligence, Knowledge Engineering, Experimental Knowledge Systems, System Building Tools, Reasoning with incomplete information, Reasoning with uncertain information, Knowledge Acquisition, Problem solving frameworks.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document contains the technical papers for the Expert Systems Program which were presented by the key research specialists from the research activities participating in this program sponsored by the Information Processing Techniques Office, Defense Advanced Research Projects Agency. The reviews of these papers were presented at a workshop conducted on 16-18 April 1986, at Asilomar Conference Center, California.

TABLE OF CONTENTS

Page

TECHNICAL PAPERS

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

Page

TECHNICAL PAPERS (Cont'd)

Use the title on the front cover.
Per Mr. Lee S. Baumann, Science Applications
International Corp.

FOREWORD

A workshop for research personnel involved in the DARPA program on Expert Systems was held in Palo Alto and at the Asilomar Conference Center, Monterey, California, from 16-18 April 1986. The purpose of the workshop was to demonstrate working systems tools and to review progress on the technical aspects of the research being undertaken. Research organizations participating in the workshop included the University of Massachusetts at Amherst; Ohio State University; Stanford University; the Information Sciences Institute of the University of Southern California; Bolt, Beranek and Newman Laboratories, Inc.; General Electric Corporation; Teknowledge, Inc.; and the IntelliCorp Company. Representing the Department of Defense in addition to the DARPA program manager were experts from the Rome Air Development Center, the Air Force Wright Aeronautical Laboratories, the Space and Naval Warfare Systems Command, and the Naval Underwater Systems Command. Also attending was a representative from Texas Instruments, the integration contractor for the Navy Battle Management Program.

This proceeding is intended to document important progress being made in the knowledge-based systems part of the DARPA Strategic Computing Program. The papers included give a good insight into the current accomplishments. Included in this foreword is a short documentation of the demonstrations that were presented but are not further described in the proceedings.

The workshop met on Wednesday, 16 April 1986, in the new offices of Teknowledge, Inc., in Palo Alto, California. The first morning consisted of a series of live demonstrations given by IntelliCorp, Ohio State University and Teknowledge. CDR Allen Sears, the DARPA program manager, welcomed the forty attendees to the demonstration and thanked the Teknowledge people for their assistance in setting up the demonstrations, providing the necessary equipment, and their hospitality in providing conference space to view the programs. Ohio State University provided the lead off demonstration. Dr. Chandrasekaran, the principal investigator, explained that the program was a prototype mission planning associate in the domain of an offensive counter air planning task. This is, he explained, a generic tool using DSPL representation. DSPL is a language developed at Ohio State which uses knowledge representation rich in planning primitives. Dave Herman, Dean Allemang and Anne Keuneke of Ohio State explained the workings of the system as the demonstration progressed. The program accepted the plan inputs and by use of design plans selected the aircraft type and ordnance configuration most appropriate for the mission factors under consideration. In making its selection the program uses a functional representation of the plan and the capture of the agents understanding of how things work. This includes as a piece of knowledge the order that things are considered in the planning cycle. The audience was able to see on the screen the progression of the logic flow as the events progressed.

Dr. Rick Hayes-Roth explained the genesis of the Teknowledge research effort as the creation of a foundation on which to build systems with reusable knowledge processing modules and skeletal systems, modularity and standard interfaces, encapsulation and cooperative systems, integration of technologies, and the ability to take partial solutions off the shelf and put them together into new systems thus providing customized solutions to new problems. Hayes-Roth stated that the program is a twenty four month effort of which they were now eleven months into the research. The Teknowledge system, called ABE, was able to integrate new modules into its tools catalogue and to provide a capability to use whichever tools best suited the problem domain. The system architects' catalog contains applications, customizations, skeletal systems, capabilities, abstract data types, frameworks, and languages in a descending order of layered structures. As goals, the ABE project deems it important to import technologies, layer systems, and glue them together in a robust and disciplined way. The demonstration covered six items: the system architects catalog, a first example, composing frameworks, importing a capability, variations, and composing with heterogeneous frameworks. Assisting in the demonstration were Lee Erman, Jay Lark, Terry Barnes, Kamal Bijlani, Michael Fehling, Bruce Bullock, and Neil Jacobstein.

The IntelliCorp demonstration was presented by Richard Fikes. He explained that the essence of their program was to take pieces of A.I. technology and integrate them for use in systems. The outcome is to develop tools which may be used by others. KEE, the central IntelliCorp product, has been in use for several years, Fikes noted, and the protocols for access and use of the system have remained standard. The recent effort is to develop new tools, such as distributed knowledge bases, and to fit these new tools into KEE for use by applications developers. The DARPA program has now been on-going for one year and a new initial set has been produced called DARPA-KEE. They have built interfaces to an assumption based truth maintenance module and to world based problem solving routines. The demonstration was designed to include model based reasoning, symbolic description and reasoning about descriptions. The domain selected involved knowledge based tools to aid the dispatcher of a trucking delivery system over a mid-west geographical area. Involved were manual context exploration, a semi-automatic task completion rule system, and programmatic automatic problem solving routines. The progression of the task was easily followed on the terminal screen as the problem moved from initiation to suggested solutions and as new parameters were added or changed.

CDR Sears remarked that the demonstration proved that a lot has happened in the year since the program was initiated and that we are now looking at bringing technology to the applications developers. This, he noted, will require planning to insure successful implementation.

The remainder of the workshop was conducted at the Asilomar Conference Center. Each of the organizations attending presented one or more technical reviews of the status of the expert systems research being undertaken in the DARPA program. This proceeding contains copies of those reviews in order to provide a wide distribution of the program and results achieved to date. Following the technical talks, the participants discussed applications and transition strategy, future goals, and integration of expert system technology with other parts of the DARPA research program. The program concluded with a discussion of high level tools for expert systems led by Dr. Chandrasekaran of Ohio State University.

The cover layout for this proceedings was created by Tom Dickerson of the Graphics Department at SAIC using diagrams of a multicast-map from the paper: "CAREL: A Visable Distributed Lisp," by Byron Davies of the Knowledge Systems Laboratory at Stanford University and of the Texas Instruments Corporation. The diagrams are samples from the execution of the IDENTIFY-YOURSELF program which is described in Davies paper included herein. This proceedings has been provided to the Defense Technical Information Center (DTIC) and copies may be secured from that agency.

Lee S. Baumann
Science Applications International
Corporation
Workshop Organizer

AUTHOR INDEX

TECHNICAL PAPERS

The BBN Laboratories Knowledge Acquisition Project: KREME Knowledge Editing Environment

Glenn Abrett and Mark H. Burstein

BBN Laboratories
10 Moulton Street
Cambridge, MA 02238

Abstract

One of the major bottlenecks in large-scale expert system development is the problem of knowledge acquisition. the construction, maintenance, and testing of large knowledge bases. The BBN Laboratories Knowledge Acquisition Project is investigating ways of easing these problems and. where possihle, auloma ing the knowledge acquisition process This paper details the current state of development of the KREME Knowledge Representation Editing and Modeling Environment KREME is an extensible experimental environment for developing and editing knowledge bases using a variety of styles of representations It provides tools for effective viewing and browsing in each kind of representational base, automatic consistency checking, and *macro-editing* facilities to reduce the burdens of large scale *knowledge base revision and reformulation*. Our goal is to explore a number of approaches to knowledge acquisition and knowledge editing that could be incorporated into existing and future full-scale expert system development environments.[1]

1. Introduction

1.1. The Knowledge Acquisition Problem

There is substantial agreement within the AI community that the way to make expert systems more closely approximate the level of performance exhibited by people is to give the systems more knowledge. The creation of the large and detailed bodies of knowledge needed to substantially improve performance has proven to be excrutiatingly painful. Beyond a certain point, several factors make the building of very large knowledge bases a practical impossibility with current technology.

Knowledge comes in many forms.

Human knowledge about the world comes in many disparate forms Squeezing all the knowledge that an expert system needs into one, or at best two, representational formalisms (e.g rules and frames) is difficult, time consuming, often inappropriate and, in many cases, an inadequate solution to the task at hand.

Managing large knowledge bases is difficult.

As knowledge bases grow in size and complexity they strain the capacities of software tools for knowledge editing, maintenance, and validity checking. Viewpoints at the right level of detail are hard to construct, consistency checking takes up more and more time, and global reorganizations and modifications can no longer be done easily one piece at a time. Eventually, user confidence in the internal coherence of the knowledge base erodes and must be restored by the inefficient, incomplete, and indirect method of running applications programs using the knowledge base.

Previously encoded knowledge is not re-used.

It is customary to start building a new expert system with an empty knowledge base, even though the completed knowledge base will contain at least some general knowledge about the world. To make matters worse, this general world knowledge is usually entered in a fragmentary and sketchy manner that adds little to the power of the system If general knowledge about the world could be transferred across systems, the gradual accumulation of detail, precision, and richness which would occur would tremendously enhance the performance and robustness of most individual expert systems.

1.2. Overview of the BBN Knowledge Acquisition Project

Our goal has been to develop an environment in which the problems of knowledge acquisition faced by every knowledge engineer attempting to build a large expert system are minimized To this end, we have organized the task of developing knowledge acquisition tools into two stages. First, we are developing a well-integrated knowledge representation, editing and modeling environment. dubbed KREME. Knowledge engineers and subject matter experts with some knowledge of basic knowledge representation techniques will find it easy to use KREME to acquire, edit. and view from multiple perspectives knowledge bases that are several times larger than those found in most current systems KREMF provides, within a uniform environment, special purpose editing facilities that permit knowledge to he represented and viewed in a variety of formalisms appropriate to its use, rather than forcing all knowledge to be represented in a single, unitary formalism During phase two of the project, we will consider such automatic kinds of knowledge acquisition as developing representations from examples, and learning by analogy.

In addition to a general editing environment, the first phase has also focused on developing tools that provide the kinds of validation and consistency checking

[1] This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by RADC under contract number F30602-85-C-0005.

so essential during the development or modification of knowledge bases. As the size of knowledge bases grow, and more people become involved in their development, this aspect of knowledge acquisition becomes increasingly important. In the hybrid or multi-formalism representational systems that are becoming prevalent [11, 2, 19], techniques must be provided for consistency checking not only within a single representational system, but between related systems.

A third important area of investigation in developing the KREME editing environment has been the attempt to provide of facilities for large-scale revisions of portions of a knowledge base. Our experience indicates that the development of an expert system inevitably requires systematic, large scale revisions of portions of the developed representation. This is often caused by the addition or redefinition of a task the system is to perform. These kinds of systematic changes to a knowledge base have, to date, only been possible by painstaking piecemeal revision of each affected element, one at a time. Our initial approach has been to provide a *macro-editing* facility, in which the required editing operations can be demonstrated by example and applied to specified sets of knowledge structures automatically. We plan to provide a library of such generic macro-editing operations for the most common and conceptually simple (though potentially difficult to describe) operations during phase two of the project.

1.3. The KREME Knowledge Editor

KREME attempts to deal with the inextricably related problems of knowledge representation and knowledge acquisition in a unified manner by organizing multiple representation languages and multiple knowledge editors inside of a coherent global environment. A key design goal for KREME was to build an environment in which existing knowledge representation languages, appropriate to diverse types of knowledge, could be integrated and organized as components of a coherent global representation system. As it is presently conceived (and for the most part implemented) the KREME Knowledge Editor can be thought of as an extensible set of globally coherent operations that apply across a number of related knowledge representation editors, each tailored to a specific type of knowledge. Our approach has been to integrate several existing representation languages in an open ended architecture that allows the extension of each of these languages. In addition, we have provided for the incorporation of additional representation languages to handle additional types of knowledge.

To accomplish this goal, we envisioned a decomposition of existing knowledge representation techniques, to be implemented as objects or FLAVORS [6], in terms of which we could reimplement existing representation languages. Each object encoding an aspect of some representation would be responsible for its own display, editing and internal forms. By organizing this "meta-knowledge base" modularly, behavioral objects implementing inheritance behavior, subsumption testing, and coreference mechanisms, etc., could be "mixed in" to a number of representational subsystems.

The current implementation of KREME partially accomplishes our goal. We have organized a small library of component behavioral objects for knowledge representations and succeeded in reimplementing our frame language in terms of this object base. We expect this library to be an extremely useful set of building blocks as we attempt various extensions to the expressive power of our system.

The current version of KREME contains individual editors for three distinct representation languages, one for frames, one for rules and one for procedures. The frame and procedure editors are fully integrated into the global environment and the rule editor is in the process of becoming so. Eventually, the rule editor, the procedure editor and a functional method editor will all be accessible through a global mechanism that treats these types of knowledge as forms of procedural attachment to concepts. In phase two of the project, we plan to add a language for representing causal and other qualitative constraint systems, and several types of instantiation mechanisms, including a truth maintenance system for propositional representation.

1.4. The KREME Frame Language

Much of the work done in the current implementation of KREME has been focused on building a knowledge editor for a frame representation language. Such languages have been well researched, and while we had to have some frame language on which to base our initial editor, we did not want to design and implement a new one. Our most important criteria for a suitable frame representation language were that it:

1. Allowed multiple inheritance.
2. Was a logically worked out mature language.
3. Had some mechanism for internal consistency checking.
4. Would allow individuals to be instantiated as objects from the definitions of frames.
5. Was built on a modular object oriented base so that the language could be decomposed in such a way as to make it easily extensible.

NIKL (the definitional or frame language component of KL-TWO) [9, 14, 19] seemed an ideal candidate. It is a fully worked out frame representation language that allows multiple inheritance, is reasonably expressive and, perhaps most importantly, contains a fully worked out automatic classification algorithm that could be easily adapted to provide a powerful mechanism for consistency checking and enforcement during knowledge base development. However, no object-oriented implementation of NIKL existed, and the NIKL classifier was not designed to allow *modification* and *reclassification* of previously defined concepts. A second frame language, known as MSG, had been built as part of BBN's STEAMER project and was readily available. MSG is object oriented in both of the above senses but it has no classifier and is not as mature or thoroughly specified a language as NIKL.

To develop KREME, we elected to reimplement NIKL as an object oriented language using MSG as a guide. The NIKL data structures were decomposed into a modular hiererchy of flavor definitions, and the KREME version of NIKL was then built out of these flavors. This enabled us to incorporate a great deal of the fairly sophisticated instantiation mechanism of MSG with minimal effort. In the process, we were also able to re-implement the NIKL classifier algorithm to provide the kind of reclassification capability required for a knowledge editing environment. We will refer to this enhanced, object oriented implementation of NIKL as KREME Frames.

The remainder of this section will review the basic features of the KREME Frames language. As the definitional syntax of KREME Frames coincides almost exactly with the structure of the NIKL language, interested readers are referred to [9] for more detail. Section 2 will describe the KREME editing environment and the frame editor. Section 3 will discuss the classifier, and its use in an interactive editing environment.

1.5. Definition of KREME Frames

In KREME, a frame is called a *concept*. Collections of concepts are organized into a rooted *inheritance* or *subsumption lattice* sometimes referred to as a *taxonomy* of concepts. A single distinguished concept, usually called THING, serves as the root or *most general concept* of the lattice. Figure 1-1 shows a simple subsumption lattice.

A concept has a *name*, a textual *description*, a *primitiveness* flag, a list of *defined parents* (concepts that it *specializes* or is *subsumed by*), a list of *role restrictions*, a list of *role equivalences*, and a list of concepts that it is *disjoint from*[2]. In KREME, as in NIKL, a concept may be subsumed by more than just the concepts that are its defined parents. Thus, classified concepts in a KREME hierarchy also contain distinct lists of those concepts that directly subsume it, and those which it directly subsumes or are its direct children.

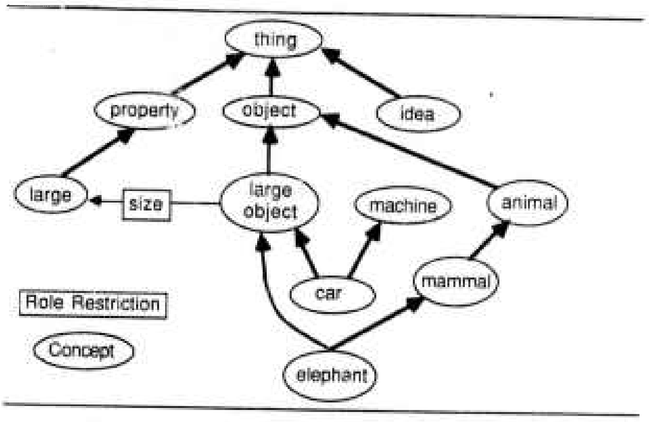


Figure 1-1: A Simple Concept Taxonomy

```
(defconcept HOUSE
   :primitive t
   :specializes (building)
   :role-restrictions
      ((residents (a person) nil (a person))
       (front-door (a door) (1 1) (a door)))
   :equivalences
      ((main-entrance) (front-door))
   :disjoint (office-building apartment-building))
```

Figure 1-2: LISP form of a KREME frame definition

The lists of role restrictions, role equivalences and disjoint concepts are collectively referred to as the *features* of a concept. If each concept can be thought of as defining a unique category, then features of the concept define the necessary conditions for inclusion in that category. If a concept is not marked as *primitive* (a case sometimes referred to as a *defined* concept) the features also constitute the complete set of sufficient conditions for inclusion in that category. A concept inherits all features from those concepts above it in the lattice (those concepts that subsume it, and, thus, are more general) and may define additional features that serve to distinguish it from its parent or parents.

Role restrictions define the necessary slot-value pairs for any instance to be considered a member of the class defined by a concept. A role restriction consists of a role name, a value restriction, a number restriction and an (optional) default form[3].

The role name refers to an object called a *role*. Roles in KREME, as in NIKL and some other frame languages like KEE [5], and KnowledgeCraft [7], are actually distinct, first class objects. Roles describe *relations* between concepts. A *role restriction* at a concept is thus a specification of the ways a given role can be used to relate that concept to other concepts. As first-class objects, roles form their own distinct taxonomy, rooted at the most general possible role, usually called RELATION. Figure 1-3 shows a portion of a simple role taxonomy.

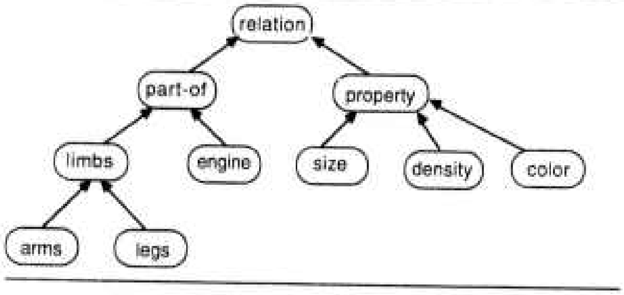


Figure 1-3: A Simple Role Taxonomy

A role has a *name*, a *description*, a list of roles that it *specializes*, a *domain* and a *range*. In a formal sense, a role is a two-place relation that maps instances of concepts in its domain onto sets of instances in its range. The domain of a role is the most general concept at which the role makes sense. That is, it specifies the class of things for which the role can name a slot. The range of a role specifies the general class of concepts that can serve as values in slots defined using that role. All concepts filling slots whose name is a given role must be elements of the range of that role.

Each role restriction at a concept has as part of its definition a *value restriction*, which is the class of allowed values for that slot. The value restriction must always be a sub-class of the range of that role, and a subclass of the value restrictions defined for that role at all concepts subsuming the one restricted. At present, following the structure of NIKL, value restrictions must be defined concepts. We expect to relax this constraing in the near future.

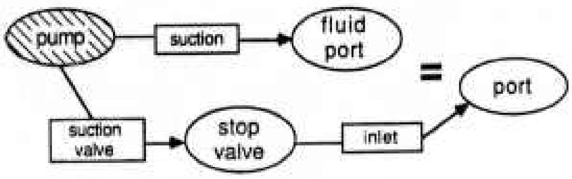
Role restrictions also include a *number restriction* that specifies the minimum and maximum (if any) number of things that may be related by the role to the concept at any given time. For example, if all elephants have four legs, then the concept ELEPHANT might be defined to restrict the role LEGS to *Exactly 4* ELEPHANT-LEGS[4]. A number restriction must be at least as specific as all the number restrictions for the same role at any of the concepts parents[5].

Role Equivalences describe slots (and slots of slots) that *by definition* refer to the same entities. They are defined as pairs of paths whose referents are the same concept. A path is a list of role names, the head of which is a role restricted at the concept defining the

[2] One concept is *disjoint from* another if being one precludes being the other.

[3] Defaults were not part of the definition of NIKL

equivalence. Each subsequent role (slot name) in a path must be a valid slot in the concept that is the value restriction of the previous role in the path. The referent of a path is the value restriction of the last role restriction in the chain. Figure 1-4 shows a simple example of role equivalence.



The SUCTION of the PUMP is equivalant to the INLET of the SUCTION VALVE of the PUMP.

Figure 1-4: A Role Equivalence

Concepts marked as *primitive* (sometimes referred to as *Natural Kinds*) have no complete set of sufficient conditions. For example, an ELEPHANT must, by necessity, be a MAMMAL, but without an exhaustive list of the attributes that distinguish it from other mammals, it must be represented as a primitive concept. WHITE ELEPHANT, on the other hand, might be completely described by stating that it is a specialization of ELEPHANT, where the role COLOR was restricted to WHITE.

KREME Frames permit slots to have default values as well as value restrictions. If present, the default must be the description of some concept which satisfies the restrictions on the role at that concept. The default is used as a slot filler for instances of a concept that do not specify a value for the slot at instantiation time. Defaults are inherited from the most specific parent at which they are defined, just as in most other frame languages, rather than by logical set intersection, as the classifier does for other KREME concept features. Specialization of defaults is not enforced. Figure 1-5 shows an example of default inheritance. Here, the default color of elephant is grey, while the color of a white elephant is white, which is not a specialization of grey.

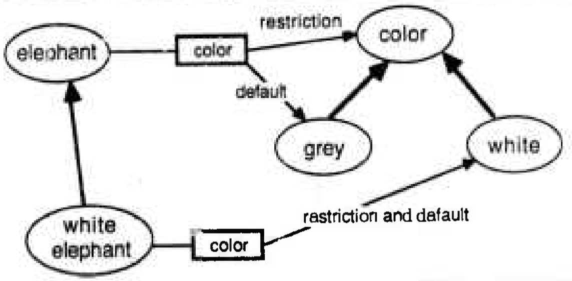


Figure 1-5: Restrictions and Defaults

1.5.1. Instantiation

We envision that a number of different instantiation mechanisms may be appropriate for KREME Frames. NIKL, as part of the KL-TWO system, instantiates concepts as predications in the RUP truth maintenance system [8]. On the other hand, MSG instantiated concepts as flavor instances, and this is the instantiation mechanism currently provided by KREME Frames. We plan to provide a truth maintenance system as an alternative form of instantiation in the future.

When a concept is defined, a corresponding flavor is also defined. This flavor is composed of the flavors corresponding to the concept's immediate parents and an additional flavor called KROBJECT which provides the additional functionality required for instances of KREME Frames.

Instances of a concept (also known as objects) are created by the MAKE-OBJECT function. MAKE-OBJECT creates an instance of the concept's corresponding flavor, installs defaults in unfilled slots, and installs coreference-handling objects in each slot for which a role equivalence was defined at the concept. The same coreference object is placed in all equivalent slots. These objects are "transparent" to the slot access and modification functions. Modifying any equivalenced slot changes the value of the coreference object, and accessing such slots returns the coreference object's value (rather than the object itself).

2. The Knowledge Editor

2.1. Background

The KREME Knowledge editor currently consists of three editor modules, a frame editor, a procedure editor, and a rule editor, and a large tool-box of editing techniques that are shared among the editor modules.

The original design goal was a global editing environment that could accommodate distinct editor modules for the various kinds of knowledge that would be represented. However, from the point of view of the user, there would be a single editor with the interfaces between the modules completely transparent. Moreover, the user would see a single, integrated knowledge base that had various means for organizing different types of knowledge. The user would move through this space by pointing at various knowledge chunks which would cause the system to present an appropriate view. Alternatively, the user could directly request a specific view for a specific piece of knowledge.

2.2. Basic Features

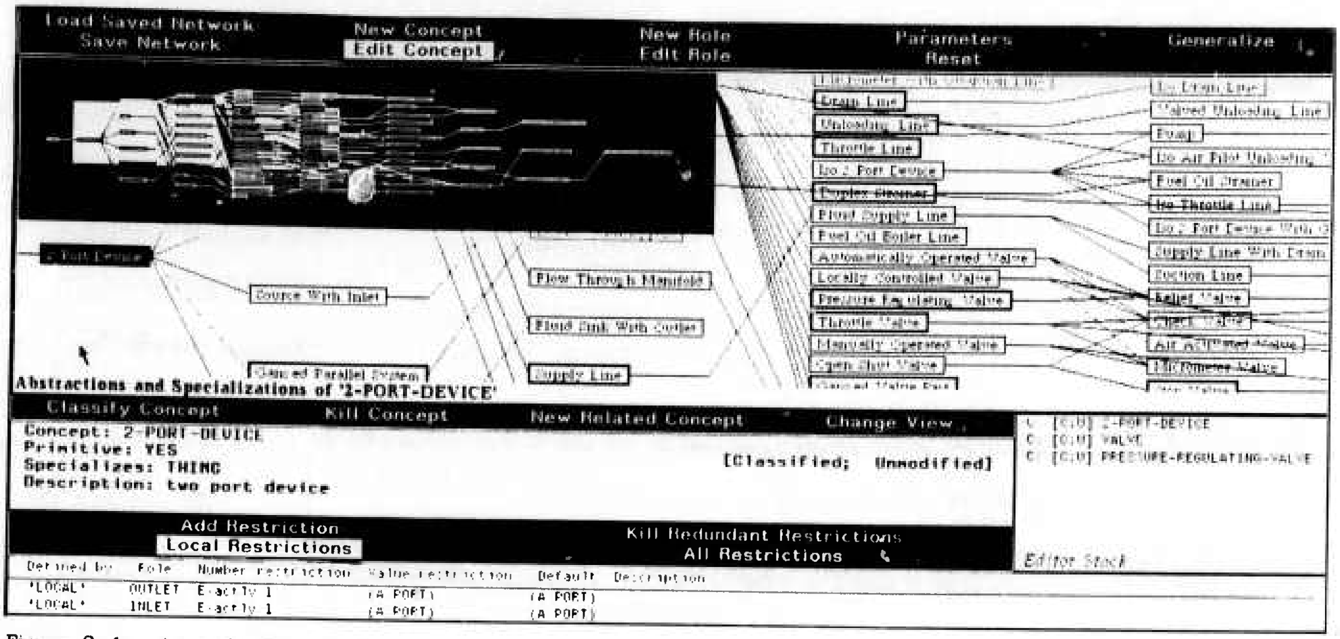
2.2.1. Views

Each distinct type of representation included in the system (currently concepts, roles, procedures and rules) has defined for it one or more views. A view is a collection of panes in a Symbolics window configuration, each of which displays some aspect of the particular piece of knowledge being edited and/or a set of editing operations on it. A view can show various aspects of the specific piece of knowledge as well as various details of the context in which the piece exists.

When the user desires to enter or edit a specific piece of knowledge, the system opens the most appropriate view for the type of knowledge and the editing operation requested. When editing a particular piece of knowledge, the user has available a menu of different views which are appropriate for different aspects of that knowledge and can be accessed from a menu.

[4] E.g., Number restriction: min = 4, max = 4; Value Restriction: (an ELEPHANT-LEG).

[5] A number restriction of *Exactly 1* (min = max = 1) is more specific then a number restriction of *At most 2*(min = 0, max = 2).



Concept: 2-PORT-DEVICE
Primitive: YES
Specializes: THING
Description: two port device

[Classified; Unmodified]

C [C;U] 2-PORT-DEVICE
C [C;U] VALVE
C [C;U] PRESSURE-REGULATING-VALVE

Defined by	Role	Number restriction	Value restriction	Default	Description
LOCAL	OUTLET	Exactly 1	(A PORT)	(A PORT)	
LOCAL	INLET	Exactly 1	(A PORT)	(A PORT)	

Figure 2–1: A g h with overview

2.2.2. Pointing

Pointing with the mouse is the primary means for performing editing operations, browsing, adding, and modifying definitions. In general, all visible references to an object can be pointed at, in order to view the object in more detail. For example, a concept can be displayed as a node in a graph, as a value restriction or default, as a parent of another concept, or as an item on the editor stack. Whatever the form of the display, the displayed item will respond to the same set of operations when someone points at it. Similarly, when the system requires the entry of a concept name, the user may either type the name or point at any visible concept name. In windows displaying features of concept definitions, pointing also is used to tell KREME to replace parts of those definitions.

Commands that cannot be performed by pointing directly at an object are usually contained in command menus which are associated with particular windows in each editor view. Such commands are used for changing views, entering new concept definitions, loading and saving taxonomies, etc.

2.2.3. The Grapher

The KREME grapher is a powerful, generalized facility that rapidly draws lattices of nodes and links. At present, its main use is to provide a dynamically updated display of the concept or role currently being edited and all of its classifier determined abstractions and specializations. Other concepts may be added to the displayed graph at any time simply by pointing at a node that is already present and requesting all of *its* abstractions or specializations to be displayed as well. Nodes and their children (or just the children) may also be concealed or removed from a presented graph if they are not relevant and are making it hard to read other portions of the graph. One may also point at nodes to show a textual form of their current definition and to edit the definitions (which pushes the current definition on the editor stack, as it does by pointing at it in other displays).

An important feature of the grapher is that it can display graphs that are much larger than the window through which it is viewed. When dealing with large taxonomies, pointing at the graph anywhere else but at nodes and dragging the mouse causes the grapher to pan in the direction of mouse motion, making previously obscured portions of the graph instantly visible as though one was moving a window across a larger page. The grapher also provides an "overview" facility to show the shape of the full graphed lattice. Pointing at positions in the overview is another way to move to a particular part of the lattice. Figure 2–1 shows a graph of one portion of the STEAMER frame base, with the overview exposed.

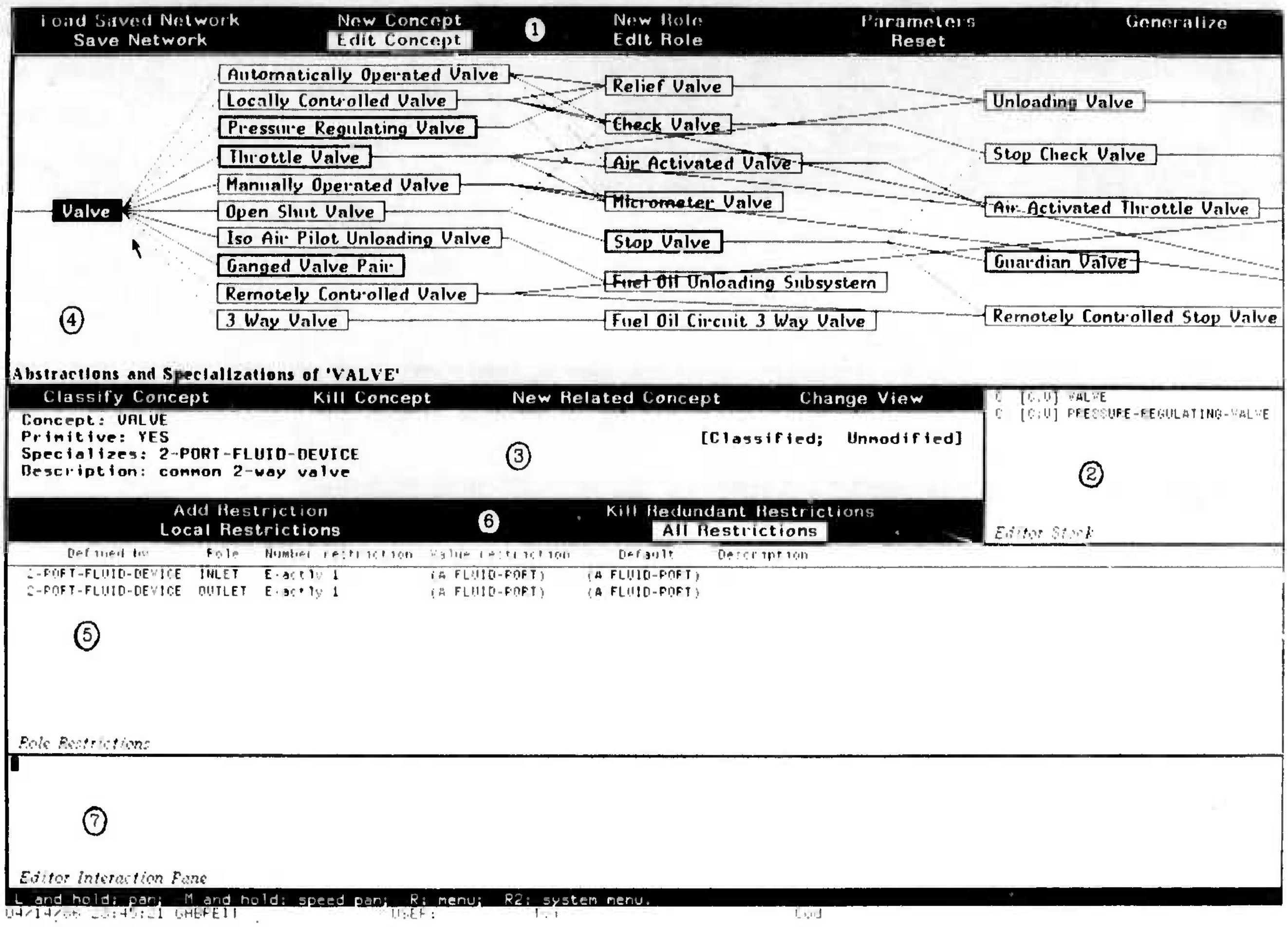
Currently, the grapher can be used to display only directed lattices with no loops, e.g., specialization hierarchies and relationships like part–whole. We expect to use the grapher to display arbitrary networks of relationships between between sets of concepts. These other kinds of views are critical for displaying partially ordered plan sequences, causal relationships and constraint systems in general.

2.2.4. Buffers and the Editor Stack

The editor maintains a level of indirection between the knowledge being edited and the representation of that piece of knowledge in the knowledge base. This is done by the mechanism of editor buffers, analogously to the distinction between a text editor buffer and an associated file. Changes are always made to *definition objects*, which can be subsequently classified. The editor maintains a stack or list of the objects that have been edited, and constantly displays this list, indicating which ones have been modified and not reclassified.

The top item in the stack is the definition currently being viewed and edited. The user is free to modify this definition in any way without directly effecting the knowledge base. When the modified definition is to be placed into the knowledge base a defining function appropriate to the type of knowledge (e.g., classification for concepts and roles), is executed and the knowledge base is modified.

The editor stack is always visible in its own window and provides one convenient method for browsing. The user may make any definition item currently in the stack the top, visible item by pointing at it. The object will be



```
Concept: VALVE
Primitive: YES                                  [Classified;  Unmodified]
Specializes: 2-PORT-FLUID-DEVICE
Description: common 2-way valve
```

```
C  [C,U] VALVE
C  [C,U] PRESSURE-REGULATING-VALVE
```

```
Defined by           Role    Number restriction  Value restriction  Default          Description
2-PORT-FLUID-DEVICE  INLET   Exactly 1           (A FLUID-PORT)     (A FLUID-PORT)
2-PORT-FLUID-DEVICE  OUTLET  Exactly 1           (A FLUID-PORT)     (A FLUID-PORT)
```

```
04/14/86 23:45:21 GHEFELI        USER:
```

Figure 2-2: The Main Concept Editing View

displayed in the same editor view as when it was last edited.

2.2.5. Files and Multiple Language Support

All definitions manipulated by the editor are read and stored in lisp-readable text files of defining forms. Files are created by the SAVE command which converts each of the items of the current knowledge base to its LISP defining form and writes it to the file specified. The files are in human readable form and can be edited offline using an ordinary text editor. In fact, KREME can read files that were developed independently using a text editor or some other frame editor.

Files are read in using the LOAD command. A file can be loaded into a blank KREME knowledge base or can be loaded on top of an already existing knowledge base. This mechanism, which relies heavily on the use of the classifier to keep things coherent, enables KREME to organize information from multiple knowledge bases to create a single unified whole.

KREME currently will read and write definitions in either its own frame language syntax or in NIKL syntax. In addition, there is some customization of the displays viewed while editing networks in either of these languages (e.g., the presence of defaults in role restrictions). This flexibility makes it possible for KREME to be used regularly to examine and update a knowledge base of approximately 1000 roles and concepts for a natural language query system that was built using KL-TWO. KREME can also read files of MSG defining forms, providing us access to the extensive STEAMER knowledge base of concepts and procedures.

We feel that this multiple language handling facility is a crucial feature of KREME and are committed to extending it, where possible, to other representation languages. A rich library of input translation programs will enable a knowledge base builder, working in KREME, to draw upon many previously existing knowledge bases to create a larger and more detailed whole. It is our opinion that this kind of flexibility will be crucial if knowledge bases developed in different languages are ever to related and conveniently modified to create a greater whole. Given the large intersection of features provided by most current-day frame language representation systems, we do not see this as an impossible goal. In the near future, we will be considering extensions of KREME Frames to provide an environment in which many KEE knowledge bases could conceivably be edited. One of our goals in redesigning the classifier was to make such extensions feasible.

2.3. The Frame Editor

The editor for the KREME Frames representation language is the most fully realized editor in the KREME system. Although we have a host of improvements and additions planned for it, the current operational version of the frame editor is already an extremely useful tool for the creation, modification and viewing of KREME Frames networks. The main components of the frame editor are discussed in the section which follows.

2.4. Windows and Views

The current KREME frame editor has six views, each a fixed configuration of windows appearing at once on the screen. Three windows (screen regions) are common to all of these views, the **global command window**, the **editor stack window**, and the **state window**. Figure 2-2 shows the main concept editing view, which contains most of the windows used for editing portions of a concept's definition. The descriptions of each window below will refer to the numbers superimposed on that figure.

The **global command window** (1) contains commands that operate on the network as a whole. It is always visible.

The **editor stack window** (2), which is also always visible, shows the names of the things being edited and some information about their current edit state (e.g., whether they have been modified). Items in the stack window can be removed from the editor, made the currently visible edit item, or reclassified (if modified) by pointing at them.

The **state window** (3), which is visible in all views for concepts and roles, displays the name, textual description, primitive class flag, parents and information on the classification state of the item.

The **concept graph window** (4) displays a dynamically updated graph of all of the abstractions and specializations of the current concept. This view provides constant visual display of the relative position of the concept being edited in the subsumption hierarchy.

The **role restrictions window** (5) displays a table of the role restrictions for the current concept. Columns in the table show the source (where it was inherited from) of the restriction, its role name, value and number restrictions, default value, and a description.

This window can also be used to display the concept's *inverse role restrictions*, which are all of the restrictions that use the concept as their value restriction. This display resembles the role restrictions display, though some parts of it cannot be edited.

The **role restrictions command window** (6) This menu contains commands for the role restrictions window. Currently, commands are available to display the locally defined restrictions, the full inherited set of restrictions, or the inverse restrictions. In addition, there is a command to delete redundant defined restrictions that would be inherited anyway.

The **Editor Interaction Window** (7) is a Lisp Listener which can be scrolled backward and forward through a history of the current session. This window also is used for some data entry and messages.

Four other views are currently defined for concepts, and one view is defined for roles.

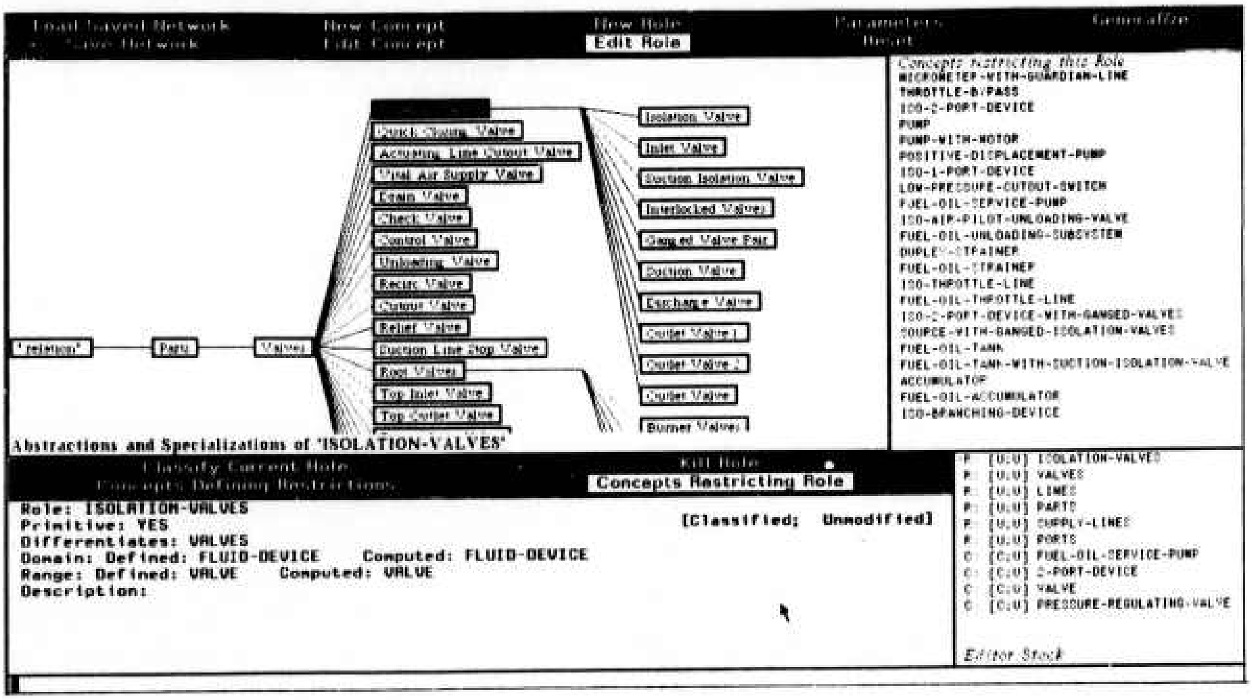
The **role editing** view (figure 2-3) appears whenever the *Edit Role* or *New Role* commands are issued. It contains windows showing a graph of the role network highlighting the currently visible role, and another displaying the concepts that restrict the role. The role editing view also contains a role editing commands window.

The four other concept views mix some of the windows above with windows for displaying and editing disjoint classes, role equivalences, and inverse role restrictions. In addition to the global commands window, the editor stack and state windows, these views show the following.

- An enlarged graph window, filling most of the screen, for viewing large sections of the concept hierarchy. (No display or commands for editing role restrictions are provided in this view.)

- Windows for a concept's inverse restrictions, role restrictions, equivalences and disjoint classes, but no graph.

- Enlarged regions for all concept features, role restrictions, equivalences and disjoint classes (but no graph).



Abstractions and Specializations of 'ISOLATION-VALVES'

Concepts restricting this Role

```
MICROMETER-WITH-GUARDIAN-LINE
THROTTLE-BYPASS
ISO-2-PORT-DEVICE
PUMP
PUMP-WITH-MOTOR
POSITIVE-DISPLACEMENT-PUMP
ISO-1-PORT-DEVICE
LOW-PRESSURE-CUTOUT-SWITCH
FUEL-OIL-SERVICE-PUMP
ISO-AIR-PILOT-UNLOADING-VALVE
FUEL-OIL-UNLOADING-SUBSYSTEM
DUPLEX-STRAINER
FUEL-OIL-STRAINER
ISO-THROTTLE-LINE
FUEL-OIL-THROTTLE-LINE
ISO-2-PORT-DEVICE-WITH-GANGED-VALVES
SOURCE-WITH-GANGED-ISOLATION-VALVES
FUEL-OIL-TANK
FUEL-OIL-TANK-WITH-SUCTION-ISOLATION-VALVE
ACCUMULATOR
FUEL-OIL-ACCUMULATOR
ISO-BRANCHING-DEVICE
```

```
Role: ISOLATION-VALVES                                   [Classified;  Unmodified]
Primitive: YES
Differentiates: VALVES
Domain: Defined: FLUID-DEVICE    Computed: FLUID-DEVICE
Range: Defined: VALVE    Computed: VALVE
Description:
```

```
P  [U:U] ISOLATION-VALVES
P  [U:U] VALVES
P  [U:U] LINES
P  [U:U] PARTS
P  [U:U] SUPPLY-LINES
P  [U:U] PORTS
C  [C:U] FUEL-OIL-SERVICE-PUMP
C  [C:U] 2-PORT-DEVICE
C  [C:U] VALVE
C  [C:U] PRESSURE-REGULATING-VALVE
```

Editor Stack

Figure 2-3: The Role Editing View

- The structure editing windows and the macro editor displays, described in section 4 below.

2.5. Operations

The basic operations used to make new concepts or roles, change existing ones, and delete concepts and roles from the network are discussed in the sections which follow.

Making new concepts. Clicking on the *New Concept* command in the global command menu will cause a menu of possibilities to pop up. From this pop-up menu, the user can choose to make a new concept that is similar to the currently visible concept or to some other concept, a specialization of the current concept or some other concept, or a specialization of several concepts.

When the initial form for the new concept has been specified the system creates a new concept definition for it and shows this new definition in the main concept view. The user is then free to add specific details (slots, equivalences, additional parents, etc.) to the new concept definition, classify it, or edit other concepts, leaving the new concept definition on the editor stack to be finished and classified later. There are no constraints on the order of these operations. The new concept definition is treated like any other concept definition in an editor buffer, except that it is marked as never having been classified.

Making new roles. The operations for adding new roles are essentially the same as those for making new concepts.

Adding and modifying slots. Whenever the window displaying role restrictions is visible, as in the main concept view, role restrictions can be added or modified. A new slot is added to the defined slots of the concept with the *Add Slot* command. When this command is issued, the system asks for a role name, a value restriction, a number restriction and a default form. Any of these items can be entered by typing or by pointing to the desired name or form if it is visible. If a role or concept named in a role restriction or default does not exist the system will offer to make one with the name given.

The user may modify any defined slot or any slot that is inherited from a parent or created by the classifier. Slots are modified by pointing at the appropriate subform and then either typing in or pointing to a replacement form. If any portion of an inherited or classifier created slot is modified, the new slot definition becomes part of the definition of the concept heing edited.

Modifying parents. The system displays the classifier determined parents of a concept in two places in the main concept view. The concept graph displays them as part of the abstraction hierarchy of the concept. In addition, the state pane shows both the defined and direct or computed parents of the concept. The classifier may have found that the concept specializes some concepts more specific than the defined parents, thus defined parents may or may not be direct parents. In the state pane, defined parents that are not direct parents are preceded by a "-", while classifier determined parents that were not defined parents are preceded by a "+".

Adding new defined parents to a concept's definition is done by clicking on the *Add Parent* command and typing a concept name or pointing to any visible concept. The system prohibits users from defining concepts as parents of concepts which subsume them. (This would form an abstraction-specialization loop.)

Defined parents may be deleted by clicking on their names in the list of parents displayed in the state window. A parent can either be deleted or "spliced out". Splicing out a parent both deletes that parent from the list of defined abstractions and makes the deleted parent's parents parents of the current concept. That is, it connects the current concept to (some of) its grandparents. Commands are also available to delete all defined parents that the classifier has determined are not direct parents, and to make all classifier-discovered parents part of the concepts definition.

Changing names and killing concepts and roles. KREME allows the user to change the names of concepts and roles or to delete them completely. Name changing is accomplished simply by pointing at the concept or role's name in the state pane and entering a new name. Changing the name of a concept or role directly effects the network, since the name of the concept definition, as well as the name of the corresponding classified concept (if there is one), is changed. All pointers to the concept (as a parent of other concepts, in value restrictions, as the domain or range of roles etc.,) are automatically updated with the new name both in the classified network and in all editor buffers.

Killing concepts is a somewhat complicated operation, because of the need to reconfigure the network following the deletion. In essence the *Kill* command splices a concept out of the taxonomy by connecting all of its children to all of its parents. Any concept that used to define the concept as a parent is reclassified. If the concept was used as a value restriction, the editor tries to find an appropriate parent to substitute for the killed concept. Because this attempt is not always successful, user interaction is sometimes required.

Our current version of *Kill* is only one of several that might prove useful. For example, We plan to provide a second kill function that deletes the entire lattice under the killed concept (the concept and all of its children) and a third *Kill* function that preserves the properties of the killed concept by either moving them up to the concepts parents or down to all of its children.

Adding and deleting equivalences or disjoint classes. KREME provides commands to add equivalences and disjoint classes. For equivalences, the user enters two paths whose referents are to be equated, and the system checks to make sure that both paths are valid (all slots along the path are defined) and that the referents of the paths are subsumption related to each other (that is, the restrictions on the referents of both paths are consistent). For disjoint classes, the system checks whether the concept entered can be disjoint from the current one (i.e., a concept cannot be disjoint from its parents). To delete an equivalence or disjoint concept the user merely clicks on its display in the equivalence or disjoint concept window, respectively.

Deleting redundant slots. Clicking on the *Delete Redundancies* command causes the system to delete any defined slots whose definitions are the same as the inherited definitions. This operation alters the definition of the concept, but not its classification or completed description.

3. Classification in KREME-FRAME networks

3.1. Background

One of the most time consuming tasks in building knowledge bases is maintaining internal consistency. Adding, deleting and modifying slots and parents in a frame taxonomy may affect the subsumption relations between frames and, perhaps more important, may alter the sets of properties inherited by more specific frames. The possible consequences of a change in one part of a network grows rapidly as taxonomies get larger. Consequently, the size and complexity of knowledge bases is limited by the extent to which automatic means are provided for consistency checking.

A central feature of the NIKL representation language is a classification algorithm that allows one to build networks of NIKL concepts that are not only consistent (all subsumption links in the network are consistent with the sets of properties enclosed by nodes) but also, for all practical purposes, complete (all subsumption links in the network that are logically entailed by the sets of properties enclosed by the nodes are explicit in the network).

Unfortunately the NIKL classifier can only handle monotonic changes to a concept hierarchy. NIKL can construct a consistent and complete network from a file of randomly ordered concept definitions and users may add new concept definitions to existing networks, but once a concept has been placed in a network, it cannot be modified or deleted, a severe shortcoming for an interactive knowledge editor.

In order to develop a fully interactive knowledge editing system we had to extend the NIKL classifier so that it could deduce all of the consequences of any modification to any part of the intertwined concept/role taxonomies, and effect the *reclassification* of all concepts and roles necessary to maintain internal consistency.

The remainder of this section will give a brief description of the KREME classifier. For a formal description of the NIKL classifier algorithm see [14, 15]. For a more complete description of a somewhat simpler interactive classifier see [1].

3.2. Completion

Completion refers to the basic inheritance mechanism used by KREME Frames to install all inherited features of a concept in its internal description. Given a set of defined parents and a set of defined features, the completion algorithm determines the full, logically entailed set of features at a concept (or role). Completion always occurs before classification or reclassification of a role or concept.

The completion algorithm is broken up into modular chunks that correspond to the decomposition of the frame language. There is a distinct component that deals with role restriction inheritance, another component that deals with disjoint class inneritance, a third that deals with role equivalence inheritance and so on. This organization makes it quite straightforward to extend the language with new features that handle inheritance in different ways.

A concept inherits all the role restrictions from all of its direct parents and adds them to the list of restrictions that it defines locally. For each role naming a slot in the combined list, the algorithm creates a single restriction that conjunctively combines all restrictions for that role at that concept. The effective value restriction is either the single most specific of all the value restrictions for that role at the concept, or a conjunction of several, if no single one is subsumed by all the others. The effective number restriction for each slot is similarly determined by intersecting the number ranges in all of that slot's role restrictions.

Complications arise when there is more than one restriction for a given role in the initial list, none of which is more specialized than all of the others. Figure 3-1 illustrates one way this can occur, when the most specific value restriction is inherited from one parent (ANIMAL) and the most specific number restriction is inherited from another parent (4-LIMBED-THING) to form the restriction of LIMBS at 4-LIMBED-ANIMAL.

Figure 3-2 shows another example of completion in which the resulting value restriction must logically be the conjunction of several concepts. Since ANIMAL-WITH-LEGS is an ANIMAL, and a THING-WITH-LEGS all of its LIMBS must be both ORGANIC-LIMBs and LEGs. If the concept ORGANIC-LEG, specializing both ORGANIC-LIMB and LEG, exists when ANIMAL-WITH-LEGS is classified for the first time, the classifier will find it and make it the value restriction of the slot LEGS at ANIMAL-WITH-LEGS. If it does not exist, the classifier stops and asks if the user would like to define it.

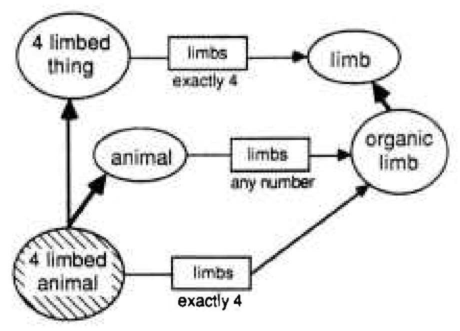


Figure 3-1: Inheriting Number and Value Restrictions

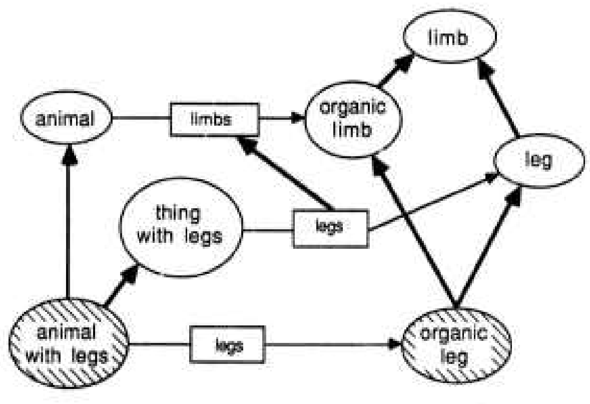


Figure 3-2: Combining Value Restrictions

In general, whenever a value restriction can only be defined as a conjunction of several concepts, KREME offers to form a concept representing the conjunction, and asks for a name for the new concept.[6] As it turns out, forming the suggested conjunction is not always the right thing to do. It often indicates a missing subsumption relationship between the concepts involved. KREME provides several options at this point, as described in section 3.6.

3.3. Subsumption checking

The KREME classifier algorithm is built around a modularly constructed test for a valid subsumption relationship between two objects, based on their effective, inherited features. When a definition is being classified, it is repeatedly compared to other, potentially related, objects in the lattice to see whether its completed definition subsumes or is subsumed by those other objects. The subsumption test compares features of one with features of the other. For C1 to subsume C2 in this sense means that the features of C1 form a proper subset of the features of C2.

KREME partitions the work of this subsumption check in much the same way it deals with inheritance. Each feature type (i.e role-restriction, disjoint-class etc) decides whether, with respect to that type, C1 subsumes C2, C1 is equivalent to C2, or C1 does not subsume C2. If any of these tests return DOES-NOT-SUBSUME, the the entire subsumption check fails immediately. If all of the checks return EQUIVALENT or SUBSUMES, then the subsumption test succeeds as long as there was one vote for SUBSUMES. The advantage of this kind of modular organization is extensibility. If a new feature that contributes to concept subsumption is added to the language one need only define a subsumption predicate for that feature, and objects having that feature will be appropriately classified.

3.4. The Classifier

The basic classifier algorithm takes a completed definition (that is, a definition plus all its effective, inherited features) and determines that definition's single appropriate spot in the lattice of previously classified definitions. The result of a classification is a unique set of the most specific objects that subsume the definition and a unique set of the most general objects that are subsumed by the definition. When the classified definition is installed in the lattice all the concepts that subsume its features will be above it in the lattice and all the concepts that are subsumed by its features will be below it.

The details of the classifier's implementation and operation are beyond the scope of this paper. It should be noted that the basic classifier is nearly functionally equivalent to the NIKL classifier. However, NIKL merges concepts that are exactly classifier equivalent, while the KREME classifier does not normally do this. The decision not to merge concepts in KREME is due in part to the different environments in which these classifiers are being used. In an editing environment, where definitions are expected to change, there may be more to a concept's definition than had been stated when it was first defined. In addition, we foresee a time when not all of a concept's defined properties are classifier sensitive. In such an environment, merging concepts when their classifier sensitive properties are identical would be a mistake.

3.5. Reclassification of KREME networks

We are now ready to give a brief description of the mechanism that KREME uses to propagate modifications of a definition to related concepts and roles. The KREME classifier is invoked whenever a concept or role is defined or redefined. The classifier first *completes* the definition by gathering all of its inherited features, and then determines exactly where it should be placed in the lattice. If the object has never been classified before, the basic classifier algorithm is run to find the most specific parents and children of the completed definition, and insert the new object into the network.

If the new definition redefines a previously classified object, the process is more complex. First, the previously classified object must be spliced out of the network, and the basic classifier algorithm is run to find the correct position for the new definition. Since changing the subsumption relationships of an object can change the positions of objects referring to it, the reclassifier must then find all other objects that must be reclassified because of the change. The system compares the previously classified object with the redefined object in order to determine which other objects, dependent on the old definition, might be affected by the change. These objects must all be reclassified. As one might expect, reclassifying those other objects may itself cause further reclassifications to be necessary.

The reclassification algorithm which accomplishes this resembles the consistency maintenance algorithms found in truth maintenance systems. A queue of objects waiting for reclassification is maintained, called the *pending reclassification queue*. As each object is reclassified, all objects that could be affected by the changes caused by its reclassification are collected and placed in the queue if they weren't there already.[7]

Although the above algorithm is relatively straightforward in outline, its efficiency and correctness depends on determining exactly those dependent objects that need reclassification. The algorithms efficiency depends on reclassifying only those objects that require it (i.e., whose classifier determined position may change). Its accuracy and completeness depend on reclassifying all objects which require it.

The power of reclassification in an editing environment can be illustrated with the following relatively simple example. Suppose a knowledge base developer had defined both GASOLINE-POWERED-CAR and INTERNAL-COMBUSTION-POWERED-CAR as specializations of CAR, but had inadvertantly defined INTERNAL-COMBUSTION-ENGINE as a kind of GASOLINE-ENGINE. In this situation, the classifier would deduce that INTERNAL-COMBUSTION-POWERED-CAR must be a specialization of GASOLINE-POWERED-CAR, as shown in figure 3-3, since the former restricted the role ENGINE to a subclass of the latter's restriction of the same role.

[6] The NIKL classifier forms such conjunctive concepts automatically, but does not give them names.

[7] Concepts that depend on each other pose special problems, but the details of how this is handled are beyond the scope of this document.

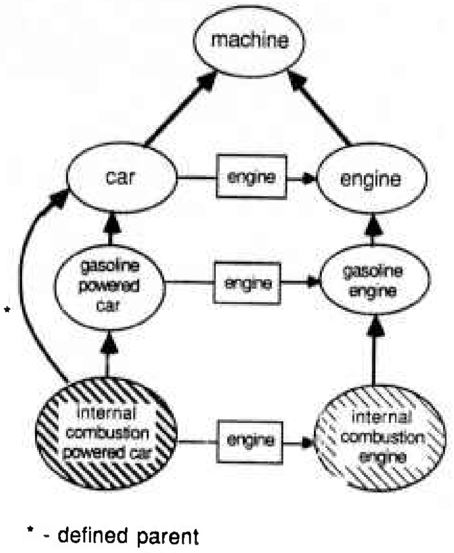


Figure 3-3: An Error Affecting Classification

Redefining INTERNAL-COMBUSTION-ENGINE as a kind of ENGINE, rather than a GASOLINE-ENGINE and reclassifying causes all of INTERNAL-COMBUSTION-ENGINE's dependents to also be reclassified, including INTERNAL-COMBUSTION-POWERED-CAR. Since GASOLINE-ENGINE no longer subsumes INTERNAL-COMBUSTION-ENGINE, the restrictions for GASOLINE-POWERED-CAR no longer subsume those of INTERNAL-COMBUSTION-POWERED-CAR, and the classifier therefore finds that GASOLINE-POWERED-CAR does not subsume INTERNAL-COMBUSTION-POWERED-CAR. This is shown in figure 3-4.

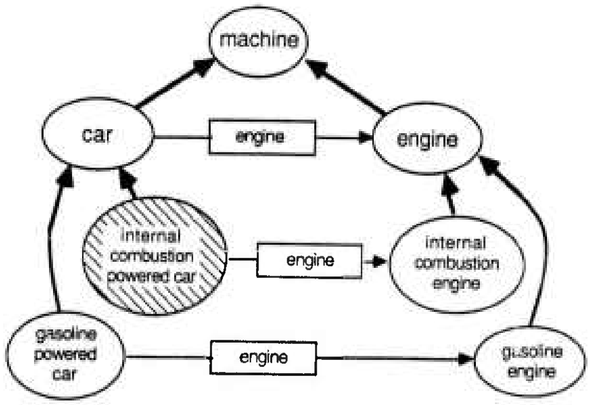


Figure 3-4: After Reclassification

3.6. Editor Interactions with the Classifier

The following sections describe several ways in which the frame editor and the classifier interact to support the knowledge acquisition process.

Defined/Completed feature displays. The frame editor uses the classifier's completion algorithm in its display and editing of role restrictions (slots), role equivalences and disjoint classes, and in displaying the set of all concepts using a role in a value restriction (in the role editing view). When the role restrictions window is visible, the user may toggle between a display that shows the defined role restrictions for the current concept and a display that shows all the effective role restrictions at the concept. When all role restrictions are displayed, the user may modify a restriction that was inherited or created by the completion algorithm. Modifying a restriction automatically adds the modified restriction to the list of defined restrictions at the concept. Similar mechanisms are available for viewing and modifying role equivalences, disjoint concepts and concepts restricting a role.

Classification from the editor. One especially useful feature of the KREME frame editor is its ability to immediately display the effects of classifying a concept or role definition. When the user modifies a concept or role's definition and classifies it, the editor redisplays the relevant visible windows to show all classifier added information. For example, the graph of a concept will show the concept's possibly modified place in the taxonomy. Links added or deleted by the classifier seem to appear or disappear instantaneously.

Making new concepts and roles needed by the classifier. The KREME classifier sometimes needs to form new concepts in order to satisfy some logical relationship. This occurs primarily during role restriction completion, when the effective value restriction for a slot can only be described as a conjunction of two defined concepts, rather that a single concept (See section 3.2). It also happens occasionally when a similar condition arises in determining the effective restriction on the range of a role. These classifier required conjunctions are sometimes called CMEETs.

While forming the appropriate conjunction is the logically correct thing to do to ensure consistency of the knowledge base *as then defined*, it often turns out that the conjunction suggested by the classifier is needed because one of the concepts to be conjoined has been improperly defined. In particular, a CMEET condition most frequently arises because the concept used as the value restriction of a role in the concept being classified is not subsumed by the restriction for the same role at a higher concept, and the restriction must logically satisfy both constraints. This is illustrated in figure 3-5. The figure shows 2-PORT-TANK defined as both a TANK and a 2-PORT-DEVICE. Each of those concepts restricts the role INLET-VALVE. The classifier finds that the restriction for slot INLET-VALVE at 2-PORT-TANK must be both a VALVE and a STOP-VALVE, given the restrictions of that slot at 2-PORT-TANK's parents. Since STOP-VALVE was not defined as a kind of VALVE, the conjunction is not the single concept STOP-VALVE, and so the classifier asks if it should create a new concept, the CMEET of VALVE and STOP-VALVE.

Whenever the KREME classifier requires that a CMEET be formed, it stops and queries the user, explains the situation and requests a name for the concept to be formed for the conjunction, and enumerates several alternative options. If all of the concepts are defined correctly, and the proposed CMEET correctly describes the required restriction, the user simply enters a name for the new concept and classification continues. If the problem really lies with an existing definition, as is the case with VALVE and STOP-VALVE, the user can choose an alternative course of action, rather than introducing a useless new concept. Most often, the correct action is to alter the subsumption relations between the named concepts, so that one of them is subsumed by the others.

This is done simply by naming one of the concepts to be conjoined instead of giving a new name. In our example, the user would simply type STOP-VALVE, in response to the query. The classifier would then make STOP-VALVE a kind of VALVE and continue classifying 2-PORT-TANK, resulting in the relations shown in figure 3-6

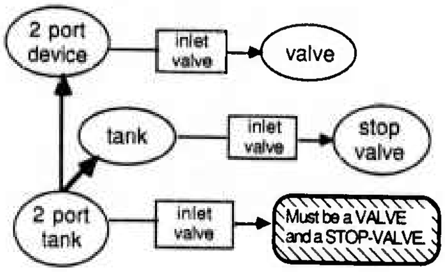


Figure 3-5: Discovering a missing subsumer.

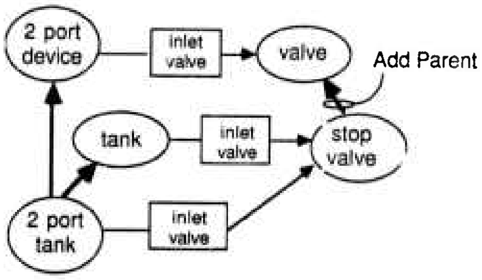


Figure 3-6: After interaction with the classifier

This interaction effectively allows a user to correct an oversight in a previously defined concept's definition at the point the error is detected by the classifier's completion algorithm. By making the classifier less "automatic" in this way, we have made it more effective as a consistency maintenance tool, and avoided some of the problems incumbant in using a classifier with a less than totally complete and accurate knowledge base.

We are investigating additional ways in which the classifier, as well as other kinds of consistency maintenance facilities, can be used interactively to aid the acquisition and refinement of knowledge bases. We feel this kind of functionality will become increasingly important as the size of knowledge bases grows.

4. Macro Editing of Knowledge Bases

An important focus of the first phase of the BBN Knowledge Acquisition Project that will be continued in phase two is an investigation of and development of tools supporting *macro-editing* procedures for automatic modification and enhancement of partially defined knowledge bases. The need for methods of expressing and packaging conceptually clear *reformulations* of concepts and other representations, as well as similar facilities for developing new concepts from old ones is clear.

We are taking two different approaches to this problem. First, we have developed a macro facility for reformulations that can be expressed as sequences of standard, low-level editing operations which allows users to define editing macros that can be applied to sets of concept definitions by giving a single example. Second, we are building a small library of functions providing operations that cannot be defined simply as sequences of low level editing operations. Our main purpose is to collect and categorize these utilities, and explore their usefulness in a working environment. Our hope is that a large fraction of these operations can be conveniently described using the macro facility, as it is more accessible to an experimental user community than any set of "prepackaged" utilities, and can be more responsive to the, as yet, largely unknown special needs of that community

The current state of this research effort is described below. First, we will describe and provide illustrations of the macro-editing facility. Then we will describe an example of the latter class of operations, a "generalization" utility for discovering and presenting potentially useful generalizations of concepts to the knowledge engineer.

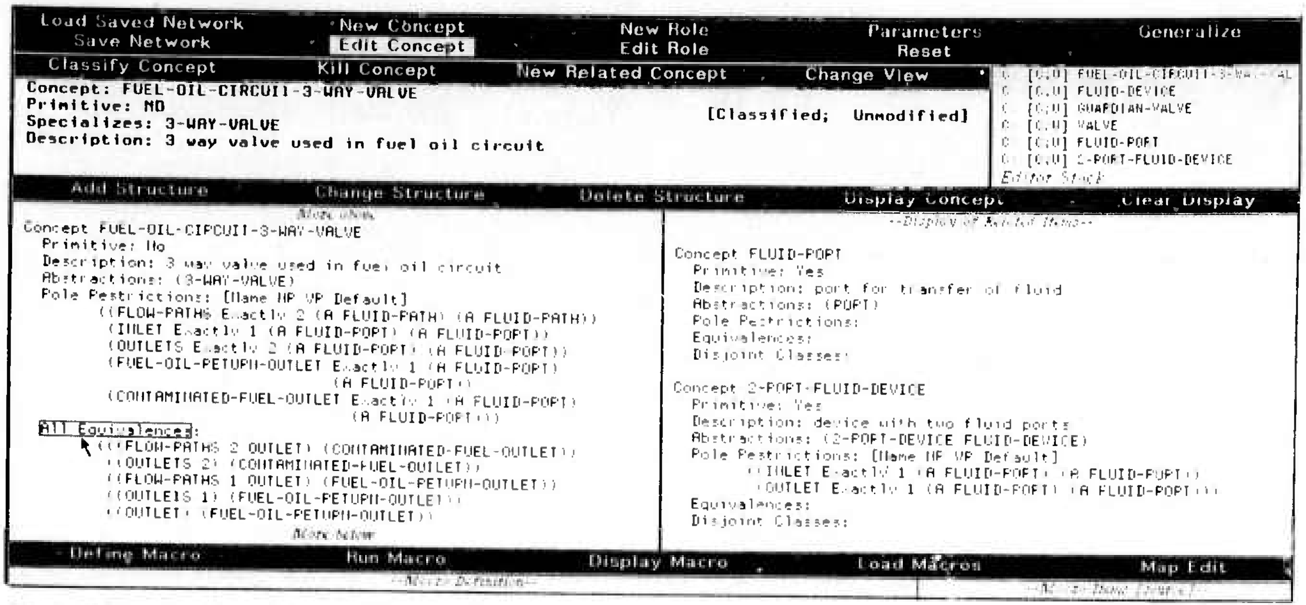
4.1. The Macro and Structure Editor

One of the views available when editing concepts in KREME is the *macro and structure editor*. This view (See figure 4-1.) provides display and editing facilities for concept definitions, which is based loosely on the kind of structure editor provided in many LISP environments. The view provides two windows for the display of stylized defining forms for concepts. The *current edit window* displays the definition of the currently edited concept (the top item on the editor stack). The *display window* is available for the display of any number of other concepts. Any concept which is visible in either window can be edited, and features can be copied from one concept to another by pointing. Both windows are scrollable to view additional definitions as required.

As in the normal KREME editing views, both inherited and defined features can be displayed. Clicking the mouse over the keyword indicating each feature class in a concept's definition (e.g., Abstractions., Role Restrictions., Equivalences., etc.) toggles the display of that component between defined and all inherited features of that type. That is, clicking on the *Role Restrictions* changes the display of the concept's role restrictions from locally defined role restrictions to *All Role Restrictions* and vice versa.

There is a menu of commands for displaying and editing definitions that includes the commands **Add Structure**, **Change Structure**, **Delete Structure**, **Display Concept** and **Clear Display**. Arguments (if any) to these commands may be described by pointing or typing. Thus, to delete a role restriction, one simply clicks on **Delete Structure** and the display of the restriction to be deleted. Adding a structure is done by clicking on **Add Structure**, the keyword of the feature class of the concept one wishes to add to (e.g., **Role Restrictions:**) The new restriction itself may be copied from a displayed concept by pointing, or a new one may be entered from the keyboard. Changing (that is, replacing) a structure can be done either by pointing in succession at the **Change Structure** command, the item to be replaced, and the thing to replace it with. In most cases, **Change Structure** can also be invoked simply by pointing at the structure to be replaced, without the menu command

The last two commands in the structure view's main menu provide the means to change what is displayed in the display window. Pointing at **Display Structure** and then at any visible concept name places the definition of



```
Concept: FUEL-OIL-CIRCUIT-3-WAY-VALVE
Primitive: NO
Specializes: 3-WAY-VALVE                                  [Classified;  Unmodified]
Description: 3 way valve used in fuel oil circuit
```

```
C  [C;U] FUEL-OIL-CIRCUIT-3-WAY-VAL
C  [C;U] FLUID-DEVICE
C  [C;U] GUARDIAN-VALVE
C  [C;U] VALVE
C  [C;U] FLUID-PORT
C  [C;U] 2-PORT-FLUID-DEVICE
```

```
Concept FUEL-OIL-CIPCUIT-3-WAY-VALVE
  Primitive: No
  Description: 3 way valve used in fuel oil circuit
  Abstractions: (3-WAY-VALVE)
  Pole Pestrictions: [Name NP VP Default]
        ((FLOW-PATHS Exactly 2 (A FLUID-PATH) (A FLUID-PATH))
         (INLET Exactly 1 (A FLUID-PORT) (A FLUID-PORT))
         (OUTLETS Exactly 2 (A FLUID-PORT) (A FLUID-PORT))
         (FUEL-OIL-PETURN-OUTLET Exactly 1 (A FLUID-PORT)
                                 (A FLUID-PORT))
         (CONTAMINATED-FUEL-OUTLET Exactly 1 (A FLUID-PORT)
                                   (A FLUID-PORT)))
  All Equivalences:
       (((FLOW-PATHS 2 OUTLET) (CONTAMINATED-FUEL-OUTLET))
        ((OUTLETS 2) (CONTAMINATED-FUEL-OUTLET))
        ((FLOW-PATHS 1 OUTLET) (FUEL-OIL-PETURN-OUTLET))
        ((OUTLETS 1) (FUEL-OIL-PETURN-OUTLET))
        ((OUTLET) (FUEL-OIL-PETURN-OUTLET)))
```

```
Concept FLUID-PORT
  Primitive: Yes
  Description: port for transfer of fluid
  Abstractions: (PORT)
  Pole Pestrictions:
  Equivalences:
  Disjoint Classes:

Concept 2-PORT-FLUID-DEVICE
  Primitive: Yes
  Description: device with two fluid ports
  Abstractions: (2-PORT-DEVICE FLUID-DEVICE)
  Pole Pestrictions: [Name NP VP Default]
        ((INLET Exactly 1 (A FLUID-PORT) (A FLUID-PORT))
         (OUTLET Exactly 1 (A FLUID-PORT) (A FLUID-PORT)))
  Equivalences:
  Disjoint Classes:
```

Figure 4-1: The Macro Structure Editor View

that concept in the display window. **Clear Display** removes all items from the display window. Individual concepts can be deleted from the display window by pointing at them and clicking. The **Edit Concept** command is used to change what is displayed in the current edit window. Editing a new concept moves the old edit concept to the bottom of the display window.

4.2. Developing Macro Editing Procedures

These operations, together with the globally available commands for defining new concepts and making specializations of old concepts essentially by copying their definitions, provide an extremely flexible environment in which to define and specify modifications of concepts with respect to other defined concepts. Virtually all knowledge editing operations can be done by a sequence of pointing steps using the current edit window and the display window. This style of editing is also used in the rule editor (See section 5.). This combination of editing features and mouse-based editor interaction style provides an extremely versatile environment for the description, by example, of a large class of editing macros.

The remaining windows in the Macro and Structure Editor View are used for defining, editing, and running macros composed of structure editing operations. Macro operations are defined by editing a concept for which the macro will make sense, and then invoking the **Define Macro** command from a menu. Until the macro definition is terminated, all editing and concept display operations performed are recorded as steps in the macro. Some basic facilities are also provided for editing (inserting and deleting steps, changing referents) macros once they are defined.

If the macros defined in this fashion are intended to work on concepts other than those for which they were defined, the operations recorded cannot refer directly to the concepts or objects which were being edited when the macro was defined. Instead, a kind of implicit variablization takes place, to replace the named objects with their relationship to the initially edited object. In most cases, these indirect references can be thought of as references to the *location* of the object in the structure editor's display windows. In fact, each new object that is displayed or edited in the course of defining a macro is placed on a stack called the *macro items list*, together with a pointer to the command that caused the item to be displayed.

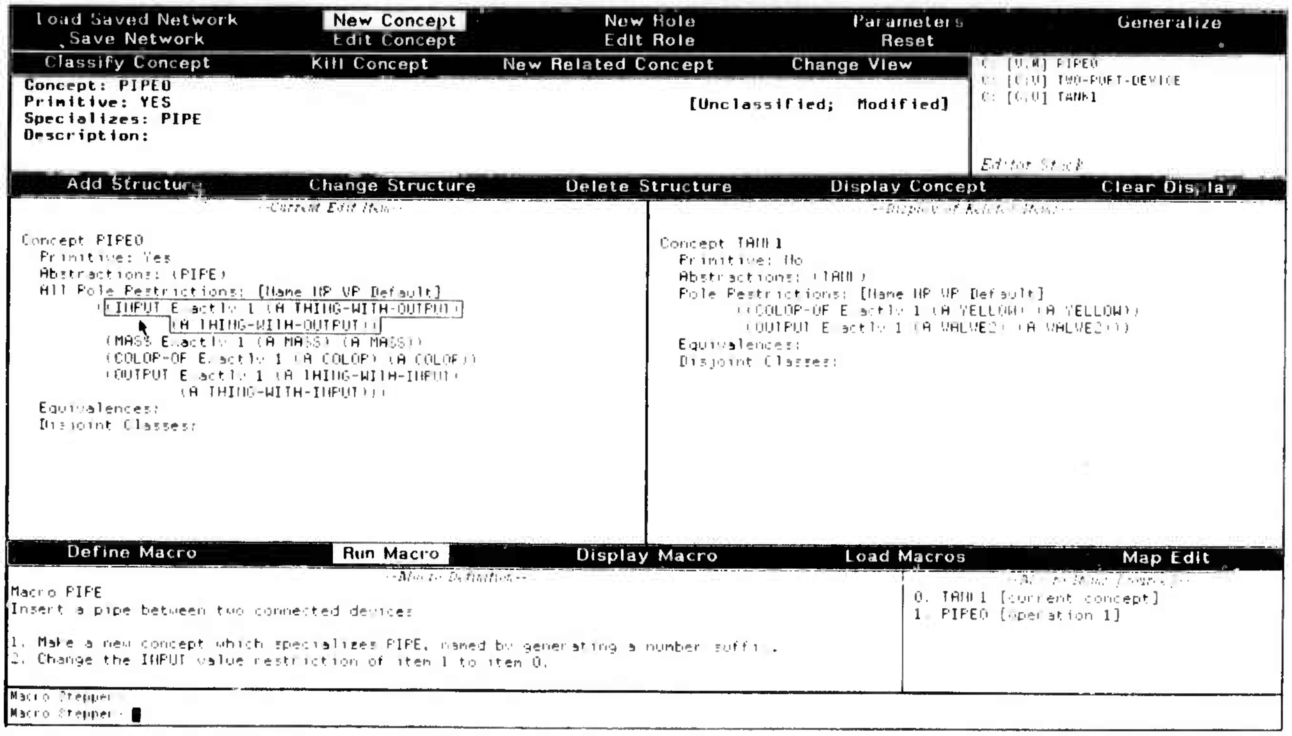
For example, if one was editing the concept ELEPHANT, a command to **Display** the concept that was the value restriction of the role LEGS at that concept would both place ELEPHANT-LEG in the display window and add that concept to the macro items list. Thereafter, all editing commands issued that involve pointing at ELEPHANT-LEG or any part of it are recorded in the macro as operations on the item in the macro item list at the position ELEPHANT-LEG was when the macro was defined. The utility of this form of reference can be made clear with a couple of examples.

4.2.1. Macro Example 1: Adding Pipes

When the STEAMER [20] system was developed, a structural model of a steam plant was created to represent each component in the steam plant as a frame, with links to all functionally related components (e.g., inputs and outputs) represented as slots pointing at those other objects. So, for example, a tank holding water to be fed into a boiler tank through some pipe that was gated by a valve was represented as a frame with an OUTPUT slot whose value was a VALVE. The OUTPUT of that VALVE was a BOILER-TANK. The pipes through which the water was conveyed *were not* represented since they had no functional value in the simulation model.

If it became important to model the pipes, say because they introduced friction or were susceptible to leaks or explosions, then the representational model that STEAMER relied on would have required massive revision. Each component object in the system would have needed editing to replace the objects in its INPUT and OUTPUT slots with new frames representing pipes that were in turn connected by their OUTPUT slots to the next component in the system.

One of our goals in developing the KREME macro editor was to be able to make such changes, which are simple to describe but require many tedious editing operations to accomplish, given the number of concepts affected. In the example below, we show how a macro is defined that can be applied to all objects in a system with OUTPUT slots, in order to generate and insert PIPEs



```
Concept: PIPE0
Primitive: YES
Specializes: PIPE                                   [Unclassified;  Modified]
Description:
```

```
C: [U,M] PIPE0
C: [C,U] TWO-PORT-DEVICE
C: [C,U] TANK1
```

```
Concept PIPE0
  Primitive: Yes
  Abstractions: (PIPE)
  All Role Restrictions: [Name NR VR Default]
        ((INPUT Exactly 1 (A THING-WITH-OUTPUT)
            (A THING-WITH-OUTPUT))
         (MASS Exactly 1 (A MASS) (A MASS))
         (COLOR-OF Exactly 1 (A COLOR) (A COLOR))
         (OUTPUT Exactly 1 (A THING-WITH-INPUT)
              (A THING-WITH-INPUT)))
  Equivalences:
  Disjoint Classes:
```

```
Concept TANK1
  Primitive: No
  Abstractions: (TANK)
  Role Restrictions: [Name NR VR Default]
        ((COLOR-OF Exactly 1 (A YELLOW) (A YELLOW))
         (OUTPUT Exactly 1 (A VALVE2) (A VALVE2)))
  Equivalences:
  Disjoint Classes:
```

```
Macro PIPE
Insert a pipe between two connected devices

1. Make a new concept which specializes PIPE, named by generating a number suffix.
2. Change the INPUT value restriction of item 1 to item 0.
```

```
0. TANK1 [current concept]
1. PIPE0 [operation 1]
```

```
Macro Stepper
Macro Stepper: █
```

Figure 4-2: View when defining the PIPE macro

into those slots The macro also sets the OUTPUTs of those PIPEs to be the concept that was the old value of the OUTPUT slot in the concept edited.

In this example, the macro is defined by editing a simplified representation of a tank (TANK1) connected (by role OUTPUT) to a valve (VALVE2), as shown in figure 4-2. The sequence of steps required is shown in figure 4-3, as they appear in the *Macro Definition* window. (The italic comments in parentheses do not appear in the actual window) Each step describes an editing operation invoked with the appropriate mouse operations, starting with the old definition of TANK1, as shown in the *Current Edit Item* window in figure 4-2. Figure 4-4 shows the state of the editor at the end of this definition process.

The PIPES macro shown here is sufficient to insert concepts representing pipes between concepts with a single OUTPUT and the concepts represented as receiving that output. The macro works as long as the role OUTPUT, or a specialization of that role, exists at the affected concepts.

The current KREME macro and structure editor is still a very preliminary version, and there are still a number of issues to be addressed We are working on the general problem of extending the macro facility so that macros of this type will work when component objects have multiple OUTPUT slots, with different names. What is required is a way to specify that a macro should be applied to *all* such slots.

4.2.2. Example 2: Changing features into concepts

Our second example is of a more common kind of restructuring that occurs when developing frame knowledge bases In developing frame representations, the choice must often be made between giving frames a slot to denote that the concept has some attribute and doing the same thing by defining it as specializing another concept denoting the set of all objects with that attribute Neither option is exclusive, but only one way is typically needed for the purposes of a given application.

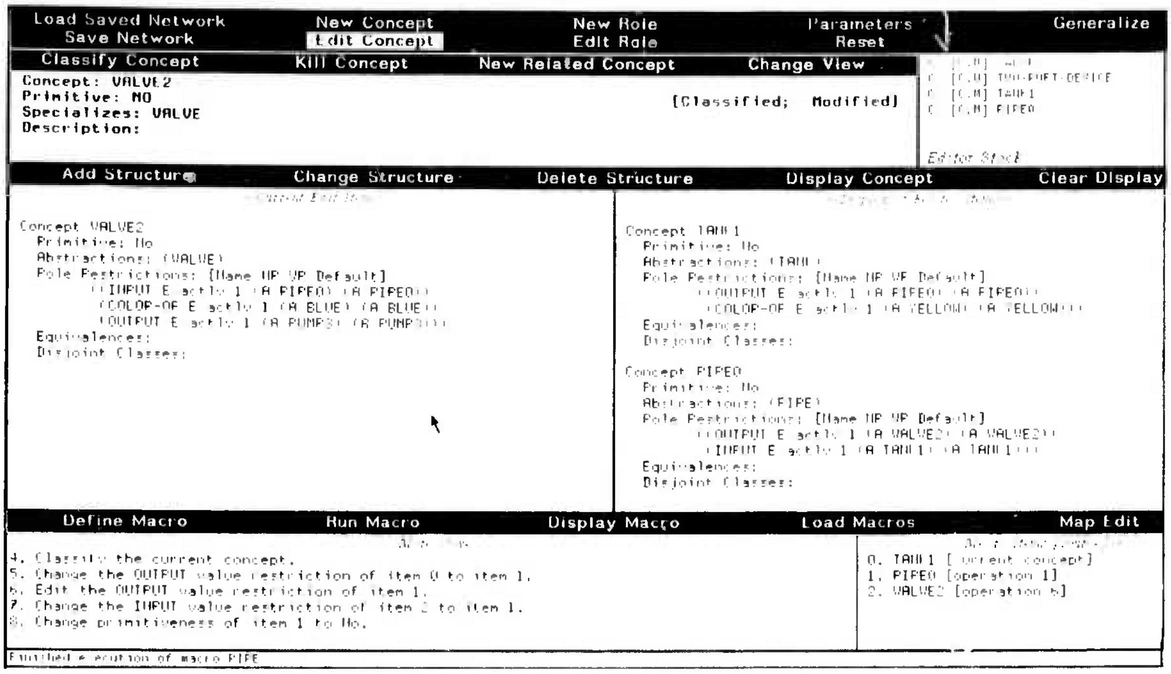
Steps in PIPE macro:

Edit TANK1
Click on **Define Macro** (*Makes Macro Item 0 = TANK1*)

1. Make a new concept which specializes PIPE, named by generating a number suffix (*Creates PIPE0 as item 1, puts it in the current edit item window*)
2. Change the INPUT value restriction of item 1 (*INPUT of PIPE0*) to item 0 (*TANK1*).
3. Change the OUTPUT value restriction of item 1 (*OUTPUT of PIPE0*) to the OUTPUT value restriction of item 0 (*OUTPUT of TANK1 = VALVE1*).
4. Classify the current edit concept (*Defines PIPE0*).
5. Change the OUTPUT value restriction of item 0 (*OUTPUT of TANK1 was VALVE1*) to item 1 (*PIPE0*).
6. Classify item 0 (*TANK1*).
7. Edit the OUTPUT value restriction of item 1 (*Creates item 2 = VALVE1*).
8. Change the INPUT value restriction of item 2 (*INPUT of VALVE1 = TANK1*) to item 1 (*PIPE0*).

End Macro PIPE

Figure 4-3: Steps in PIPE Macro



```
Concept: VALVE2
Primitive: NO
Specializes: VALVE
Description:
                                        [Classified;  Modified]
```

```
C  [C,M] TWO-PORT-DEVICE
C  [C,M] TANK1
C  [C,M] PIPE0
```

```
Concept VALVE2
  Primitive: No
  Abstractions: (VALVE)
  Role Restrictions: [Name NR VR Default]
         ((INPUT Exactly 1 (A PIPE0) (A PIPE0))
          (COLOR-OF Exactly 1 (A BLUE) (A BLUE))
          (OUTPUT Exactly 1 (A PUMP3) (A PUMP3)))
  Equivalences:
  Disjoint Classes:
```

```
Concept TANK1
  Primitive: No
  Abstractions: (TANK)
  Role Restrictions: [Name NR VR Default]
         ((OUTPUT Exactly 1 (A PIPE0) (A PIPE0))
          (COLOR-OF Exactly 1 (A YELLOW) (A YELLOW)))
  Equivalences:
  Disjoint Classes:

Concept PIPE0
  Primitive: No
  Abstractions: (PIPE)
  Role Restrictions: [Name NR VR Default]
         ((OUTPUT Exactly 1 (A VALVE2) (A VALVE2))
          (INPUT Exactly 1 (A TANK1) (A TANK1)))
  Equivalences:
  Disjoint Classes:
```

```
4. Classify the current concept.
5. Change the OUTPUT value restriction of item 0 to item 1.
6. Edit the OUTPUT value restriction of item 1.
7. Change the INPUT value restriction of item 2 to item 1.
8. Change primitiveness of item 1 to No.
```

```
0. TANK1 [current concept]
1. PIPE0 [operation 1]
2. VALVE2 [operation 6]
```

```
Finished execution of macro PIPE
```

Figure 4-4: View after defining the PIPE macro

Steps in COLOR-OBJECTS macro

Edit RED

Click on **Define Macro**
(*Makes Macro Item 0 = RED*).

1. Make a new concept which specializes OBJECT, named by adding as prefix item 0's name (*Creates RED-OBJECT as item 1, puts it in the current edit item window*)

2. Change the COLOR-OF value restriction of item 1 to item 0 (*RED*)

3. Change the primitiveness of item 1 to No

4. Classify item 1. (*This finds all concepts with COLOR-OF slots restricted to RED, and makes them specializations of RED-OBJECT.*)
The remaining steps make these specialization links *defined links*, and remove the COLOR-OF slots completely.

5. Do on SPECIALIZATIONS of item 1. Add item 1 to the parents of iteration item. (*This makes each red object have defined parent RED-OBJECT.*)

6. Do on SPECIALIZATIONS of item 1. Classify iteration item.

7. Change the primitiveness of item 1 to Yes.

8. Delete the COLOR-OF restriction of item 1

9. Do on ALL SPECIALIZATIONS of item 1. Delete the COLOR-OF restriction of iteration item.

10. Classify item 1

Figure 4-5: Changing RED to RED-OBJECT

Quite frequently the choice made early on in the development of a KB proves to be inappropriate, and massive editing is required to convert the accumulated representation base. A macro facility of this type will make these decisions easier to reverse and, therefore, less disruptive and costly in their pragmatic consequences.

We illustrate this kind of restructuring operation with a macro that provides a way of forming a concept RED-OBJECT denoting the set of all objects with the role restriction COLOR = RED, and then removing those COLOR slots. Figure 4-5 shows this macro's steps.

This macro uses the classifier to help make some of the required deductions. First, for a given COLOR, say RED, it defines RED-OBJECT, a non-primitive specialization of OBJECT, with COLOR-OF restricted to RED. Classifying this concept automatically places all other objects with COLOR-OF restricted to RED (or specializations of RED) beneath it in the specialization hierarchy[8], which simplifies the job of defining the macro considerably.

The remaining steps in the macro remove the COLOR-OF restriction from RED-OBJECT and all of its specializations. First, the concepts the classifier found to specialize RED-OBJECT must be given RED-OBJECT as one of their defined parents. RED-OBJECT must also be made primitive before it is reclassified, since it no longer has any defined features to distinguish it from OBJECT.

The steps required to add defined parents to specializations of RED-OBJECT and to remove their COLOR-OF restrictions make use of the KREME MAP-EDIT command. This command is used to perform a single editing operation on a set of concepts related to the one

[8] RED-OBJECT must be marked non-primitive, since it is fully defined by the feature that distinguishes it from OBJECT, its restriction of the COLOR-OF slot to RED. If marked primitive, it would only subsume concepts that defined it as one of their parents.

being edited (e.g., direct specializations, all specializations, abstractions, all abstractions). The limited iteration mechanism provided by MAP-EDIT has proven useful in several macros, and at present we have not found the need to extend the macro language wi'h further control mechanisms.

4.2.3. Future Directions

Work on macro editing has really just begun. However, it already shows promise as a method for accomplishing a number of large scale restructurings of knowledge bases which are relatively simple to describe, but tedious to perform. As example 2 above shows, macros can also make use of the classifier to discover relationships in the knowledge base and exploit them.

At present, the macro editor is only available for editing concepts in the KREME frame language. As the PIPEs example shows, there are still limitations on its capabilities, even there. We are continuing to develop the abilities of the macro editor, and in future will have versions that can be used with the other representation languages that KREME can manipulate. As it stands, the system is already powerful enough to describe a number of transformations between semantically equivalent though functionally and syntactically distinct representations. We are building a library of these operations so that other users of KREME will not be required to reinvent them.

We see our investigation of macro editing as only the first step in developing a knowledge reformulation facility that will have and make use of more understanding of the logical structure of the represented knowledge as well as providing a basic means of describing procedures to manipulate the syntactic structure of knowledge representations. During the second phase of this project, we will be attempting to generalize the functionality provided by this library in a system that is capable of reasoning about the kinds of structural changes the macro editor can perform.

4.3. The Generalizer

One of the tasks faced by knowledge engineers in developing robust computerized knowledge bases is getting experts to express their often unconscious *generalizations* about their domains of expertise. While much of the detailed information about particular problems can be accessed and represented by looking at specific examples and problems, the expert's abstract classification of problem types and the abstract features he uses to recognize those problem types are less readily available.

Experienced knowledge engineers are often able to discover and define useful generalizations that help organize the knowledge described by a human domain expert. The expert, although not previously aware of such a generalization, will often immediately perceive its relevance to and existence within his own reasoning processes, going so far as to suggest improvements, related generalizations, more abstract generalizations and so forth.

An automatic facility for deducing potentially useful generalizations from a network of relatively specific concepts would be an extremely useful capability for a knowledge editing system to provide. An overriding difficulty in building such an engine is the difficulty of establishing criteria for determining what constitutes an "interesting" or useful generalization.

As an initial experiment in automatic generalization within frame taxonomies, KREME provides a relatively simple generalizer algorithm that deals with this difficulty by relying on the user to select from a set of potential generalizations discovered essentially by exhaustive search. Potentially useful generalizations are found by searching for sets of concept features (primarily role restrictions) that are shared by several unrelated concepts. Finding concepts with a given set of features is relatively easy since KREME indexes all concepts under each of its features.

When the generalizer finds a set of at least k features shared by at least m concepts, where k and m are user setable parameters, the system forms the most specific concept definition that would enclose all of the features but would still be more general than any concept in the set. This concept definition is displayed to the user. For example, figure 4-6 shows three concepts that are all ANIMALs and independently define the slot WINGS. Given this, the generalizer would suggest forming a specialization of ANIMAL with the slot WINGS that these concepts would all specialize. If the user wanted to introduce this concept, he would respond by naming the new generalization, which is then classified and inserted into the network. The features that are enclosed by this new, more general concept, are removed automatically from each of the more specific concepts being generalized. Figure 4-7 shows the result with a new concept named FLYING-ANIMAL.

As one might imagine, the generalizer algorithm is fairly slow (taking about 8 minutes to go through a network of 500 concepts and 300 roles). It must look at a fair percentage of all the possible combinations of features in the network. Consequently, we have designed the algorithm to run in a low priority background process, looking for generalizations only when the editor is waiting for input from the user.

As yet, the effectiveness of this generalizer remains substantially untested. We have used tried it on the two reasonably large taxonomies that we have available, and it finds several potential generalizations in each, but the real test must wait until there are new applications under development using the KREME environment. The taxonomies that we have available currently have been carefully developed over long periods of time, and have

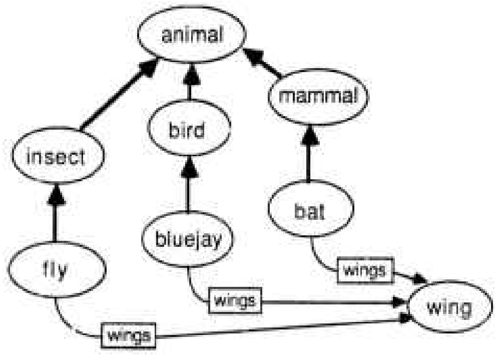


Figure 4-6: Find a Generalization

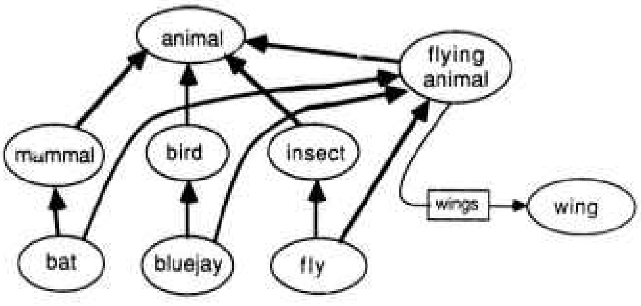


Figure 4-7: After Generalization Added

few remaining "holes".

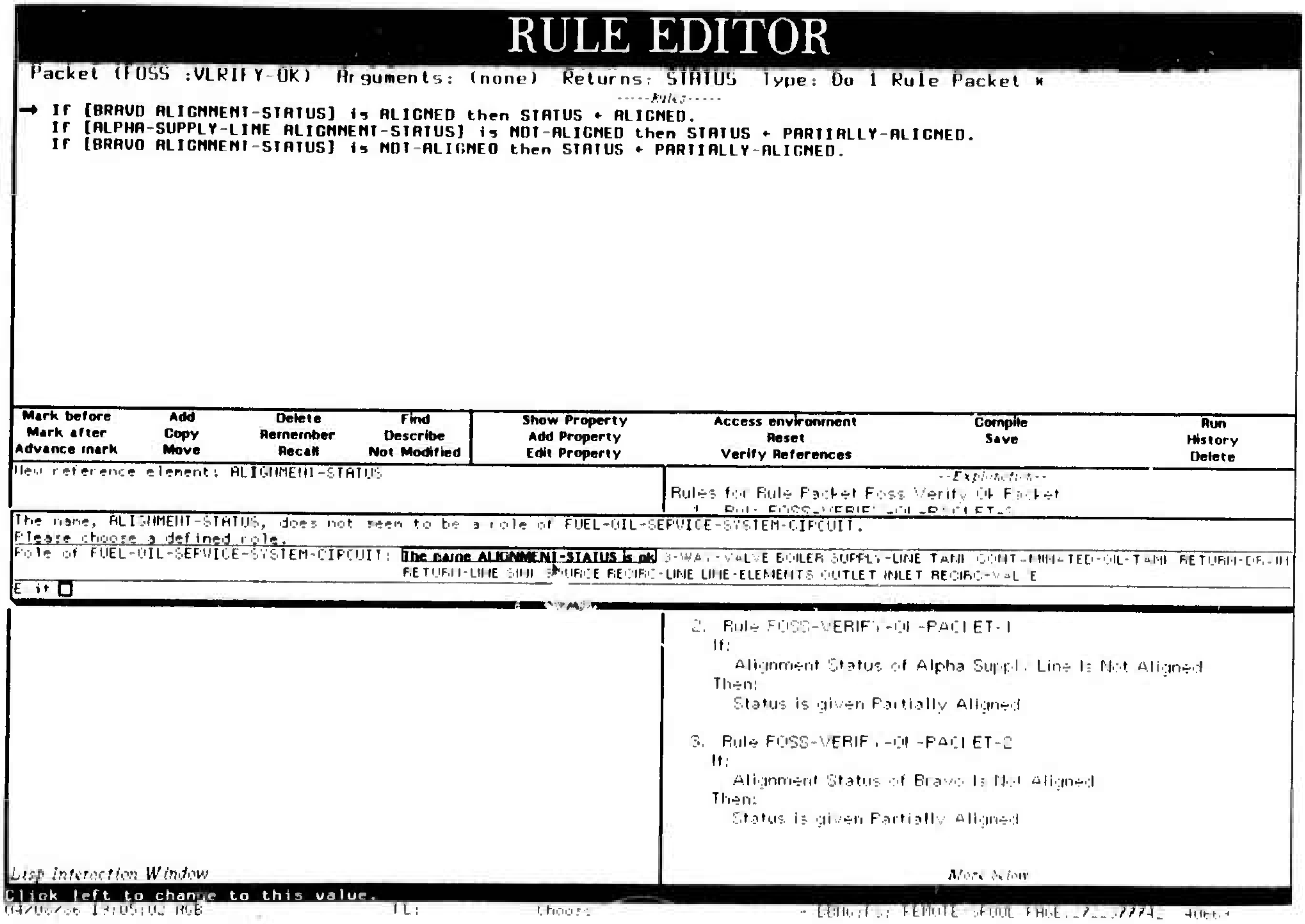
We are also considering developing another version of this generalizer that would attempt to find new concepts in sets of conditions repeatedly appearing as parts of rules. Introducing such concepts could conceivably simplify, and reveal more of the structure of the reasoning involved in rule sets. It might also make extending such rule sets easier. A generalizer of this type will be investigated during phase two of the project.

5. Editing Rules in the KREME Environment

We are in the process of incorporating into the KREME environment an editor for rules written in the FLEX rule language [16]. FLEX is similar to the rule-based portion of the LOOPS language and currently runs on a Symbolics 3600. FLEX provides rule packets, and rule objects. Rule packets provide a way to organize rules. Rule packets can be invoked like functions, with arguments and local variables, and return values via the ZETALISP multiple-values mechanism. Flex incorporates a mechanism for dealing with uncertainty, based on that in EMYCIN [18]. The system also provides an elementary history and tracing mechanism, and an explanation system that produces pseudo-English explanations from rule traces.

The forward chaining rule packets come in four varieties, indicating the type of control mechanism used for rule firing.

- o **do-1-rule-packets** execute the first rule whose test succeeds.
- o **do-all-rule-packets** execute all rules whose tests succeed.



```
Packet (FOSS :VERIFY-OK)  Arguments: (none)  Returns: STATUS  Type: Do 1 Rule Packet *

→ If [BRAVO ALIGNMENT-STATUS] is ALIGNED then STATUS ← ALIGNED.
  If [ALPHA-SUPPLY-LINE ALIGNMENT-STATUS] is NOT-ALIGNED then STATUS ← PARTIALLY-ALIGNED.
  If [BRAVO ALIGNMENT-STATUS] is NOT-ALIGNED then STATUS ← PARTIALLY-ALIGNED.

New reference element: ALIGNMENT-STATUS

The name, ALIGNMENT-STATUS, does not seem to be a role of FUEL-OIL-SERVICE-SYSTEM-CIRCUIT.
Please choose a defined role.
Role of FUEL-OIL-SERVICE-SYSTEM-CIRCUIT: The name ALIGNMENT-STATUS is ok 3-WAY-VALVE BOILER SUPPLY-LINE TANK CONTAMINATED-OIL-TANK RETURN-OR-IN
                                         RETURN-LINE SINK SOURCE RECIRC-LINE LINE-ELEMENTS OUTLET INLET RECIRC-VALVE

Rules for Rule Packet Foss Verify Ok Packet
  1. Rule FOSS-VERIFY-OK-PACKET-3

  2. Rule FOSS-VERIFY-OK-PACKET-1
     If:
       Alignment Status of Alpha Supply Line Is Not Aligned
     Then:
       Status is given Partially Aligned

  3. Rule FOSS-VERIFY-OK-PACKET-2
     If:
       Alignment Status of Bravo Is Not Aligned
     Then:
       Status is given Partially Aligned

Click left to change to this value.
04/06/86 13:05:02 RGB          FL:          Choose          ← LOGIN: REMOTE SPOOL PAGE 172277742   40664
```

Figure 5-1: The FLEX Rule Editor

- o **while-1-rule-packets** repeatedly test all rules, firing one, until no tests succeed.
- o **while-all-rule-packets** repeatedly fires all rules whose tests succeed, until none succeed

An important feature of FLEX is the capability to compile rules into a lower level language, and run without the rule interpreter present. For example, forward chaining rule packets can now be compiled directly into LISP functions. This compiling can be handled by a separate code generator or translator which can produce code for other languages.

Rule packets in FLEX can be connected to KREME frame systems or other data contexts by specifying an *access environment*. An access environment is an object that receives messages dealing with the accessing of values for references in the rules. It handles all messages to get or set the values of variables and their confidences. Flex uses the notion of *paths*. These are composed references. Flex sends the access environment messages to resolve paths that it encounters in rules. When connected to KREME frame hierarchies, these paths describe role or slot chains, as in role equivalences.

5.1. The FLEX Rule Editor

The original FLEX rule editor, shown in figure 5-1, was a predecessor of the KREME structure editor, in terms of its functionality and style of interaction. Thus, its functionality closely resembles that for the frame editor described above. One defines and edits rules by specifying and filling out portions of rule *templates*. The user refines these templates either by using the mouse to copy parts of existing rules or by pointing at slots to be filled and typing in the desired values. Once a rule-set has been developed, the FLEX editor provides commands to run packets and debug them. It can also generate traces or rule histories paraphrased in pseudo-English. Mechanisms are also provided for deleting and reordering rules, and loading and saving them from files.

5.2. Interactions with the Frame editor

Although FLEX was originally designed as a stand-alone system, packages of rules can now be written that refer to instances of KREME Frames using the KREME Frames-ACCESS-ENVIRONMENT. This access environment provides the interface functions necessary for FLEX rules to refer to KREME frame instances, and their slots. It also allows one to write rule packets that serve as methods on frames.

The KREME access environment allows the FLEX rule editor to validate references (paths) to slots in KREME frames when building and debugging rules. When an unresolvable reference is encountered, the invalid portion of the path is pinpointed and a menu of possible actions to fix it is offered to the user. The options at this point include switching to a KREME view in which the suspect concept or role can be edited, defining new concepts, changing the invalid path element, and changing the root element of the path.

We are still in the process of integrating this rule system into the KREME world. In the near future, it will also be possible to associate rule packets with concepts, and browse or edit those packets from within the KREME editing environment.

6. Editing Procedures in the KREME Environment

6.1. The KREME Procedure language

6.1.1. Background

An obvious weakness of many knowledge representation languages is their inability to handle declaratively expressed knowledge about procedures as partially ordered sequences of actions, particularly if that knowledge is represented at multiple levels of abstraction. Although a number of systems have been developed that do various forms of planning, [4, 12, 13, 17], most have not encoded their plans in an entirely declarative or inspectable fashion. Certainly the current generation of expert system tools does not provide for the description of this kind of knowledge. Although it is clear that much of an expert's knowledge about a domain is about procedures and their application, little work has been done on devising ways to capture that information directly.

The STEAMER project began to address the issue of declarative representations for procedures in the course of developing a mechanism to teach valid steam plant operating procedures. The representation system developed for this task had to be directly accessible to the students who were the system's users, and it had to serve as a source of explanations when errors were made. STEAMER was able to describe these procedures, decompose them, show how they were related to similar procedures and, in general, deal with them at the "knowledge level" [10] rather than as pieces of programs or rule sets. Although the syntax of the language was quite primitive, with no provisions for branching or iteration, the mechanisms for procedural abstraction, specialization and path or reference reformulation that formed the heart of the language seemed to form the kernel of an extremely useful representational facility.

The STEAMER procedure language was well integrated with the MSG frame language that was one of the starting points for KREME Frames, and minimal effort was necessary to incorporate a very similar language into KREME. We refer to the results of this effort as KREME Procedures. We expect to expand the KREME Procedures language, and provide much improved editing facilities for procedures in the near future.

6.1.2. Basic syntax

A *procedure* consists of a its *name*, its *description*, the *action* that the procedure is meant to accomplish, a list of *steps*, and a list of *ordering constraints* that determine the partial ordering of the steps. Procedures are attached to specific frames (concepts).

A *step* consists of an *action* and a *path*. The path (as in role equivalences) refers to a particular concept which is said to be the *object* of the step. For example, a concept called SUCTION-LINE might have a slot for a part named PUMP, which is restricted to being a CENTRIFUGAL-PUMP. We might define a procedure for ALIGNing the SUCTION-LINE which would have a step to OPEN the DISCHARGE-VALVE of the PUMP. This would be expressed in step form as OPEN (PUMP DISCHARGE-VALVE) and would indicate a step that opened the discharge valve of the centrifugal pump which was the pump of the suction line.

A *constraint* is an ordering between two steps (the before step and the after step). Each constraint is supported by a *principle*. A principle consists of its name, a description of its rationale and a numeric priority.

Each step in a procedure may either be a primitive action or another procedure. If the object of a step defines a procedure for the action of that step then this procedure is said to be a sub-procedure of the enclosing procedure. Using our example from above, the ALIGN procedure attached to the concept SUCTION-LINE could have a step ALIGN <PUMP>. If the concept CENTRIFUGAL-PUMP, which is the object of this step in ALIGN<SUCTION-LINE>, defined a procedure for the action ALIGN, then the step ALIGN <PUMP> could be expanded into the steps of the procedure for aligning a centrifugal pump.

6.1.3. Procedural abstraction and structure mapping

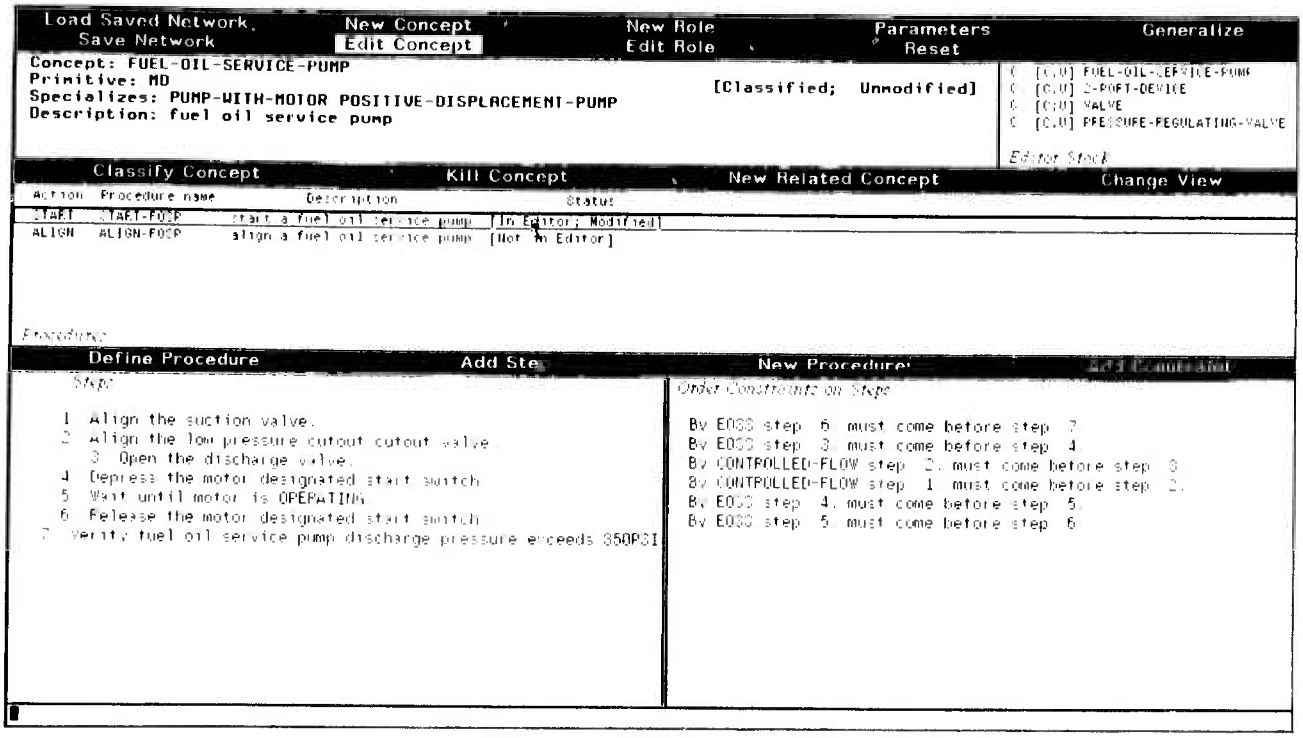
For knowledge acquisition purposes, it would be very useful if procedures were represented in an abstraction hierarchy like that for concepts. In a strong sense, saying that one abstract procedure subsumes another seems infeasible. However much power can be gained if abstract procedures form templates upon which more specific procedures can be built, much as was done in NOAH [13]. For example, if you have some idea about how to grow plants in general, and you want to grow tomatoes, you will use your knowledge about growing plants in general as a starting point for learning about growing tomatoes. The final procedure for growing tomatoes will include some (presumably more detailed) versions of steps in the more general procedure, and may also include steps that are analogous to those used in growing other plants for which more detailed knowledge exists.[9] KREME Procedures has a mechanism for building templates of new procedures out of abstract procedures. When a new procedure is being defined at a concept, the procedural abstraction function determines whether any of that concept's parents have a procedure for accomplishing the same action. If one or more do, the new procedure organizes the steps and their ordering constraints, with suitably reconstructed paths, to form a template on which the new procedure can be built. As yet this facility does not have the ability to do detailed reasoning with constraints on steps, as NOAH does. We expect to greatly expand this capability during phase two of the project.

6.2. The Procedure Editor

When procedures are attached to particular concepts, a **procedure editing view** is one of the views available for that concept. In this view, the editor displays a list of all of the existing KREME Procedures for the current concept (See figure 6-1.) When the procedures view is visible, the user can choose to delete any existing procedure, edit a procedure or create a new procedure. Several procedures can be edited simultaneously, with the topmost procedure in the procedure list window being the current, visible procedure.

The current procedure (of the current concept) has its steps and ordering constraints displayed. Steps and constraints can be added to or deleted from the current procedure. Editing of the current procedure can be interrupted by the user choosing another procedure to edit, switching views for the controlling concept or interrupting the edit of the controlling concept.

When the user is satisfied with the definition of a procedure he has edited, it is ready to be inserted into the knowledge base. The **Define Procedure** command accomplishes this by first ordering the procedure's steps based on their ordering constraints. If the constraints are contradictory, the user must resolve the contradiction by eliminating constraints or by making some constraints higher priority than others. Next, a



Concept: FUEL-OIL-SERVICE-PUMP
Primitive: MD
Specializes: PUMP-WITH-MOTOR POSITIVE-DISPLACEMENT-PUMP
Description: fuel oil service pump

[Classified; Unmodified]

C [C,U] FUEL-OIL-SERVICE-PUMP
C [C,U] 2-PORT-DEVICE
C [C,U] VALVE
C [C,U] PRESSURE-REGULATING-VALVE

```
Action  Procedure name  Description                 Status
START   START-FOSP      start a fuel oil service pump  [In Editor; Modified]
ALIGN   ALIGN-FOSP      align a fuel oil service pump  [Not In Editor]
```

```
  1  Align the suction valve.
  2  Align the low pressure cutout cutout valve.
  3  Open the discharge valve.
  4  Depress the motor designated start switch
  5  Wait until motor is OPERATING
  6  Release the motor designated start switch
  7  Verify fuel oil service pump discharge pressure exceeds 350PSI
```

```
By EOSS step 6 must come before step 7
By EOSS step 3 must come before step 4
By CONTROLLED-FLOW step 2 must come before step 3
By CONTROLLED-FLOW step 1 must come before step 2
By EOSS step 4 must come before step 5
By EOSS step 5 must come before step 6
```

Figure 6-1: The Procedure Editor

[9] For a detailed discussion of related issues see Carbonell [3] on derivational analogical planning.

procedure object is made and associated with the classified version of the concept in the knowledge base. The procedure may also be compiled into a flavor method that becomes part of the behavior associated with the concept. After a procedure has been installed, the procedure editor redisplays the procedure steps, showing them in their proper partial order.

Clicking on a step that is itself a procedure causes the editor to replace the step with the steps of that procedure, adjusting the paths of the expanded steps and adding appropriate constraints so that the expansion falls logically between the steps surrounding its unexpanded form in the original procedure.

A step expansion can be closed by clicking on any of the expanded steps. The editor simply replaces the expanded set of steps with the original step and adjusts constraints accordingly. Expanded steps are made permanent when the **Define Procedure** command is invoked.

A new procedure is entered by typing the procedure name, description and action. The procedure editor checks to see if any of the parents of the controlling concept have procedures for the same action. If so, an initial procedure template is built by combining the steps and constraints of all the inherited, more abstract procedures. The paths of the steps are adjusted to use "local" slot names, as much as possible, using the concept's role equivalences as described in section 6. The procedure definition object thus formed is then displayed for editing.

The KREME Procedures language is currently being refined for use in a new training system under development at BBN. That system will teach diagnostic procedures for the maintenance of a large electronics system. We expect that KREME will greatly ease the knowledge acquisition problems faced by the developers of that system. It will also provide the first serious test of the effectiveness of the KREME acquisition environment in general.

7. Conclusion

The goal of the BBN Labs Knowledge Acquisition Project is to build a versatile experimental computer environment for developing the large knowledge bases which future expert systems will require. We are pursuing this goal along two complementary paths. First, we have constructed a flexible, extensible, Knowledge Representation, Editing and Modeling Environment in which different kinds of representations (initially frames, rules, and procedures) can be used, and we can investigate the acquisition strategies for a variety of types of knowledge representations. In building and equipping this "sandbox", we are adapting and experimenting with techniques which we think will make editing, browsing, and consistency checking for each style of representation easier and more efficient, so that knowledge engineers and subject matter experts can work together to build with significantly larger and more detailed knowledge bases than are presently practical.

Now that we are well along in constructing a first, experimental version of the editing environment, we are beginning to address the second aspect of our research plan, the development of more automatic tools for knowledge base reformulation and extension. An important part of this endeavor is the discovery, categorization and use of explicit knowledge about knowledge representations, methods for viewing different knowledge representations, techniques for describing knowledge base transformations and extrapolations, techniques for finding and suggesting useful generalizations in developing knowledge bases, semi-automatic procedures for of eliciting knowledge from experts, and extensions of consistency checking techniques to provide a mechanism for generating candidate expansions of a knowledge base.

Our ultimate goal is to explore a number of approaches to knowledge acquisition and knowledge editing that could be incorporated into existing and future development environments, not to develop the definitive knowledge editing environment. AI is still a young field, and new knowledge representation techniques will continue to be developed for the foreseeable future. We are attempting to provide a laboratory for experimenting with new representation techniques and new tools for developing knowledge bases. If we are successful, many of the techniques developed in our laboratory will be adopted by the comprehensive knowledge acquisition and knowledge representation systems required to support the development and maintenance of AI systems in the future.

ACKNOWLEDGEMENTS We would like to thank the other members of the BBN Labs Knowledge Acquisition Project, Richard Shapiro and Albert Boulanger. Rich and Albert jointly developed the FLEX rule system and editor, and Rich was also largely responsible for the implementation of the Macro and Structure Editor. Dr. Ed Walker read and substantially edited several draf s of this paper.

References

[1] Balzac, Stephen R.
A System for the Interactive Classification of Knowledge.
Technical Report M.S. Thesis, M.I.T. Dept of E.E. and C.S., 1986.

[2] Brachman, R.J., Fikes, R.E., and Levesque, H.J.
Krypton. A Functional Approach to Knowledge Representation.
IEEE Computer, Special Issue on Knowledge Representation , October, 1983.

[3] Carbonell, Jaime G.
Derivational Analogy. A theory of reconstructive problem solving and expertise acquisition.
In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (editor), *Machine Learning, Volume II*, pages 371-392.Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[4] Ernst, G.W. and Newell, A.
GPS. A Case Study in Generality and Problem Solving.
Academic Press, New York, 1969.

[5] IntelliCorp.
KEE Software Development System.
IntelliCorp, 1984.

[6] Keene, Sonya E. and Moon, David.
Flavors. Object-oriented Programming on Symbolics Computers.
Symbolics, Inc.
1985.

[7] Carnegie Group, Inc.
KnowledgeCraft.
Carnegie Group, Inc., 1985.

[8] McAllester, D. A.
Reasoning Utility Package User's Monuol.
Technical Report AI Memo 667, M.I.T. A.I. Laboratory, April, 1983.

[9] Moser, Margaret.
An Overview of NIKL.
Technical Report Section of BBN Report No. 5421, Bolt Beranek and Newman Inc., 1983.

[10] Newell, A.
The knowledge level.
AI Magazine 2(2).1-20, 1981.

[11] Rich, C.
Knowledge Representation Languages and Predicate Calculus. How to Have Your Cake and Eat It Too.
In *Proc. AAAI*, pages 192-196. 1982.

[12] Sacerdoti, E. E.
Planning in a Hierarchy of Abstraction Spaces.
Artificial Intelligence 5(2).115-135, 1974.

[13] Sacerdoti, Earl D.
A structure for plans and behavior.
Technical Report 109, SRI Artificial Intelligence Center, 1975.

[14] Schmolze, J. and Israel, D.
KL-ONE. Semantics and Classification.
In *Research in Knowlege Representation for Natural Language Understanding, Annual Report. 1 September 1982 to 31 August 1983*.BBN Report No. 5421, 1983.

[15] Schmolze, J.G., Lipkis, T.A.
Classification in the KL-ONE Knowledge Representation System.
In *Proc. 8th IJCAI.* 1983.

[16] Shapiro, Richard.
FLEX. A Tool for Rule-based Programming.
Technical Report 5643, BBN Labs., 1984.

[17] Stefik, Mark.
Planning with Constraints. MOLGEN.
Artificial Intelligence 16(2).111-169, 1981.

[18] van Melle, W.
A domain independent production-rule system for consultation programs.
In *Proceedings of IJCAI-6*, pages 923-925. August 1979.

[19] Vilain, Marc.
The Restricted Language Architecture of a Hybrid Representation System.
In *Proceedings, IJCAI-85*, pages 547-551. International Joint Conferences on Artificial Intelligence, Inc., August, 1985.

[20] Williams, M., Hollan, J., and Stevens, A.
An Overview of STEAMER. An Advanced Computer-Assisted Instruction System for Propulsion Engineering.
Behavior Research Methods and Instrumentation 14.85-90, 1981.

Experimental Knowledge Systems Laboratory Progress Report on Reasoning Under Uncertainty

University of Massachusetts
Amherst, Mass, 01003

1. Introduction

This paper describes four projects to develop techniques for reasoning under uncertainty in knowledge systems. The work is based on the premise that knowledge about sources of uncertainty and evidence should be represented explicitly, so that knowledge systems can reason about their uncertainty. This position raises many questions: How should knowledge about uncertainty be represented? what aspects of uncertain situations should be explicit? How should evidence be combined? How can a system minimize its uncertainty? How are decisions taken under uncertainty? These and other questions are the foci of the four research efforts described here. One project has resulted in an architecture for planning medical consultations, that is, determining appropriate questions, tests, and treatments given previous results during the consultation. The goal of the project is to integrate current research on explicit, sophisticated control with explicit reasoning about uncertainty: the causes of uncertainty and characteristics of evidence effect control decisions. A second project shares this concern for control: we have developed a general method for constructing decisions under uncertainty. By classifying decision-making situations, one can "read off" actions that will transform uncertain decisions into more tractable ones. This opens the possibility of sophisticated control by table lookup. The third and fourth projects focus on the representation of uncertainty. One proposes a model for reasoning about the uncertainty inherent in semantic matching problems. The other extends this work to a view of common sense inference as "generalized syllogisms" over an associative knowledge base.

This report is taken from three recent papers: "Managing Uncertainty in Medicine" by Paul Cohen, David Day, Jeff Delisio, Mike Greenberg, Rick Kjeldsen, and Paul Berman, M.D.; "A Typology for Constructing Decisions" by Adele Howe and Paul Cohen; "Classification by Semantic Matching" by Paul Cohen, Philip Stanhope, and Rick Kjeldsen. The section on plausible inference was written by Paul Cohen and David Lewis.

2. Management of Uncertainty in Medicine

2.1 Introduction

MUM is a knowledge-based consultation system designed to manage the uncertainty inherent in medical diagnosis (the acronym stands for Management of Uncertainty in Medicine). Managing uncertainty means planning actions to minimize uncertainty or its consequences. Thus it is a control problem – an issue for the component of a knowledge system that decides how to proceed from an uncertain state of a problem. Uncertainty can be managed by many strategies, depending on the kind of problem one is trying to solve. These may include asking for evidence, hedging one's bets, deciding arbitrarily and backtracking on failure, diversification or risk-sharing, and worst-case analysis. The facility with which a consultation system such as MUM manages uncertainty is evident in the questions it asks: it should ask all necessary questions, no unnecessary questions, and it should ask its questions in the right order. These conditions, especially the last one, preclude uniform and inflexible control strategies. They prompted the development of the MUM architecture in which control decisions are taken by reasoning about features of evidence and sources of uncertainty.

2.2 The Goals of MUM

MUM diagnoses chest pain and abdominal pain. This includes taking a history, asking for physical findings, ordering tests, and prescribing trial therapy. Physicians call a diagnostic sequence of questions and tests a *workup*. MUM's *primary* goal is to generate workups for chest and abdominal diseases that include, in the correct order, all necessary questions and tests and none that are superfluous. Since we built MUM to study the management of uncertainty, the goal of correct diagnosis is secondary to generating the correct workup. We were influenced by a distinction physicians make between *retrospective* diagnosis, in which all evidence is known in advance and the goal is to make a correct diagnosis, and *prospective diagnosis*, which emphasizes the workup and proper management of the patient, even under uncertainty about his or her condition. MUM is definitely prospective. Figure 1 illustrates part of the workup for coronary artery disease. Clearly, we could build a system that follows this and other stored workups, but the point of the research is to be able to reason about the features of evidence, and the uncertainty in partially-developed diagnoses, to decide which questions to ask next. If MUM does this properly then its questioning will correspond with a standard workup, or at least be a reasonable alternative workup.

2.2.1 Managing Uncertainty and Control

MUM is based on the idea that managing uncertainty and controlling a complex knowledge system are manifestations of a single task, namely, acquiring evidence and using it to solve problems. There would be little basis for variation in problem-solving strategies if all evidence was equally costly, reliable, available, and pertinent; but if available and attainable evidence is differentiated along these and other dimensions, then problem-solving can be guided by the ideal of maximum evidence for minimum cost. For example, here is a strategy for focusing attention on available evidence:

```
CONTEXT:        to minimize cost

CONDITIONS:     $test_1$ and $test_2$ are pertinent, and
                $test_1$ is potentially-confirming, and
                $test_2$ is potentially-supporting, and
                $\mathrm{cost}(test_1) >> \mathrm{cost}(test_2)$

ACTIONS:        begin
                do $test_2$
                if supporting then do $test_1$
                        else do not do $test_1$
                end
```

That is, given cheap, weak evidence and expensive, strong evidence, get the weak evidence first and don't incur the cost of the strong evidence unless the weak evidence lends support. The rule serves to manage the uncertainty associated with the weak evidence – it says seek strong corroboration only if the weak evidence is positive. It also uses features of evidence such as cost and reliability to control the acquisition of evidence; for example, it explains why an angiogram (an expensive, risky, and excruciating test) is done only after a stress test in Figure 1. We distinguish these functions – managing uncertainty and control – only because uncertainty and control have, with a few exceptions noted below, been viewed as different topics. In fact, if control decisions are based on features of evidence, then control and managing uncertainty are the same thing. This is the principle that motivates the design of MUM discussed in Section 2.3.3.

2.2.2 Related Work

The close association between control and managing uncertainty has been apparent in the literature on sophisticated control for several years [1] but is largely absent from the AI literature on reasoning under uncertainty. Three important results have emerged from research on control: First, complex and uncertain problems must be solved *opportunistically* and *asynchronously* – working on subproblems in an order dictated by the availability and quality of evidence (Hayes-Roth and Lesser, 1977). Second, since control tends to be accomplished by *local* decisions about focus of attention, the behavior of complex knowledge systems sometimes lacks global coherence. Coherence can be achieved by *planning* sequences of actions instead of selecting individual actions by local criteria[2]. Third, programs are impossible to understand if the factors that affect control decisions are *implicit*. For example, the focus of attention in Hearsay-II was difficult to follow because it depended on many numerical parameters calculated from data and combined by empirical functions with "tuning" parameters (Hayes-Roth and Lesser, 1977). A better approach is to explicitly state and reason about the implicit factors, called *control parameters* (Wesley, 1983), that the numbers represent (Davis, 1985; Clancey, 1983). If the control parameters are features of evidence and uncertainty, then control strategies can be developed to manage uncertainty.

This last point colors our reading of the AI literature on reasoning under uncertainty. Much of it is concerned with the mathematics of combining evidence, the calculation of *degrees of belief* in hypotheses. (A representative sample includes Shortliffe and Buchanan, 1975; Duda, Hart, and Nilsson, 1976; Zadeh, 1975; Shafer, 1976. See Cohen and Gruber, 1985; and Bonissone, 1985, for literature reviews, including nonnumeric approaches to uncertainty; and Szolovits and Pauker, 1978 for a discussion of uncertainty in medicine.) Degrees of belief *can* serve as control parameters, but it is necessary to maintain a distinction between combining evidence and control. Otherwise, degrees of belief (and

[1] For example, the classic paper by Erman, Hayes-Roth, Lesser, and Reddy (1980) is called "The Hearsay-II speech understanding system: Integrating knowledge to resolve uncertainty."

[2] Personal communication, Victor Lesser.

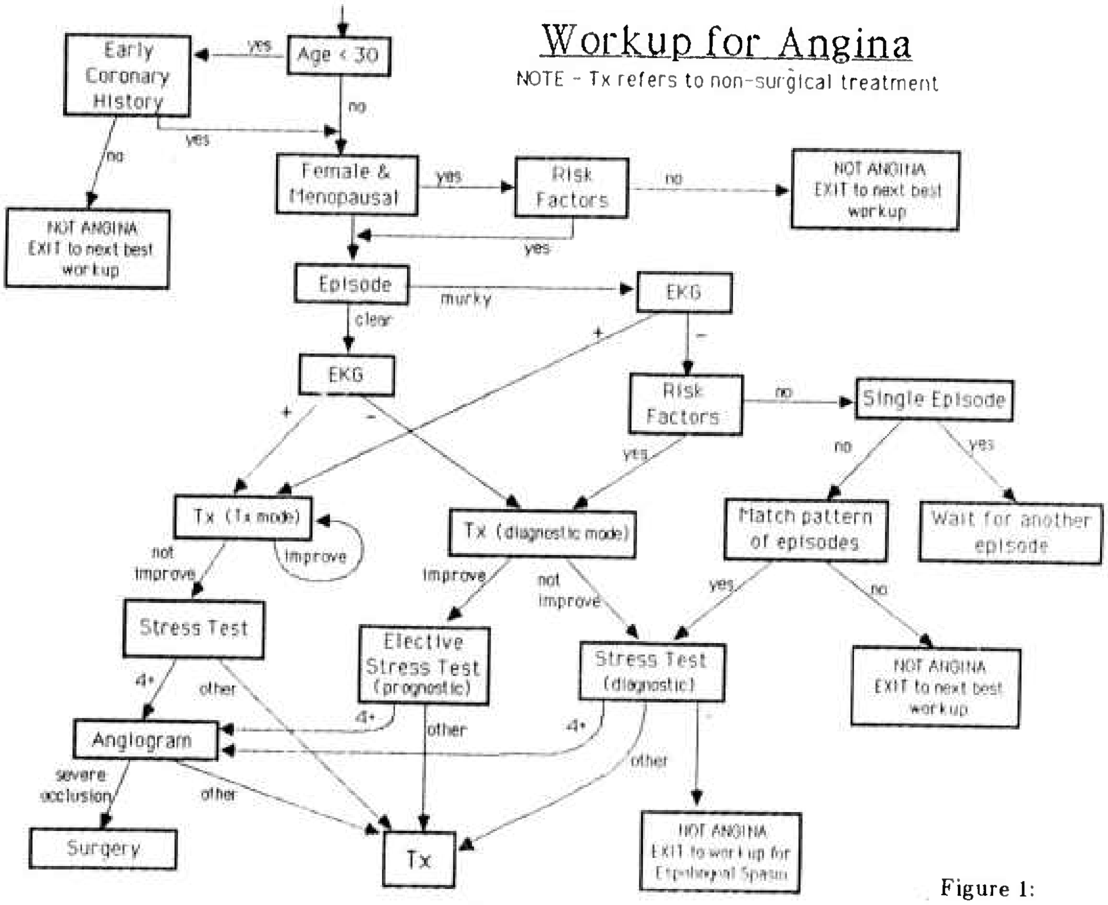


Figure 1:

the functions that combine them) have to be "tuned" not only to find the most likely answer but also to focus attention in a reasonable way. Inevitably they become ambiguous summaries of implicit control parameters. For example, MYCIN's certainty factors contained probabilistic and salience information, an indirect result of using them to focus attention (Buchanan and Shortliffe, 1985).

Another important reason to maintain the distinction between combining evidence and control is that combining evidence is only a part of the problem of reasoning under uncertainty. Other aspects include formulating decisions, assessing the need for more evidence, planning how to get it, deciding whether it is worth the cost and, if it isn't, hedging against residual uncertainty. In MUM we address the problem of combining uncertainty in the context of these other tasks.

2.3 An Architecture for Managing Uncertainty

Managing uncertainty in MUM requires many kinds of knowledge, discussed in this section. Anticipating section 2.3, on control, it may be useful to think of data moving bottom-up through Figure 2 as it triggers hypotheses and is requested by MUM's planner.

2.3.1 Types of Knowledge

Data, Evidence, and Interpretation Functions. Evidence is abstracted from data through interpretation functions. All data about a patient are stored in frames that describe personal history, family history, tests, history of episodes, and other data. Interpretation functions map data to evidence; for example, information that a patient smokes 3 packs of cigarettes a day is abstracted to the evidence *heavy-smoker* by an interpretation function that maps data about smoking habits to one of *(non-smoker light-smoker moderate-smoker heavy-smoker)*. Interpretation functions are of-

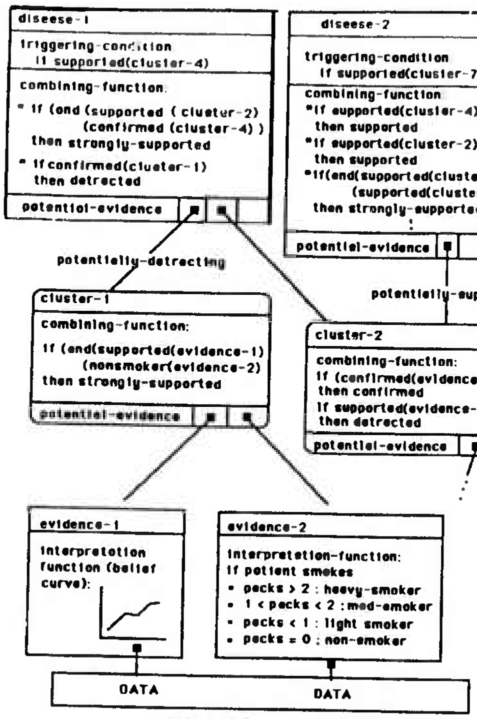


FIGURE 2: KNOWLEDGE STRUCTURES IN MUM

ten graphs called *belief curves* that relate ranges of a continuous data variable to belief in evidence. Figure 3 shows a belief curve relating the duration of chest pain to the evidence *classic-anginal-pain*. Belief curves and other interpretation functions are acquired from an expert. They provide the same functionality as fuzzy predicates (Zadeh, 1975), and generalize Clancey's view of data abstraction as categorical (Clancey, 1983).

Features of Evidence. Evidence may be characterized by its cost, reliability, and roles. The cost of evidence reflects monetary cost as well as discomfort and risk to the patient (later versions of MUM will separate these and other determinants of cost). Reliability refers to several factors, including false-positive and miss rates of tests, and also the belief in evidence derived from belief curves (e.g., is *classic-anginal-pain* at least *supported* by data about the pain duration?) The most important feature of evidence is the *roles* it can play with respect to evaluating hypotheses. MUM recognizes five roles, two of which are symmetric pairs:

Potentially-confirming and potentially-discon firming. If evidence plays a potentially-confirming role with respect to a hypothesis, then acquiring it *might* confirm the hypothesis, though not all potentially-confirming evidence will, in actuality, confirm. For example, an EKG confirms the hypothesis of angina only if "positive" (i.e., shows ischemic changes.) Once confirmed (or disconfirmed), a hypothesis requires no further evidence, though a diagnostician may continue working to disconfirm other hypotheses, especially if they are dangerous.

Potentially-supporting and potentially-detracting. Like *potentially-confirming* and *potentially-disconfirming*, but not conclusive. However, combinations of supporting or detracting evidence may be confirming and disconfirming, respectively (see "Combining Func tions," below). The combination referred to as cluster-2 (Fig. 2) is *potentially-supporting* with respect to disease-2; cluster-1 is *potentially-detracting* with respect to disease-1.

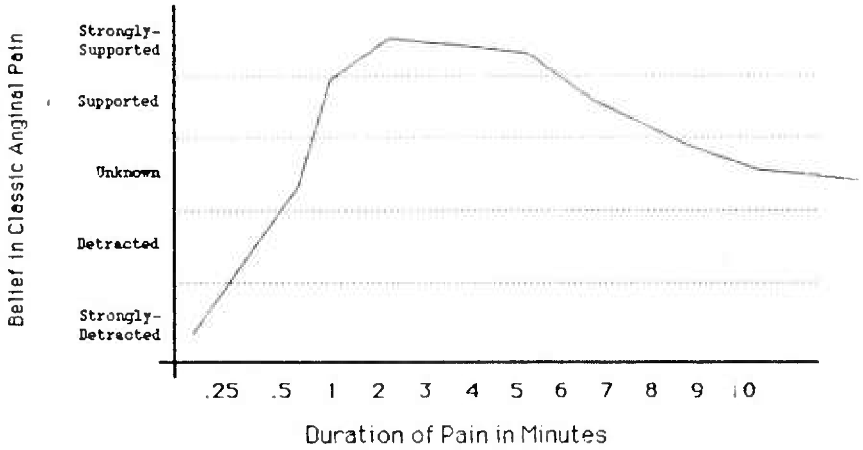
Trigger. A piece of evidence plays the *triggering* role with respect to a hypothesis if its presence focuses attention on the hypothesis, or "brings the hypothesis to mind," or, in MUM, adds the hypothesis to a list of potential diagnoses. Cluster-4, if it is *supported* triggers disease-1 (Fig. 2). This role of evidence is found in virtually all medical expert systems.

Modifying. Some evidence does not support or detract from a hypothesis so much as it alters the way diagnosis proceeds. For example, risk factors for coronary artery disease (e.g., hypertension, elevated cholesterol) play a *modifying* role with respect to the hypothesis of angina since diagnosis will proceed aggressively if they are present and less aggressively otherwise.

These are the only roles currently used in MUM; others are contemplated. Note that evidence can play multiple roles with respect to any hypothesis; for example, risk factors are both *potentially-supporting* and *modifying* with respect to angina; and most *triggers* are individually or in combination with other evidence at least *potentially-supporting* (e.g., note the roles cluster-4 plays with respect to disease-1 in Fig. 2). Also, one piece of evidence can play different roles with respect to several hypotheses (illustrated by the roles cluster-2 plays with respect to disease-1 and disease-2 in Fig. 2). Finally, note that some evidence potentially plays two symmetric roles, while some are "asymmetric". For example, a stress test will either support coronary artery disease or detract from it, while an EKG supports angina if it is positive and is useless otherwise. That is, EKG plays a *potentially-supporting* role only.

Figure 3

A belief curve plotting the datum "Duration of Pain in Minutes" vs. belief in the evidence "Classic-Anginal-Pain"



Clusters. Physicians often see collections of evidence that play particular roles in diagnosis; for example, shortness of breath that comes on suddenly but is unrelated to exercise or other inciting factors *triggers* the diagnosis of pulmonary embolism. Just as evidence has roles with respect to clusters, so clusters have roles with respect to diseases, and these roles need not be supporting; for example, the cluster *(patient-age < 30 and no-fami'y-history-of-coronary-events)* plays a *potentially-detracting* role with respect to all coronary diagnoses of chest pain. Instead of saying that the available evidence is a poor match to coronary diagnoses, we can say the evidence is a good match to a cluster that potentially detracts from or disconfirms coronary diagnoses.

Combining Functions. Every cluster includes a function, specified by the expert, that combines the available evidence for the cluster and returns a value for the cluster given evidence. The values returned by combining functions are just "realizations" of potential roles of evidence. For example, the value returned by the combining function of a cluster supported by *potentially-confirming* evidence could be *confirmed*. The value for a cluster with several pieces of *potentially-detracting* evidence might be *disconfirmed*, or perhaps *detracted*. Combining functions are further discussed below.

Diseases. A disease is technically a cluster. It is a collection of clusters, each of that plays an evidential role in diagnosis and is combined by combining functions with other clusters. Thus diseases reside at the top of a hierarchy of clusters (as shown in Fig. 2), each of which has its own combining function and specifications of the roles played by the clusters below it.

Strategic Knowledge. We characterize strategic knoledge as heuristics for deciding which triggered disease hypotheses to focus on, and how to go about selecting actions to gather evidence pertinent to these hypotheses. These heuristics have the same contingent nature as Davis' meta-rules (Davis, 1985) and control rules in Neomycin (Clancey, 1985). Strategies are represented as rules which include:

- *conditions* for selection of the strategy;
- a *focus policy* which guides the choice of a subset of the triggered disease hypotheses to focus on;
- *planning criteria* which guide the selection of actions to gather evidence for and treat diseases currently in the focus.

Examples of focus policies are *plausibility* (choose hypotheses based on their degree of support); *criticality* (focus on hypotheses that, if true, would require immediate action); and *differential* (focus on hypotheses that offer alternate explanations for the symptoms). Examples of planning criteria are *cost* (prefer evidence that is easy to obtain, and inexpensive on some cost metric); *roles* (prefer potentially-confirming over potentially-supporting); and *diagnosticity*, meaning that a given result has the potential to increase the belief in one hypothesis and decrease belief in the other, as indicated by belief curves.

2.3.2 Combining Evidence and Propagating Belief

MUM combines evidence with local combining functions, as shown in Figure 2. Typically, knowledge systems require three functions to combine evidence and propagate belief. These are illustrated in the context of two inference rules:

$$\begin{aligned} &\text{R1: (A AND B)} \rightarrow \text{C} \\ &\text{R2: (D AND E)} \rightarrow \text{C} \end{aligned}$$

One function calculates the degree of belief (dob) in a conjunction from degrees of belief in the conjuncts:

$$\mathbf{dob}(\text{AND A B}) = F_1(\mathbf{dob}(A)\mathbf{dob}(B))$$

The second function calculates the degree of belief in a conclusion from

a) the degree of belief in its premise (computed by F_1)

b) the "conditional" degree of belief in the conclusion given the premise;

often called the degree of belief in the inference rule:

$$\mathbf{dob}(\text{C}_{R1}) = F_2(\mathbf{dob}(\text{AND A B}), \mathbf{dob}(\text{C}|(\text{AND A B})))$$

The third increases the degree of belief in a conclusion when it is derived by independent inferences:

$$\mathbf{dob}(C_{R1\&R2}) = F_3(\mathbf{dob}(C_{R1}), \mathbf{dob}(C_{R2}))$$

In MUM, these three kinds of combining are maintained, but with two important differences. First, there are no *global* functions corresponding to F_1, F_2, and F_3; all combining is done by functions local to clusters. Second, instead of the usual numeric degrees of belief, MUM has seven levels of belief: *disconfirmed, strongly-detracted, detracted, unknown, supported, strongly-supported, confirmed.* These are just "realizations" of the roles of evidence described earlier.

Combining evidence and propagating belief in MUM is illustrated in Figure 2. Each cluster, including diseases, has its own local combining function, specified by an expert. For example, cluster-1 is *strongly-supported* if the data *support* evidence-1 and if the data on a patient's smoking habits support evidence that he or she is a nonsmoker. This is a conjunction of evidence of the kind calculated by F_1, above. Another is found in the combining function for disease-1. If cluster-2 and cluster-4 are both *confirmed*, then disease-1 is *strongly-supported*. This illustrates the kind of combining for which F_2, above, is required: even when the evidence for a disease is itself certain, the conditional belief in the disease given the evidence may not be certain. Disease-2 also contains a conjunctive rule, but the entire combining function illustrates the corroborative situation for which F_3 is needed. In this case, cluster-4 and cluster-2 *individually* play *potentially-supporting* roles, and taken together [illegible]ease the level of belief in disease-2 to *strongly-supporting*.

Local combining functions have many advantages. Foremost is the ease with which an expert can specify *precisely* how the level of belief in a cluster depends on the levels of belief in the evidence for that cluster. Control of combining evidence is not relinquished to an algorithm, but is acquired from the expert as part of his or her expertise. Since local combining functions are specific to clusters, they can be changed independently. And since the values passed between them in MUM are few, it is easy to trace back the derivation of a level of belief and pinpoint a faulty local combining function. The prospect of having to acquire many functions seems daunting, but we have found it easy and intuitive, and much easier to explain than a global numeric method.

2.3.3 Control of Diagnosis in MUM

Strategic control knowledge, which may be acquired and modified like any other domain knowledge, will be described in the context of the basic control loop which it directs. The implementation of MUM's basic control involves three components:

User Interface: uses data description frames in the knowledge base to ask questions and create patient data frames for the results;

Matcher: uses the interpretation and combining functions to record the effect incoming data has on the belief states for clusters and disease frames, and triggers new hypotheses as appropriate;

Planner: uses strategic control rules from the knowledge base to guide the selection of focus and the planning process.

The planner controls the user interface and the matcher by requesting their services as described below.

Basic Control. The planner follows a basic control loop within which it interprets strategic control rules. It is implemented in a blackboard system, with knowledge sources specified in the same syntax as that which strategic control rules are compiled into. This facilitates modification of the basic control described here as dictated by the strategic knowledge. The design of the blackboard system was influenced by Hayes-Roth (1985), and shares the emphasis on explicit solution to the control problem. We first describe the basic control loop, then strategies and their selection.

The basic control loop is initiated with the choice of a *strategic phase*. All strategic phases but one include a *focus policy* that directs MUM's attention to a subset of candidate hypotheses. This is followed by the generation of short-term plans to gather evidence and select treatment pertinent to these hypotheses (the rule in Section 2.2.1 represents such a plan). Since the effort of developing lengthy plans may well be wasted in a domain permeated with uncertainty, we currently constrain plans to single actions or sequences of two actions where the applicability of the second depends on the outcome of the first. Several short-range plans may be generated and executed.

Carrying out plans typically consists of invoking the user interface to request some information, updating the status of the diseases with the matcher, and conditional continuation of the plan. When no short-term plans remain, the system iterates the basic control loop to determine if a new strategic phase is appropriate, update the focus, and generate new short-term plans. MUM may respond to asynchronous events such as the alteration of a previously obtained data item by interrupting this basic control loop to reconsider its strategy.

Strategic Control. We represent MUM's overall strategy as an ordered set of rule-like strategic phases, shown in Figure 4. Each phase has conditions that activate it. Once activated, a phase controls MUM's focus of attention and the choice of actions pertaining to the hypotheses in this focus.

The phase **Get General Picture** is invoked when the system is started, and may also be used if all previously considered hypotheses are ruled out. It has no focus policy because no hypotheses are active when it is invoked. It directs the planner to ask for evidence that plays the **potential-trigger** role for one or more hypotheses, pursuing the lowest-cost evidence first. The cluster **initial-consultation** (consisting of age, sex, and primary complaint) meets the criteria of potentially triggering many hypotheses and costing little. The initial consultation usually triggers some hypotheses, which result in a new strategic phase being selected. If no hypotheses were triggered, the planner asks for potential-triggers of higher cost.

The **Initial Assessment for Triggered Hypotheses** phase is invoked when new hypotheses are triggered. Since the conditions of the other strategic phases depend somewhat on the level of belief in candidate hypotheses, this phase gathers preliminary evidence for the hypotheses. The focus is on the triggered hypotheses, so only evidence playing some role relative to these hypotheses is considered by the planner. This phase directs the planner to gather low-cost evidence for the hypotheses. For example, MUM asks about aspects of the patient's episode (the event which is the primary complaint) which bear on the triggered hypothesis, and about risk factors.

As soon as the easy questions for triggered hypotheses have been asked, MUM decides between the next two phases based its belief in the hypotheses and whether any of the hypotheses are *critical*, that is, require immediate treatment if supported. Critical hypotheses are dealt with first.

The **Deal With Critical Hypotheses** phase places all candidate critical hypotheses in MUM's focus. The short range planner is then directed to attempt to rule out these hypotheses. It begins with potentially disconfirming or potentially-detracting evidence. If it fails to

Strategic Phase:	Get General Picture.
Conditions:	No candidate hypotheses.
Focus Policy:	None.
Planning Criteria:	Evidence must play trigger role; prefer low cost on all cost metrics.
Strategic Phase:	Initial Assessment for Triggered Hypotheses.
Conditions:	One or more hypotheses are triggered.
Focus Policy:	Focus on triggered hypotheses.
Planning Criteria:	Must be low on all cost metrics; prefer stronger roles.
Strategic Phase:	Deal With Critical Possibilities
Conditions:	There are critical hypotheses which have not been confirmed. disconfirmed or strongly detracted, and if they are detracted, no other hypothesis is confirmed.
Focus-Policy:	Criticality.
Planning Criteria:	Rule Out if possible, else gather support. Utility of evidence. Low cost first; as needed let discomfort and monetary cost increase.
Strategic Phase:	Discriminate Strongest Hypotheses
Conditions:	More than one hypothesis is supported.
Focus-Policy:	Plausibility.
Planning Criteria:	Diagnosticity. Low cost first. Utility of evidence. Substitute high cost confirmation for one hypothesis with lower cost disconfirmation for the other.

Figure 4: Four Strategic Phases in MUM's Diagnosis

find any, then it looks for potentially-supporting evidence. It will not seek evidence that plays a lesser potential role than evidence it already has. For example, it will not seek potentially-supporting evidence for a hypothesis that is already strongly supported, but rather focuses on potentially-confirming evidence. The planner will focus on low-cost evidence first, but it is not prohibited from pursuing high-cost evidence as it was in the previous phase.

If the focus of attention is not captured by critical hypotheses, it is dictated by plausibility. The strategic phase **Discriminate Strongest Hypotheses** discriminates competing alternatives with as little cost to the patient as possible. As before, the potential roles of evidence are used to decide whether it is worth acquiring.

Currently MUM stops work when a hypothesis is confirmed and no critical hypotheses remain in its focus. We are implementing the next strategic phases, prognosis and treatment. Both provide evidence of diagnostic significance; for example, MUM may begin treatment for angina if it is strongly supported, rather than incur the cost of absolute confirmation. If the treatment relieves the symptoms, then it is additional evidence for the diagnosis. If not, it is evidence that detracts from the diagnosis and may support others. Since treatment provides evidence, we represent treatments as clusters, exactly the same way as we represent tests such as angiography.

The emphasis in MUM is on asking the right questions in the right order without superfluous questions. MUM's control knowledge is not yet sophisticated enough to satisfy all these criteria. It asks questions in a reasonable order, but it sometimes focuses on the wrong disease. Since MUM is a nascent system, this does not yet concern us. We believe the system is successful in providing a framework for exploring management of uncertainty by sophisticated control, that is, by making control decisions based on the roles, costs and other characteristics of evidence, the criticality of diseases, and the credibility of diagnoses.

2.4 Conclusions

MUM manages uncertainty by reasoning about evidence and its current state of belief in hypotheses. Its goal is to generate appropriate workups for chest and abdominal pain, that is, to ask the right questions in the right order without unnecessary questions. To the extent it succeeds, it demonstrates its ability to manage uncertainty, and to select the appropriate action given uncertainty. We have said this is a control task. Indeed, much of MUM's architecture is devoted to explicit, evidence-based control.

Much work remains to be done. Currently, MUM resembles a programming environment more than a medical expert system. We are devoting ourselves to building up its knowledge base of clusters, functions, and control rules, while experimenting with improved representations for them.

Although MUM was designed for medical problems and is discussed in that context, we believe the approach to uncertainty and control it engenders is general to classification problem solvers, as well as to other systems responsible for the management of uncertainty. An empty version of MUM called MU is being developed and will be tested in other domains.

3. A Typology for Constructing Decisions

3.1 Introduction

Decision making involves identifying, comparing, and ultimately selecting from among a set of alternatives. When the alternatives are not known in advance, or when the set of alternatives is large, decision making becomes a constructive, action-oriented process. The alternatives and their features, implicit in the description of a decision problem, must be compared and so must be made explicit as the problem is solved. As these comparisons are made, preferences among alternatives on features are also made explicit. We present a typology of decision-making situations that tells how to construct a decision, that is, when to add an alternative, a feature, or a preference to a developing decision.

The emphasis of this work is constructive decision making for AI programs. We focus first on problems where alternatives are supported by conflicting evidence. The many variants of this type of problem are organized into a typology of decision-making situations. Some situations permit an immediate choice between alternatives. Others require actions to further construct the decision. The typology associates appropriate actions with decision-making situations.

The typology shows how to solve "apples and oranges" problems and generalizes this result to provide a view of sophisticated control for decision-making AI programs as table lookup.

Comparing the Incomparable. Decision alternatives are compared on their salient features. Often, the values of these features cannot be easily combined. We call this the *apples and oranges problem:* When you compare apples and oranges in a grocery store you may find one fruit preferred on the basis of flavor and the other on the basis of quality. If you can combine the features to compare the alternatives on a single, composite feature, then the choice is clear. But if, as in this case, flavor and quality cannot be combined, then the choice between apples and oranges is problematic. Traditionally, the apples and oranges problem has been solved by mapping the values of features such as flavor and quality onto a uniform utility scale. The approach described here keeps the features distinct. The inevitable problem of conflicting features is solved by constructively adding features and preferences to a decision.

Closer inspection shows that the apples and oranges problem is not one, but a family of decision problems with different solutions. In this paper, we derive the space of decision problems and show how actions associated with difficult decision problems can be taken to reformulate them as easier ones.

3.2 Decision Typology

We begin with a basic decision problem in which two alternatives are compared on two features, then show how the typology of two-alternative, two-feature problems guides the construction of more complex decisions. Alternatives are referred to as p and q, features as F_i and F_j, and values of features for specific alternatives as $F_i[p]$. The symbol $\tilde{>}$ indicates preference between two values. Although we will be using some mathematical symbols, none of the *values* need be numbers; for example, we can say *flavor*(apples) $\tilde{>}$ *flavor*(oranges) without quantifying quality.

Characteristics of a Decision Two-alternative, two-feature decision problems can be characterized along five binary and ternary dimensions:

$Sd[F_i]$. A *significant difference* on feature F_i indicates that the values of the two alternatives are distinct. If a decision between alternatives p and q can be based on the values $F_i[p]$ and $F_i[q]$, then the values are distinct.

$$Sd[F_i] = \begin{cases} 1 & \text{if } F_i[p] \text{ and } F_i[q] \text{ are distinct} \\ 0 & \text{otherwise} \end{cases}$$

Otherwise indicates no significant difference or that we lack evidence to tell whether there is a significant difference.

$Sd[F_j]$ Like Sd[F_i], but for F_j.

C[F_i, F_j]. A *conflict* exists when F_i and F_j support different alternatives.

$$C[F_i, F_j] = \begin{cases} 1 & \text{if } F_i[p] \tilde{>} F_i[q] \text{ and } F_j[p] \tilde{<} F_j[q] \text{ or} \\ & \text{if } F_i[p] \tilde{<} F_i[q] \text{ and } F_j[p] \tilde{>} F_j[q] \\ 0 & \text{otherwise} \end{cases}$$

O[F_i, F_j]. One feature is often *more important* than another. This means that one feature is preferred to another (e.g., quality is preferred to flavor), or that there is a greater difference between the two alternatives on one feature than the other.

$$O[F_i, F_j] = \begin{cases} 0 & \text{if importance}(F_i) = \text{importance}(F_j) \\ ? & \text{if relative importance unknown} \\ 1 & \text{if importance}(F_i) > \text{importance}(F_j) \\ & \text{or importance}(F_i) < \text{importance}(F_j) \end{cases}$$

$\tilde{>}[F_i, F_j]$. Assuming that $O[F_i, F_j] = 1$, we need to know which feature is preferred.

$$\tilde{>}[F_i, F_j] = \begin{cases} 0 & \text{or importance}(F_i) < \text{importance}(F_j) \\ 1 & \text{if importance}(F_i) > \text{importance}(F_j) \end{cases}$$

We illustrate these dimensions in the context of the problem of selecting fruit: F_i is *quality* and F_j is *flavor*. If the quality of apples is "good" and the quality of oranges is "poor," then $Sd[F_i] = 1$ because good and poor are distinct values. Similarly, if one prefers the flavor of oranges to that of apples then $Sd[F_j] = 1$. Since apples have better quality but oranges taste better, $C[F_i, F_j] = 1$. Finally, if quality is preferred to taste $O[F_i, F_j] = 1$ and $\tilde{>}[F_i, Fj] = 1$.

The space of types characterized by these dimensions can be arranged in a table. The problem we just described is case 23 in this table, illustrated in Figure 5. In English, case 23 says "the quality of evidence for F_i[p] and F_i[q] is sufficient to claim that the difference supports a choice between p and q; the quality of evidence for F_j[p] and F_j[q] is sufficient to claim that the difference supports a choice between p and q; there is a conflict between p and q on F_i and F_j, and the feature F_i is more important than F_j."

Collapsing the Table Figure 5 does not represent all 40 combinations of the possible values of $Sd[F_i]$, $Sd[F_j]$, $C[F_i, F_j]$, $O[F_i, F_j]$, and $\tilde{>}[F_i, F_j]$. From the perspective of how a decision-maker acts, the 40 decision types contain some redundancies. Consider these cases:

Case 18a: S[F_i] = 1, S[F_j] = 0, $C[F_i, F_j] = 1$, $F_i \tilde{>} F_j$

Case 18: S[F_i] = 0, S[F_j] = 1, $C[F_i, F_j] = 1$, $F_j \tilde{>} F_i$

In English, the dimension for which your evidence supports a decision is the most important dimension. The cases are identical in the sense that a decision-maker would not act differently in response to them. Consequently, the two cases are represented only by case 18 in the table.

Case #	0	1	2	3	4	5	6	7	8	9	10	11
Sd$[F_i]$	0	1	1	0	1	1	0	0	1	0	1	1
Sd$[F_j]$	0	0	1	0	0	1	0	1	1	0	0	1
C$[F_i, F_j]$	0	0	0	1	1	1	0	0	0	1	1	1
O$[F_i, F_j]$	?	?	?	?	?	?	0	0	0	0	0	0
$\dot{>}[F_i, F_j]$	*	*	*	*	*	*	*	*	*	*	*	*

Case #	12	13	14	15	16	17	18	19	20	21	22	23
Sd$[F_i]$	0	1	0	1	0	1	0	1	0	1	0	1
Sd$[F_j]$	0	0	1	1	0	0	1	1	0	1	0	1
C$[F_i, F_j]$	0	0	0	0	1	1	1	1	0	0	1	1
O$[F_i, F_j]$	1	1	1	1	1	1	1	1	1	1	1	1
$\dot{>}[F_i, F_j]$	0	0	0	0	0	0	0	0	1	1	1	1

Figure 5: Typology of Decisions

Decision Actions The point of characterizing decisions is to select appropriate actions. In our approach there are three basic actions: *decision*, *transformation*, and *stuck*. *Decision* means choosing an alternative based on available evidence; for example, in case 8 (Fig. 1) there are significant differences between the alternatives on both features and their evidence does not conflict. The decision is straightforward.

Transformations of one decision type into another are appropriate when a decision cannot be made *given the available evidence*. In case 0 (Fig. 1), the values of the alternatives on features F_i and F_j do not distinguish the alternatives, nor do we know whether one feature is preferred. A decision in this case cannot be made with confidence, but several transformations of case 0 are possible: If further evidence about F_i potentially shows that the alternatives *can* be distinguished on F_i, then obtaining the evidence transforms case 0 into case 1 (i.e., the 0 in row $Sd[F_i]$ is replaced by a 1). Obtaining evidence of this kind for *both* features transforms case 0 into case 2. From case 2, one may confidently make a decision. Similarly, if evidence exists that F_i is preferred to F_j, then obtaining the evidence transforms case 0 into case 20. Alternatively, evidence may show that neither feature is preferred; obtaining this evidence transforms case 0 into case 6. The idea of transformations is to change one decision type into another, hopefully more facilitative, type. Transformation is an appropriate action for any decision type with 0 in either of its first three rows or ? in its fourth.

The most obvious way to effect a transformation is to seek more evidence. The table in Figure 5 allows us to plan actions to obtain evidence, thus it guides the process of constructing a decision. However, the planned transformation may not be possible; the actual transformation depends on the evidence obtained. For example, we may gather evidence about F_i with the intention of transforming case 7 to case 8. But if the evidence, when obtained, indicates that F_i and F_j actually support different alternatives, then we end up in case 11 instead of case 8.

In case 11, we are *stuck*: all available evidence about the features has been acquired, but it supports conflicting alternatives, and neither feature is preferred. From case 11, no further transformation is possible, no action is apparent. In fact, there actions appropriate for the stuck case, but they expand the decision beyond the two-alternative, two-feature case under discussion. If a decision cannot be made on the basis of evidence about the current features, then the appropriate action is to further distinguish the alternatives with additional features. Because we view decision making as a constructive process in which alternatives and features emerge only as needed, we imagine a decision-maker adding features only when stuck, that is, in case 11.

Each of the 24 decision types has at least one appropriate action. Some suggest two (see Fig. 2). These are situations in which a decision can be made, but without complete confidence. For example, in case 9 there is significant evidence for F_i, but not F_j, they don't contradict given the available evidence, and neither feature is preferred. A decision could be based on F_i, but not without some uncertainty that F_j actually supports a different alternative than F_i. Multiple actions permit different strategies for selecting specific actions. For example, a conservative strategy that tries to minimize uncertainty in decisions encourages transformations.

3.3 Extensions to a Multifeature Model

The decision tables described so far allow comparison of two alternatives on two of their features. Sometimes, as noted above, a decision cannot be based solely on these features. These situations arise in three ways. First, evidence such as the preference for features may be missing. Second, complete evidence may not support a decision; for example, the values of the alternatives on the features may be accurately known, but not significantly different to support one alternative. Third, these values may be accurately known, and significantly different, but support different alternatives. In the first situation, it is fairly obvious that we should seek the missing evidence. In the last two, it is necessary to add another feature. Psychological evidence suggests that humans in these situations add features and alternatives conservatively, what [Svenson 79] calls "choice by feedback processing." Our model emulates this iterative, constructive behavior.

Adding Features Features may be added by substituting one for another or by combining a new feature with an old one. In either case, the typology of Figure 6 suffices to represent two-alternative, multi-feature decisions. In *substitution*, one of the two features currently under consideration is discarded and a new feature is substituted. This is appropriate when we know that two alternatives are not differentiated on an feature ($Sd[F_i] = 0$). The feature does not provide a basis for a choice. It should be replaced by another, more informative, feature.

The second method for adding features is *combination*: the evidence provided by the new feature is combined with evidence accrued from previous comparisons. This is appropriate when the previous features favor different alternatives. For example, when we add another feature F_{new} to case 11, [1110*], we hope to move to column 19, [11110], or 23, [11111]. Unlike case 11, cases 19 and 23 indicate a preference between features. Assuming that the alternatives are distinguished on F_{new} (otherwise adding it would gain nothing), and assuming that a combination of two significant features are preferred to one, F_{new} in.-oduces a preference order when combined with the old feature it corroborates, resulting in case 19 or 23. Thus, the typology of Figure 6 suffices for a two-alternative, three-feature decision and, by induction, for two-alternative, multi-feature decisions. Since case 11 involves a conflict between features, F_{new} must corroborate either F_i or F_j. Thus, new evidence can clustered to support one of two alternatives. This additional support contributes to an ordering over clusters of features, represented by values in the fourth (order) and fifth (preference) rows.

Clustering is the key to extending the two-alternative, two-feature situations to two-alternative, N-feature cases and finally to N-alternative, N-feature problems, because it permits complex decision situations to be constructed iteratively within the framework of our decision typology.

Case	0	1	2	3	4	5	6	7
$Sd[F_i]$	0	1	1	0	1	1	0	0
$Sd[F_j]$	0	0	1	0	0	1	0	1
$C[F_i, F_j]$	0	0	0	1	1	1	0	0
$O[F_i, F_j]$	?	?	?	?	?	?	0	0
$\tilde{>}[F_i, F_j]$	*	*	*	*	*	*	*	*
Action	D/T	D/T	D	T	D/T	S/T	D/T	D/T

	8	9	10	11	12	13	14	15
$Sd[F_i]$	1	0	1	1	0	1	0	1
$Sd[F_j]$	1	0	0	1	0	0	1	1
$C[F_i, F_j]$	0	1	1	1	0	0	0	0
$O[F_i, F_j]$	0	0	C	0	1	1	1	1
$\tilde{>}[F_i, F_j]$	*	*	*	*	0	0	0	0
Action	D	T	D/T	S	D/T	T/D	D/T	D

	16	17	18	19	20	21	22	23
$Sd[F_i]$	0	1	0	1	0	1	0	1
$Sd[F_j]$	0	0	1	1	0	1	0	1
$C[F_i, F_j]$	1	1	1	1	0	0	1	1
$O[F_i, F_j]$	1	1	1	1	1	1	1	1
$\tilde{>}[F_i, F_j]$	0	0	0	0	1	1	1	1
Action	D/T	D/T	D/T	D/S	D/T	D	D/T	D/S

Figure 6: Decision Actions

Revised Set of Decision Actions With the ability to cluster evidence, we can determine what to do even in very difficult decision situations. The initial set of actions, *decision, transformation*, and *stuck* can be augmented. The new set is *decision*, *transformation by feature*, *transformation by order*, *substitution*, and *combination*. In transformation by feature (*Tf*), we acquire additional evidence about whether a feature distinguishes alternatives. This can change $Sd[F_i] = 0$ to $Sd[F_i] = 1$. Transformation by order (*To*) is the corresponding action for gathering order preference information. It can transform $O[F_i, F_j] = ?$ to $O[F_i, F_j] = 0$ or $O[F_i, F_j] = 1$. If complete knowledge of the alternatives is available, but a decision still cannot be made, a state can be transformed by adding a new feature, either by substitution (*Su*) or combination (*Co*).

Figure 7 contains the decision states with their appropriate actions. The actions are divided into two rows. The first row shows the actions for states with complete evidence. The second describes actions to be performed when some of the state information is missing. The transformations are listed with numbers that indicate tne set of possible states you might end up in. Note it is not possible to say exactly which of these states will arise.

The actions presented in Figure 7 are somewhat subjective. In general, combination can be done in any state. It isn't listed because other actions are often more

Case		0	1	2	3	4	5	6	7
Sd$[F_i]$		0	1	1	0	1	1	0	0
Sd$[F_j]$		0	0	1	0	0	1	0	1
C$[F_i, F_j]$		0	0	0	1	1	1	0	0
O$[F_i, F_j]$		?	?	?	?	?	?	0	0
$\tilde{>}[F_i, F_j]$		*	*	*	*	*	*	*	*
Actions	All Info	Co D	Su D	D		Su D	Co	Co D	Su D
	Part Info	Tf 0, 1,4	Tf 1,5,8 To 7, 13,14		Tf 3,4,5 To 9, 16,22	Tf 2,4 To 10, 17,18	To 11, 19,23	Tf 6, 7,10	Tf 7, 8,11

		8	0	10	11	12	13	14	15
Sd$[F_i]$		1	0	1	1	0	1	0	1
Sd$[F_j]$		1	0	0	1	0	0	1	1
C$[F_i, F_j]$		0	1	1	1	0	0	0	0
O$[F_i, F_j]$		0	0	0	0	1	1	1	1
$\tilde{>}[F_i, F_j]$		*	*	*	*	0	0	0	0
Actions	All Info	Co D	Su	Su	Co	Co	Su D	Su	Co D
	Part Info	To 15,21	Tf 9,10,7	Tf 10,11,8		Tf 12,13, 14,17,18	Tf 13,15, 17,19	Tf 14,15, 18,19	

		16	17	18	19	20	21	22	23
Sd$[F_i]$		0	1	0	1	0	1	0	1
Sd$[F_j]$		0	0	1	1	0	1	0	1
C$[F_i, F_j]$		1	1	1	1	0	0	1	1
O$[F_i, F_j]$		1	1	1	1	1	1	1	1
$\tilde{>}[F_i, F_j]$		0	0	0	0	1	1	1	1
Actions	All Info	Co	Su	Su	Co	Co	Co D	Co	Co
	Part Info	Tf 16,17, 13,14,18	Tf 17,10, 15	Tf 18,19, 15		Tf 20,13, 14,17,18		Tf 22,13, 14,17,18	

Figure 7: Revised Multi-Feature Decision Actions

appropriate; for example, substitution is more appropriate when one feature is insignificant. Decision could be made in cases other than those listed, but they would be precarious decisions.

3.4 Changes to Decision State

Adding a new feature potentially affects every cell in a decision state, that is, each value Sd$[F_i]$, Sd$[F_j]$, C$[F_i, F_j]$, O$[F_i, F_j]$, and $\tilde{>}[F_i, F_j]$. In combination with a new feature, a previously insignificant one may becomes significant (e.g., $Sd[F_i] = 0$ but $Sd[F_i and F_{new}] = 1$). Less obviously, adding a new feature can make a previously significant one insignificant. This happens when the alternatives differ so enormously on the new feature that any differences on the old one(s) cease to be significant. C$[F_i, F_j]$ may change if the new feature produces a conflict, and O$[F_i, F_j]$ and $\tilde{>}[F_i, F_j]$ change by clustering features. Within the framework of our typology, the effects of adding a new feature are:

1. to introduce a conflict where there was none
2. to take a side in a conflict
3. to join the consensus (C$[F_i, F_j, F_k] = 0$) but lend it legitimacy since $Sd[F_k] = 1$
4. to introduce an ordering where there was none (e.g. O$[F_i, F_j]$=0 but O$[F_i(F_j, F_k)] = 1$)
5. to change an ordering (e.g., $\tilde{>}[F_i, F_j] = 1$ but $\tilde{>}[F_i, (F_j, F_k)] = 0$
6. to produce a change in relative significance when adding radically divergent features.

Figure 8 shows all the possible actions and their effects for a single case in the typology, case 4. In this example, there is enough of a difference to support a decision on F_i, but not F_j and the evidence of the two features is contradictory. Four actions are appropriate: transformation by feature (the 0 value for Sd$[F_j]$ may indicate insufficient evidence), transformation by order, substitution (for F_j), and combination. Note that it is possible to return to the same state, case 4, but by different paths. Substituting F_j or combining features transforms case 4 to case 5. But note that when case 5 was reached by combining features, one of them, F_i or F_j, actually represents the evidence of two features and so supports a decision more strongly. (This difference will be represented explicitly in a more complete state table).

The Mechanics of Combining Features As mentioned above, combining features may produce major changes in the decision state. However, the set of possible new states can be enumerated. Figure 9 presents the set of possible states that can be reached by combining a new feature with all previous states.

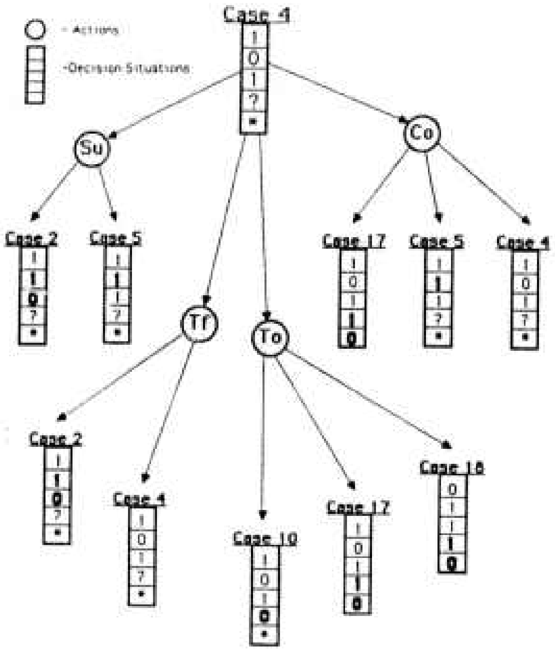


Figure 8: Single Transition with Multiple Features

$\mathrm{Sd}[F_i]$	$\mathrm{Sd}[F_j]$	$\mathrm{C}[F_i, F_j]$	$\mathrm{O}[F_i, F_j]$	$\tilde{>}[F_i, F_j]$
0	0	0	0	*
0	0	1	0	*
1	0	0	0	*
1	0	1	0	*
1	0	1	0	*
1	1	1	0	*
1	1	1	0	*
0	0	0	1	0/1
0	0	1	1	0
0	0	1	1	0
0	0	1	1	1
0	0	1	1	1
1	0	0	1	0/1
1	0	1	1	0
1	0	1	1	0
1	0	1	1	1
1	0	1	1	1
1	1	0	1	0/1
1	1	1	1	0
1	1	1	1	0
1	1	1	1	1
1	1	1	1	1

$\mathrm{Sd}[F_k]$	$\mathrm{Sd}[F_C]$	C[all]	$\tilde{>}[F_C, F_x]$	Transition	#
1	ij 0/1	0/1	?	0000* → 0/1 1 0/1 0/1 ?	0
1	ik 1	1	1	0010* → 1 0 1 1 0	1
1	ij 1	0/1	?	1000* → 1 1 0/1 0/1 ?	1
1	ik 1	1	1	1010* → 1 0 1 1 0	2
1	jk 1	1	1	1010* → 1 1 1 1 1	1
1	ik 1	0/1	?	1100* → 1 1 0/1 0/1 ?	2
1	ik 1	1	1	1110* → 1 1 1 1 0	2
1	ij 0/1	0/1	?	00010/1 → 0/1 1 0/1 0/1 ?	0
1	ik 1	1	0	00110 → 1 0 1 1 0	1
1	jk 1	1	?	00110 → 0 1 1 0/1 ?	1
1	ik 1	1	?	00111 → 1 0 1 0/1 ?	1
1	jk 1	1	1	00111 → 0 1 1 1 1	1
1	ij 1	0/1	?	10010 → 1 1 0/1 0/1 ?	1
1	ik 1	1	0	10110 → 1 0 1 1 0	2
1	jk 1	1	?	10110 → 1 1 1 0/1 ?	1
1	ik 1	1	?	10111 → 1 0 1 0/1 ?	2
1	jk 1	1	1	10111 → 1 1 1 1 1	1
1	ij 1	0/1	?	11010/1 → 1 1 0/1 0/1 ?	2
1	ik 1	1	0	11110 → 1 1 1 1 0	2
1	jk 1	1	?	11110 → 1 1 1 0/1 ?	2
1	ik 1	1	?	11111 → 1 1 1 0/1 ?	2
1	jk 1	1	1	11111 → 1 1 1 1 1	2

Figure 9: Transitions upon Combining Features

The first five columns of Figure 9 have the same values as the rows in previous tables. $\mathrm{Sd}[F_k]$ is the significant difference value of the new feature; it is always 1, indicating that the new feature discriminates the alternatives. $\mathrm{Sd}[F_C]$ is the significant difference of the combined features; the values in its column are the features that have been combined along with their possible values. C[all] shows whether there is a conflict between the combined values and the single feature. $\tilde{>}[F_N, F_x]$ describes an order between the combined feature and the single feature. The column labeled 'Transition' shows the possible transitions from that state. Finally, # indicates how many significant features had been combined to produce the F_C feature.

Figure 9 presents the single step transitions when adding features to states as represented in the two feature tables. We are currently working on a state transition diagram that will describe all the possible transitions in the construction of a decision between two alternatives.

3.5 Conclusions

We have presented a model of constructive decision making. We envision a decision-maker starting with a two-alternative, two-feature problem, then acquiring information, and perhaps adding features, under the guidance of actions associated with decision types. This model raises the intriguing possibility of controlling decision making in AI programs by table lookup. Each decision situation is first classified, then modified by one of the associated actions. The model is not intended to produce optimal solutions to complex decision problems given complete information, but rather to explore methodologies for structuring decision problems, performing symbolic comparisons, and reasoning about uncertain decisions.

Other systems have viewed decision making as a constructive process. GODDESS, a domain independent decision support system, constructs a hierarchical goal representation of decision alternatives by selectively focusing the users attention on the most crucial issues [Pearl 82]. Users assign numeric values to probabilities and importance, and the program propagates them through the structure. ARIADNE does not address the decision formulation problem, but rather emphasizes evaluation by using linear programming algorithms to produce a dominance structure for the alternatives' probabilities and utilities and by allowing the iterative addition of alternatives [Sage 84].

Three facets of the decision typology model are particularly appealing. First, two-alternative, two-feature decisions can be characterized according to the dimensions of the decision without requiring an underlying scale of comparison. Second, the typology relates actions to decision types. Finally, the model shows how to change difficult decisions into more tractable ones using well defined transformations that explicitly identify the possible results of actions.

Before the model is fully realized, we must resolve two issues. First, the conditions and mechanisms for adding new alternatives must be specified as they were for new features. We believe that alternatives can be clustered like features, so the two-alternative, two-feature typology might serve for multiple alternatives and features. The second issue is to add continuous values to the model. The binary/ternary formalism is abstract. For most situations, this abstraction is not only acceptable, but fully indicative of the appropriate actions. However, it does not explicitly capture the effects of extreme values or context. $Sd[F_i]$ indicates a disparity between alternatives on F_i, but not its magnitude. The difference in degree of differentiation between alternatives on features is captured in the $O[F_i,F_j]$ dimension, which may favor the feature that produces a great disparity. This, in turn, implies that $O[F_i,F_j]$ is *not* independent of alternatives.

4.Classification by SemanticMatching

4.1 Introduction

Classification problem solving involves matching data with pre-established prototypes (Clancey, 1984). Often the match is not exact: it may be partial because some aspects of the prototype lack matches in the data. This paper describes another kind of partial matching and the role it can play in classification problem solving. Semantic matches hold between concepts that are linked in characteristic ways in a semantic network. We have found that the degree of fit between data and a prototype depends on these semantic matches. Moreover, the likelihood of a prototype given the data (in the conditional sense) depends on these matches. In another paper we argued that degrees of belief in classification problem solvers should be interpreted in terms of semantic matches (Cohen et al. 1985). We have developed a program called GRANT that exploits semantic matching to find sources of research funding that are likely to support particular research proposals.

4.2 GRANT

GRANT is a knowledge system that finds sources of funding for research proposals. The user builds a representation of a research proposal and instructs GRANT to search for funding agencies that are likely to provide support. GRANT first constructs, then ranks, a *candidate list* of agencies. An agency is added to the candidate list if a single topic in its statement of interests is a good semantic match to a topic in the research proposal. Semantic matches exist between topics that are the endpoints of particular *paths* through a semantic network. Agencies on the candidate list are ranked by the number of semantic matches between all the topics in the proposal and all the topics in each agency's statement of interests. The best-ranked agencies are thus those that support the largest number of topics that are semantically related to the proposal.

4.2.1 Knowledge Representation

GRANT depends on a knowledge base (KB) of research topics and a set of rules for searching it. The latter is described in the next section. The KB is a semantic network of approximately 4500 node with over 800 research topics. Figure 10 shows a fragment of GRANT's knowledge about the heart, cardiovascular illness, and related topics. Nodes in the network are defined in terms of their relationships with others; for example, the heart is something with the *purpose* of circulation, the *setting* of cardiovascular illness, and an

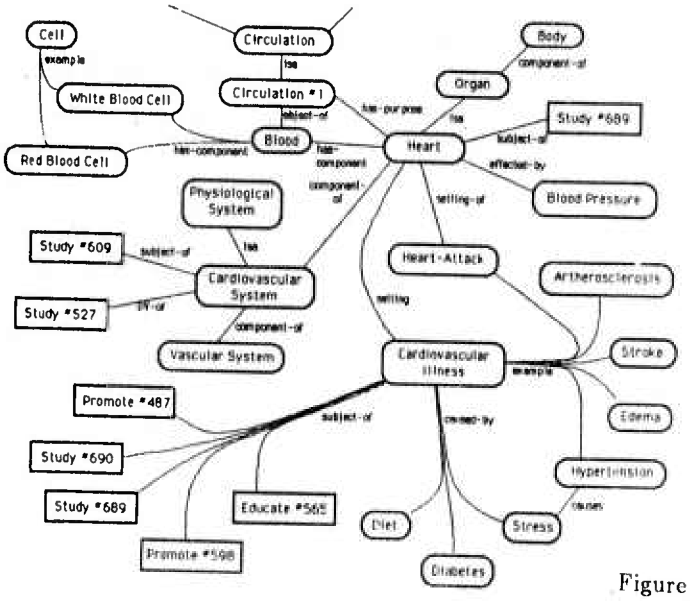


Figure 10:

example of an organ[3]. Appendix 1 lists the most common relations between topics in the GRANT KB.

The GRANT KB acts as a semantic index to funding agencies. Nodes are added to the semantic network as necessary to define the research interests of agencies. An agency is represented as a frame with slots for stated research interests, average award size, citizenship restrictions, geographic preferences, and so on. The **research-interest** slot holds pointers to instances of one or more *activities* that are linked with topics in the KB. GRANT recognizes 10 activities:

Design Educate Improve Intervene Manag
Plan Promote Protect Study Train

For example, the agency associated with *study-689* in Figure 10 is interested in funding studies of cardiovascular illness and the heart. GRANT's KB currently includes the 690 agencies that together provide most of the research monies at the University of Massachusetts.

When GRANT's user creates a research proposal, it is linked into the KB through its research interests just as funding agencies are. The frames that represent agencies and proposals have the same slots, illustrated in Figure 11.

4.2.2 Search Algorithms

GRANT finds agencies to fund a research proposal by finding *paths* between the nodes that represent the proposal's research interests and nodes associated with agencies. A *blind search* of the network in Figure 10 would begin, say, at the node *study-527* and extend to its associated node *cardiovascular system*, then to the associations of this node *physiological-system, vascular-system, heart, study-609* and so on, like ripples in a pond. If a node is found that represents a research interest of an agency, then a path has been established between the proposal and that agency. The GRANT KB includes so many agencies and is so highly connected that, on average, blind search finds 245 agencies within 4 links of any proposal. But according to our expert, on average 93.1% of these agencies are *unlikely* to fund the proposal. For GRANT to be useful, this *false-positive* rate must be reduced. One method is to avoid finding unlikely agencies, and the other is to discard them once they are found. These methods are discussed in turn.

[3] And thus, by a plausible inference, a *component-of* the body. See Section 5.

> The ABC Foundation is interested in providing both grants and direct loans in order to help promote sexual education and to help control sexually transmitted diseases. Funds are available for the management and maintenance of clinics ...

Funding-source*4:

is-a	: funding-source
title	: "ABC Foundation"
descr	: "... promote sexual education and to help ..."
topic	: manage*4

Manage*4:

is-a	: manage
topic-of	: funding-source*4
object	: clinic
subject	: sexually-transmitted-disease
focus	: gonorrhea herpes venereal-disease contraceptive
purpose	: control educate

Figure 11: The ABC Foundation is represented by the frames FUNDING-SOURCE*4 and MANAGE*4

Best-first Search. One can avoid finding unlikely agencies by pruning the paths that lead to them during search. Figure 12 shows three kinds of paths. The first is an *atomic match* between the proposal and the agency: the *object* of the proposed *study-418* is *vascular-disease*, which is also the *object* of *study-297*, a research interest of the agency. With few exceptions an atomic match indicates that the agency is likely to fund the proposal.

Since the links in GRANT are directional, and searches proceed from proposals to agencies, the path between the proposal and NHLBI is

$study-418 \xrightarrow{object} vascular-disease \xrightarrow{object-inverse} study-297$

A *path endorsement* is a generalization of a set of paths, obtained by dropping intermediate nodes and preserving only the relations. The path above is thus an instance of a general *(object, object-inverse)* path endorsement.

The second path in Figure 12 is a *semantic match* between a proposal and an agency. The proposal wants to study hypertension. Whereas an *atomic* match, represented by a path endorsement like *(object, object-inverse)*, guarantees that proposal and agency have a common interest, a semantic match ensures only that the interests of the proposal and agency are somehow related. *The nature of the relationship, represented by a path endorsement, determines the likelihood that the agency will fund the proposal.* For example, when an agency says it funds research on vascular disease, it means that it funds research on many or all kinds of vascular disease, including hypertension. This argument holds for agencies and topics in general: if agencies say they fund X, they are likely to fund instances of X. By this reasoning, if we begin a search at a proposal and follow a *(object, isa, object-inverse)* path to an agency, then the agency is likely to fund the proposal. Any path that is an instance of the *(object, isa, object-inverse)* path endorsement is apt to find a likely agency.

Just as path endorsements mark likely paths to agencies, so they mark paths to be avoided. The third path in Figure 12 is an example. The research topic of the proposal is *anorexia* and that of the agency is *bulimia*. Now bulimia is an instance of an *eating-disorder* and when an agency says it will fund the study of an instance of X it usually means that it will not fund the study of *other* instances of X. This agency is unlikely to fund the study of other eating disorders such as anorexia. In general, if a path between a proposal and an agency is an instance of the path endorsement *(object, isa, isa-inverse, object-inverse)*, then the agency is unlikely to fund the proposal and the path should be avoided.

Path endorsements thus constrain the search for agencies in GRANT. Appendix 2 lists some of GRANT's path endorsements. The complete set of path endorsements is still only a fraction of the combinatorially possible path endorsements. Any path that has not been classified as likely or unlikely is denoted *unknown*. Best-first search in GRANT proceeds as follows:

> Assume the program starts at a proposal and follows link l_i to node n_i: $\langle l_i n_i \rangle$. If a continuation of this path along link l_j to node n_j results in a path endorsement (l_i,l_j) that GRANT recognizes as poor, then n_j is pruned from the list of nodes that GRANT tries to expand. If (l_i,l_j) is a good path endorsement, then GRANT will give n_j priority to be expanded before any node n_k found by an *unknown* path $\langle l_i n_i l_k n_k \rangle$. Search from any path longer than 4 links is terminated.

Ranking Agencies by Partial Matching. The result of best-first search is a candidate list of agencies. Each is known to have a single research interest that atomically or semantically matches one research interest of the proposal. To the extent that the proposal and an agency share several common research interests, the agency is more likely to fund the proposal. Thus, GRANT ranks the candidate list of agencies by the degree of overlap between the research interests of the proposal and each agency. This is done by a partial matching function based on both atomic and semantic matching. Hayes-Roth (1978), Tversky (1977), and others measure the degree of overlap between sets in terms of set intersection and symmetric difference; for example, Tversky's *contrast model* (1977) calculates overlap this way:

$$S(a,b) = \theta f(A \cap B) - \alpha f(A - B) - \beta f(B - A).$$

The function f returns the cardinality of the set to which it is applied. If A and B are frames, then f(A $\cap$ B) is the number of slot-value pairs shared by A and B, and f(A $-$ B) is the number of slot-value pairs in A not shared by B. The parameters θ, α, and β are set empirically; in GRANT each is 1.0. If A and B are frames representing the research interests of a proposal and an agency, respectively, then $S(a,b)$ measures the number of research topics they have in common relative to those they do not share. Agencies for which $S(a,b)$ is higher are more likely to fund the proposal.

In GRANT, (A $\cap$ B) includes both atomic and semantic matches. If a path between A and B contains a single node (e.g., the first case in Fig. 12), or if the path is an instance of a likely path endorsement (e.g., the second case in Fig. 3), then f(A $\cap$ B) is incremented. Unlikely path endorsements, such as the third case in Figure 12, and unknown paths do not contribute to f(A $\cap$ B). The quantities f(A $-$ B) and f(B $-$ A) are increased when research topics in the proposal lack an atomic or semantic match to the agency, and vice versa.

In summary, GRANT searches for agencies in two stages. First it constructs a candidate list of agencies by best-first search in a semantic network of research topics, then it ranks the agencies on the list by their degree of overlap with the research proposal.

4.3 Analysis of GRANT Performance

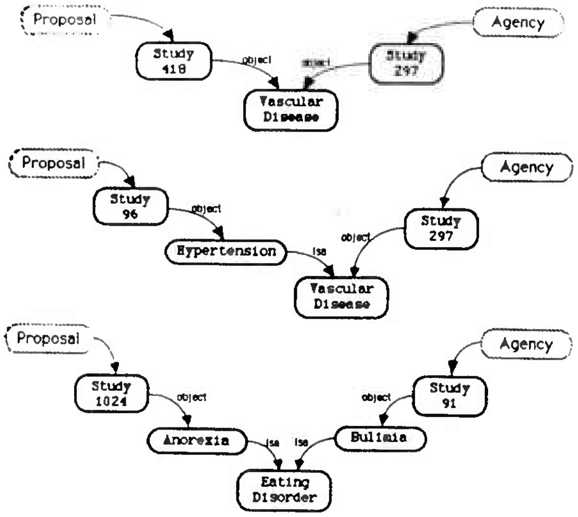
GRANT's performance has been tested at all stages of its development. The basic method is to run samples of proposals and compare the agencies selected by GRANT with the choices of our expert. Sample sizes have ranged between 20 and 30 proposals. We compute many statistics for each search from a proposal, but two are broad indicators of GRANT's performance:

$$\text{hit-rate} = \frac{\text{agencies judged good by GRANT and by the expert}}{\text{agencies judged good by the expert}}$$

$$\text{false-positive rate} = \frac{\text{agencies judged good by GRANT and bad by the expert}}{\text{number of agencies judged good by GRANT}}$$

Figure 12:

Paths Between Proposals and Agencies



We average these statistics over the searches from the individual proposals in a sample.

When we first tested GRANT (Cohen et al., 1985) its knowledge base contained approximately 700 nodes and 50 agencies. We contrasted blind and best-first search as follows: for each of 23 proposals the system searched blindly for agencies until it reached a predetermined stopping criterion. On average, blind search found 15.1 agencies per proposal. We gave our expert the list of agencies found for each proposal by blind search and asked him to rank each agency as likely or unlikely to fund the proposal. On average, only 2 agencies per proposal were considered likely; that is, the false-positive rate for blind search was (15.1 - 2)/15.1 = 86%. In contrast, best-first or path endorsement constrained search found on average just 2.78 agencies per proposal, of which 1.48 were judged likely to fund the proposal. The false-positive rate was 32%, a big improvement over blind search. The downside was a hit rate of 80%, indicating that GRANT had pruned away one likely agency in five. We have tested all subsequent versions of GRANT this same way, using blind search to find candidate agencies and an expert to rank them, then comparing best-first search with the expert's rankings. Table 1 shows best-first search statistics for several versions of GRANT. Blind search statistics are not represented; in all tests blind search had a false positive rate greater than 80%, and as the knowledge base increased in size this figure increased dramatically.

Grant, Spring 85 (700 nodes, 50 agencies)	
Hit Rate	80%
False Positive Rate	32%
Grant, Fall 85 (2,000 nodes, 200 agencies)	
Hit Rate	80%
False Positive Rate	26%
Contrast Model	
Hit Rate	76%
False Positive Rate	22%
Grant, Winter 86 (4,500 nodes, 700 agencies)	
Hit Rate	98%
False Positive Rate	61%
Contrast Model	
111: Hit Rate	96.1%
111: False Positive Rate	57%
Grant, Winter 86 (4,500 nodes, 700 agencies)	
Modified Path Endorsements	
Hit Rate	96.3%
False Positive Rate	55.8%
Contrast	
111: Hit Rate	96.4%
111: False Positive Rate	53.4%

Table I.

The differences between GRANT today and the version we tested in Spring, 1985 are its size and the incorporation of Tversky's contrast model for summing the total degree of overlap between proposals and agencies. The false positive rate of the early version, 32%, decreased during the subsequent months as the knowledge base increased to 2000 nodes with 200 agencies. At that time we introduced the contrast model, described above, and realized a further small decrease in the false positive rate, which was offset by a decrease in the hit rate. In the last two months we have again more than doubled the size of the knowledge base and more than tripled the number of agencies from the Fall, 1985 level. As a result, performance has decreased substantially. The hit rate of best-first search is 98%, but the false positive rate is 61%: the system finds virtually all the agencies it should, but nearly two-thirds of the agencies it finds are not likely to fund the proposal.

Why did the increase from Spring, 1985 to Fall, 1985 not decrease GRANT's performance, while the latter one did? Many factors are involved. First, the density of agencies is increasing. In the early version, 700 nodes supported 50 agencies – a ratio of 14:1. In Fall, 1985, the ratio was 10:1. The most recent knowledge base has a ratio of 6.4:1. It is much easier to find many agencies close to a proposal in GRANT's semantic net than it was in the past. Indeed, we have evidence to suggest that as the density of the knowledge base increases, the hit rate goes up and the false positive rate down: An intermediate version of the Winter, 1986 knowledge base included approximately 600 *orphans*, nodes used to define another node but disconnected from all other nodes. In this version, the density of nodes per agency was 5.8:1. There were too many agencies and too few associative paths to differentiate good agencies from bad ones.

A second contributor to the high false positive rate in the Winter, 1986 version is the kinds of agencies being represented. Roughly 200 of the new agencies were for the arts and humanities. Their descriptions of research interests were fairly broad and gave little basis for differentiation. Consequently, when GRANT searches in that part of the knowledge base, its false positive rate increases dramatically. A related problem is that in the most recent version of GRANT, new agencies were not represented in as much detail as old ones. Necessarily, this meant viable distinctions between agencies were lost.

The relations we use to represent agencies have not changed appreciably since the early version of GRANT, but the number of things they are required to represent is greatly increased. Combined with the fact that GRANT was developed to represent "hard science" topics and now includes arts, humanities, and social sciences, this suggests that the relations must be augmented and perhaps reworked. This also requires reworking the set of path endorsements. In fact, an experimental set of path endorsements gave somewhat better performance for the Winter, 1986 version. The hit rate remained very high but the false positive rate dropped to 55.8%.

The partial matching algorithm, based on Tversky's contrast model, was not as effective as we had hoped in pruning agencies based on the total degree of overlap between proposals and agencies. In general, the false positive rate can be reduced but not without a corresponding reduction in the hit rate. The algorithm contributes little because in most cases, a proposal shares only one research topic with an agency. Since this overlap is usually found by semantic matching, best-first search will continue to he the heart of GRANT's problem-solving method, and path endorsements will receive more attention than tuning the partial matching algorithm. The next section describes an algorithm for learning path endorsements.

4.4 In Prospect: Learning Path Endorsements

The likelihood that an agency will fund a proposal depends on the path endorsement that characterizes the semantic match between them. Path endorsements as discussed above either support the proposition that the agency will fund the proposal, or detract from it, or their support for the proposition is unknown. In practice, GRANT's path endorsements are empirically ranked into six classes: *very likely, likely, maybe, unknown,* and *trash*. Detracting path endorsements belong to the class *trash*. The class *very likely* is reserved for atomic matches. Thus, semantic matches that support the proposition that an agency will fund the proposal are differentiated only by the classes *likely* and *maybe*.

We have developed an algorithm to assign a continuous weight to path endorsements, based on whether they find likely agencies or false positives. The algorithm learns from examples presented by a human tutor. Each example is a pair of nodes for which the tutor expects GRANT to find a semantic match. The algorithm generates a set of paths between these nodes from GRANT's knowledge base, and adjusts the weight of each path to favor short paths over long ones. After many iterations, short paths that are commonly found between training examples have high weights, relative to other paths.

The algorithm has been tested on small samples of examples and it has not yet been integrated with GRANT. In prospect, however, its principle advantage is that it learns the *empirical* worth of path endorsements, in contrast to our a priori efforts to categorize path endorsements as *likely* or *maybe*. Kjeldsen (1986) describes the algorithm in detail.

Two other extensions to GRANT should be mentioned. First, we have developed an "empty" version and will be experimenting with semantic matching in other domains. Second, we are generalizing the inference rule that underlies GRANT — "if an agency is interested in X then they will be interested in Y = R(X)" — to a logic for plausible inference in associative knowledge bases. This project is discussed in the next section.

4.5 Appendix 1

Relations for funding agencies:

1. The TITLE slot should contain a text string with full title that will include the Parent Agency, Department, and Program Name.
2. The UNIQUE-ID slot should contain a text string that is the unique number assigned by the Catalogue of Federal Domestic Assistance (CFDA).
3. The FUNDING-TYPE slot should contain the type of funding that is available, e.g., *project-grant, large-grant, small-grant, direct-loan, fellowship*, or *scholarship*.
4. The CONTACT slot should contain the name, address, and phone number of the person to contact for more information and applications.
5. The DEADLINES slot should contain the application and renewal deadlines for the program.
6. The DESCRIPTION slot should contain the abstract that is provided by the agency and describes their interests and motivations.
7. The TOPIC slot should contain one or more instances of the STUDY, MANAGE, EDUCATE, or ENGINEER frames.
8. The PURPOSE slot is optional for the top-level of a *funding-source* frame since it might be present in one of the values for the TOPIC slot.

Relations for defining research interests:

1. The OBJECT slot contain the person, place, process, or thing that is being studied.
2. The SUBJECT slot contain the particular filed of study that is to be applied to the *object*.
3. The FOCUS slot should contain the particular aspect of the *subject* that is being considered.
4. The DV slot should contain the *object* that is being studied.
5. The IV slot should contain the variables that whose effect upon the dependent variable are being studied.
6. The RV slot should contain one or more variables that are being studied.
7. The PURPOSE slot should contain the overall goal of the funding source.
8. The WHO-FOR slot should contain an instance of a social-group that will benefit from the proposed research and funding.
9. The SETTING slot should contain the place in which the *object* will be studied.

10. The LOCATION slot should contain a geographical place to which funding is restricted.

Relations for organizing knowledge in GRANT's knowledge base:

1. The CAUSES slot should contain a concept that has a causal association with the node.
2. The EFFECTS slot is used to represent relationships that are not necessarily causal but nonetheless present.
3. The HAS-COMPONENT slot should contain those things that make up the node. For example, one could say that a earthquake has-component shock-wave.
4. The HAS-MECHANISM slot is used to represent those processes that a concept might have. For example a seismology has-mechanism seismometer.
5. The HAS-PURPOSE slot is used to hold an instance of an action. For example, a seismometer has-purpose measure, with the object of the measure being shock-wave.

4.6 Appendix 2

Path Endorsements for the Knowledge Base is the rule set that is used in a bottom-up data driven search from proposal to funding source. Many of these traversal rules are effectively used to prune the number of potential nodes to expand. A SUCCESS-NODE is any node that can be found as a value for wither the TOPIC or PURPOSE slot of a funding-source.

- The class SELF has 1 traversal rule
  - Self - basically an identity rule for paths of length 0
- The class VERY-LIKELY includes 7 path endorsements, all atomic matches. For example,
  - X→ subject→ Y→ subject-of→ SUCCESS-NODE
  - X→ focus→ Y→ focus-of→ SUCCESS-NODE
- The class LIKELY has over 50 path endorsements representing semantic matches between a proposal and an agency that is likely to fund it. For example,
  - X→ subject→ Y→ isa→ Z→ subject-of→ SUCCESS-NODE
  - X→ subject→ Y→ component-of→ Z→ focus-of→ SUCCESS-NODE
  - X→ done-by→ Y→ does→ object-of→ SUCCESS-NODE
- The class MAYBE has 18 path endorsements. These represent semantic matches between a proposal and funding agencies that are somewhat less likely to fund the research, for example:
  - X→ focus→ Y→ subject-of→ Z→ subject-of→ SUCCESS-NODE
  - X→ object→ Y→ focus-of→ Z→ subject-of→ SUCCESS-NODE
  - X→ object→ Y→ object-of→ Z→ focus-of→ SUCCESS-NODE
- The class UNKNOWN accepts any path less than 6 links long
- The class of UNUSABLE paths prunes GRANT's search. Among these paths are any that contain a node with an extremely high branching factor (e.g., science, education). Specific pathways of the kind listed above include
  - STEP*→ isa→ example→ Y
  - STEP*→ subfield-of→ has-subfield→ Y
  - NOT(new-investigator)→ STEP*→ new-investigator
  - NOT(minority-student)→ STEP*→ minority-student
  - X→ object→ Y→ subject-of→ Z→ focus-of→ SUCCESS-NODE
  - X→ rv→ Y→ dv-of→ SUCCESS-NODE
  - X→ subject→ Y→ isa→ Z→ dv-of→ SUCCESS-NODE

5. Plausible Inference

This research is concerned with the formal underpinnings of common sense plausible inference, the ability to give plausible answers to arbitrary questions from a very large knowledge base of associated statements. The goal is to find one or more answers to a question by consulting the knowledge base, and to say which of the answers are most credible. This has been a goal of AI since its earliest days (McCarthy, 1958, 1968), and is now seeing a resurgence (Collins, 1978a,b; Lenat et al, 1986). The motivation for such work comes from the increasing realization that powerful AI programs will depend on very large knowledge bases. It will be necessary for the system to use the knowledge base to answer questions that were not anticipated at the time of its construction. To handle both the broad ranging nature of possible queries, and to make use of large amounts of knowledge in an efficient manner, it is expected that the use of heuristics, or plausible inference rules, as well as traditional truth-preserving ones, will be necessary.

Our research is directed by these concerns, as well as by a desire to bring a formalism to plausible reasoning similar to that enjoyed by deductive logic, so that systems using plausible reasoning need not have their semantics established on a case-by-case, ad hoc basis.

The most important question to be answered about plausible inference is how to judge its credibility. Since plausible inference need not be truth-preserving, some other semantic property besides truth must be the basis of judgments of credibility. We propose to develop a semantics for common sense plausible inference based on the associations that hold between the antecedents and consequents of inferences. Our approach is strongly motivated by evidence-based control: the credibility of a statement is represented by reasons *why* it may be false, reasons that can be used to control backtracking and retraction of plausible but false inferences.

Plausible inferences, unlike deductive inferences, need not be truth-preserving. The distinction is clear in a contrast between two rules of inference, modus ponens and abduction:

Modus ponens is truth-preserving: if A → B and A are true, B cannot be false. Abduction is a rule of plausible inference because A is a plausible conclusion given A → B and B, but this conclusion is not *guaranteed* to be true, as the conclusion B is in modus ponens.

Since rules of plausible inference do not make guarantees about the truth values of their conclusions, how are we to assess the *credibility* of conclusions of plausible inference? In the deductive case we associate credibility with the semantic property *truth*: true statements are credible, false statements are not. What semantic property of conclusions derived by plausible inference will be associated with credibility? We could use truth, since some conclusions of plausible inference have truth values. The problem is that rules of plausible inference make no guarantees about these truth values, as rules of deductive inference do. So the question remains: What properties of conclusions are preserved by rules of plausible inference and are the basis for judgments of credibility?

Truth is not the semantic property we seek to preserve in plausible inference. This is because of our abiding interest in uncertainty, the state of not knowing whether a proposition is true or false. Many attempts have been made to modify deductive logic to represent uncertainty, including modal logics, 3-valued logics, nonmonotonic logics, fuzzy logics, and probabilistic logic (Turner, 1984, Zadeh, 1975; Nilsson, 1984) Some of these approaches "sequester" uncertainty by introducing a new argument that represents the uncertainty but is itself true or false. Modal logics do this. Other approaches augment the values true and false; for example, three-valued logics add the value "unknown," and fuzzy logics introduce numeric arguments. Nonmonotonic logics go further and replace the notion of truth with one of *support*. Nonmonotonic formulations differ; in McDermott and Doyle's version, the notion of truth is generalized to *support* and falsity to lack of support (McDermott and Doyle, 1980).

Although uncertain statements are neither true nor false one can say a great deal more about them. Extensions to logic, however, say little. With the possible exception of nonmonotonic logic and dependency-directed backtracking, none of the extensions to logic enable us to say why we are uncertain and what we might do about it (de Kleer, et al, 1977). Shortly, we will discuss an alternative approach, but first we must address another common paradigm in AI for plausible inference and explain why we are avoiding it.

Much of the AI community favors probabilistic representations of uncertainty. We believe that, with one exception, the semantics of these representations are opaque. The exception is when the probabilities are relative frequencies, combined by Bayes' theorem. This case is akin to deductive inference in that a semantic property (relative frequency) is guaranteed to be preserved by a rule of inference (Bayes' theorem). Just as we associated credibility with truth in deductive inference, we can associate it with relative frequency in probabilistic inference. In both cases, we can guarantee that the credibility of a conclusion can be unambiguously determined. Unfortunately, the numbers used in knowledge systems are not relative frequencies. Until we know what they represent, we cannot know whether their intent or meaning is preserved by the functions that are used to combine them. The plethora of combining functions discussed in the AI literature suggests that no common interpretation of degrees of belief is available (Duda and Hart, 1976; Pearl, 1982; Shafer, 1976).

So we are led back to the question, if truth or relative frequency are not the basis of credibility when reasoning under uncertainty, what is? What properties of statements determine their credibility, and can we guarantee that these properties are preserved by inference rules? In Section 4 we saw that the credibility of inferences depends on the semantic associations on which they are based. For example, if a researcher is interested in VLSI layout, and a funding agency is interested in electronics, the fit between them is good and the agency is apt to fund the proposal. The semantic association between electronics and VLSI is "has-subfield," and it is the basis of this plausible inference:

$$\begin{array}{c}\text{interested-in(agency, electronics)}\\ \text{has-subfield (electronics, VLSI)}\\ \hline \text{interested-in(agency, VLSI)}\end{array}$$

In brief, degree of fit between two objects, X and Y, was defined to mean that some rule of plausible inference could be invoked to conclude interested-in(agency, Y) given interested-in(agency, X).

The GRANT system (Section 4) sets the stage for the current research. It is the first step toward a common sense plausible inference system as defined above – a program that answers arbitrary questions from a large, associative knowledge base. But GRANT does not, in fact, answer arbitrary questions. It answers the single question, "If a funding agency is interested in X, will it be interested in Y?" It can be generalized to a common sense plausible inference system as follows:

1. Assume that all questions are about *properties* of objects; for example, "Does Fido have fur," or "Is coughing caused-by bronchitis." Abbreviate such questions $R(O_1,O_2)$?; for example, caused-by(coughing,bronchitis)?.
2. The answer to $R(O_1,O_2)$? is yes if the knowledge base contains O_1 and O_2 connected by R. The answer is plausible if there is a rule of plausible inference of the form

$$\begin{array}{c} Q(O_3,O_2)? \\ R(O_3,O_1) \\ \hline R(O_1,O_2) \end{array}$$

and $Q(O_3, O_2)$? is plausible. For example, imagine asking a system, "Are gin-and-tonics intoxicating?" or, has-effect(gin-and-tonic, intoxication)? Assume that the objects gin-and-tonic and intoxication are *not* linked by has-effect in the knowledge base. The question can be answered, however, by plausible inference using the rule

$$\begin{array}{c} \text{has-component}(x,y)? \\ \text{has-effect}(y,z) \\ \hline \text{has-effect}(x,z) \end{array}$$

and the knowledge that gin-and-tonics contain alcohol and alcohol is intoxicating:

$$\begin{array}{c} \text{has-component(gin-and-tonic,alcohol)?} \\ \text{has-effect(alcohol,intoxication)} \\ \hline \text{has-effect(gin-and-tonic,intoxication)} \end{array}$$

Property inheritance in frame systems is a special case of this kind of inference. The rule for property inheritance is

$$\begin{array}{c} isa(X,Y) \\ R(Y,Z) \\ \hline R(X,Z) \end{array}$$

where R is any relation. For example, isa(collie,dog) and part-of(dog,fur) implies part-of(collie,fur). The approach we propose here allows us to infer the answers to questions based on semantic associations other than isa. Thus, the approach unifies several kinds of plausible inference, including causal inference (Weiss et al, 1977).

The model of plausible inference is not complete, however, since it lacks statements about the credibility of inferences drawn by plausible inference rules. Obviously, [illegible] do not intend to include rules that draw error[illegible]onclusions, but credibility is not guaranteed, as [illegible] logic, by plausible inference. We discussed how our rules implement a notion of credibility based on degree of fit, but this still does not guarantee credibility. We know of two general approaches to this problem. One is to attach to each conclusion a set of conditions that, if met, would increase its credibility. Collins, who developed this idea, calls these *certainty conditions* (Collins, 1978b). The other is to attach a set of conditions that, if met, would decrease credibility. We have called these *negative endorsements* (Cohen, 1984). From the standpoint of control, certainty conditions can guide a system to increase its belief and negative endorsements can help a system recover from errorful conclusions by pointing to reasons a conclusion might be wrong. Obviously, both are required for evidence-based control.

Given a set of rules of plausible inference, with reasons to believe and disbelieve their conclusions, we can engage in a range of common sense plausible inference tasks. Our proposed work thus involves several stages:

- Develop common sense plausible inference rules. These are based on semantic associations, so clearly we need a set of associations at the outset. We began with the associations in GRANT's knowledge base. Next, we generated all combinations of associations of the form

$$\begin{array}{c} A_1(x,y) \\ A_2(y,z) \\ \hline A_1(x,z) \end{array}$$

These can be filtered by case-semantic considerations: y must be a particular *kind* of object to fill the A_1 case of x, and z is also restricted by its relation to y. In many cases, though, z will not fill the A_1 case of x, and so a potential rule can be filtered out. Even with this filtering, GRANT's associations generated about 600 rules of plausible inference.

The rules are further pruned by automatically generating, from GRANT's knowledge base, examples of inferences made by the rules. Thus we can select empirically a set of rules that make a high proportion of truly plausible inferences.

- Endorse the rules. Given these rules it remains to specify the conditions under which they are more or less likely to generate plausible conclusions. This work remains to be done.
- Test the rules. Recently, Cohen et al. (1985) tested GRANT by comparing its performance again that of an expert. The same approach will be used to test our common sense plausible inference system both in the GRANT domain, for which we have a very large associative knowledge base, and in other associative domains such as causal reasoning.

Further extensions involve generalizing rules of plausible inference to include conjunctions, negations, and quantification. It will probably be easy to make these extensions given the propositional form of the rules as shown above. However, the inference mechanism that underlies GRANT is a tightly-controlled spreading activation. This has several advantages that are discussed in Cohen et al. (1985), so we want to maintain this approach in our proposed work. We currently know how to model the plausible inference rules above as spreading activation, but we are not sure how to extend this approach when the rules include conjunctions, negations, and quantifiers.

The result of this work will be a set of rules of inference whose plausibility for the GRANT knowledge base has been discovered empirically and confirmed by comparison with expert judgment. We hope, however, to go beyond this result to explore the reasons WHY the rules discovered are plausible, in what situations they would not be plausible, etc. To this end, we plan to extend our work on plausible reasoning to domains that already have algorithmic solutions (e.g. deadlock prevention in operating systems). The use of an algorithmic solution as a foil for plausible ones will aid in the discovery of formal characterizations of the nature of plausible inference rules.

REFERENCES

[1] Bonisonne, P. 1985. Reasoning with uncertainty in expert systems. *International Journal of Man-Machine Studies*, **22:3**.

[2] Clancey, W.S., 1984. Classification problem solving. *Proceedings of the AAAI,* p.49.

[3] Clancey, W. 1983. The advantages of abstract control knowledge in expert systems design. In *Proceedings of the Third National Conference on Artificial Intelligence.*

[4] Cohen, P. and Stanhope, P. 1986. Finding research funds with the GRANT system. *Proc. 6th International Workshop on Expert Systems and Their Applications,* April 28-30, 1986, Avignon, France.

[5] Cohen, P. , Davis, A. , Day, D. , Greenberg, M. , Kjeldsen, R. , Lander, S. , and Loiselle, C. 1985. *Representativeness and Uncertainty in Classification Systems AI Magazine,* 6(3), 136-149.

[6] Cohen, P. and Gruber, T. 1985. Reasoning about uncertainty: A knowledge representation perspective. Pergamon Infotech State of the Art Report.

[7] Cohen, P. 1985. *Heuristic reasoning about uncertainty: An AI approach.* London: Pitman Advanced Publishing. London.

[8] Cohen, P. 1984. Progress Report on the Theory of Endorsements: A Heuristic Approach to Reasoning About Uncertainty. *COINS Technical Report 84-15.*

[9] Cohen, P. and Grinberg, M. 1983. A theory of heuristic reasoning about uncertainty. *The AI Magazine.* Summer, 1983.

[10] Cohen, P., and Feigenbaum, E. 1982. *The Handbook of Artificial Intelligence. Volume 3.* Los Altos, CA:William F. Kaufmann, Inc.

[11] Collins, A. 1978a. Fragments of a theory of human plausible reasoning. (D. Waltz, Ed.) *Theoretical Issues in Natural Language Processing.* Urbana, IL: University of Illinois.

[12] Collins, A. 1978b. *Human plausible reasoning.* Cambriuge, MA: Bolt, Beranek and Newman, Inc., Report No. 3810.

[13] Davis, R. 1985. Interactive transfer of expertise. In *Rule-based expert systems,* B. Buchanan and E. Shortliffe, (Eds.) Addison-Wesley.

[14] de Kleer, Johan; Doyle, Jon; Steele, Guy L., Jr.; Sussman, Gerald Jay, 1977. AMORD: Explicit Control of Reasoning.*Proc. Symposium on Artificial Intelligence and Programming Languages,* SIGPLAN Notices 12(8), and SIGART Newsletter, 64, 116-125.

[15] Doyle, J., 1979. A truth maintenance system. *Artificial Intelligence,* 13, 81-132.

[16] Duda, R. O. , Hart, P. E. , and Nilsson, N. 1976. Subjective Bayesian methods for rule-based inference systems. *Technical note 124,* AI Center, SRI International, Menlo Park, CA.

[17] Hayes-Roth, B. 1985. A blackboard architecture for control. *Artificial Intelligence,* Vol. 26, pp. 251-321.

[18] Hayes-Roth, F., 1978. The role of partial and best matches in knowledge systems. *Pattern Directed Inference Systems,* Waterman, D., Hayes-Roth, D., and Lenat, D. (Eds). Academic Press.

[19] Hayes-Roth, F. and Lesser, V. 1977. Focus of attention in the Hearsay-II speech understanding system. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.*

[20] Howe, A., Cohen, P. Comparing alternatives in decision making. *EKSL Memo*, University of Massachusetts, January 1986.

[21] Kahneman, D. and Tversky, A. 1982. Judgment under uncertainty; heuristics and biases. *Judgment under uncertainty; heuristics and biases*, D. Kahneman, P. Slovic, A. Tversky (Eds.) Cambridge: Cambridge University Press.

[22] Kjeldsen, Rick, 1986. Learning traversal rules for semantic nets. *EKSL Working Paper.*

[23] Lenat, Doug; Prakash, Mayank; and Shepherd, Mary CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks. *AI Magazine,* 6(4), 65-85.

[24] McCarthy, John, 1958. Mechanization of Thought Processes. *Proc. Symposium, National Physics Laboratory,* 1, 77-84, London.

[25] McCarthy, John, 1968. Programs with Common Sense. *Semantic Information Processing,* 403-418, edited by M. Minsky, Cambridge, MA: The MIT Press.

[26] McDermott, D. and Doyle, J. Non-monotonic Logic I. *Artificial Intelligence 13,* 27-39.

[27] Nilsson, Nils J. , 1984. Probabilistic Logic. *SRI AI Center Technical Note 321,* SRI International, Menlo Park, CA.

[28] Patel, V. and Groen, G. 1986. Knowledge based solution strategies in medical reasoning. *Cognitive Science,* Vol. 10, pp. 91-116.

[29] Payne, J., Braunstein, M., Carroll, J. Exploring predecisional behavior: an alternative approach to decision research. *Organizational Behavior and Human Performance*, 1978, Vol.22, pp. 17-44.

[30] Pearl, J., Leal, A., Saleh, J. GODDESS: a goal-dirested decision structuring system. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1982, Vol.4, pp. 250-262.

[31] Pearl, J. 1982, Reverend Bayes on Inference Engines: a Distributed Hierarchical Approach. *Proceedings of the National Conference on Artificial Intelligence,* Pittsburgh, PA, 133-136.

[32] Sage, A. & White, C. ARIADNE: a knowledge-based interactive system for planning and decision support. *IEEE Transactions on Systems, Man, and Cybernetics*, 1984, Vol.14, pp. 35-47.

[33] Shafer, G. 1976. *A Mathematical Theory of Evidence.* Princeton: Princeton University Press.

[34] Shortliffe, E. and Buchanan, B. 1975. A model of inexact reasoning in medicine. *Mathematical Biosciences,* Vol. 23, pp. 351-379.

[35] Svenson, O. Process descriptions of decision making. *Organizational Behavior and Human Performance*, 1979, Vol.23, pp. 86-112.

[36] Szolovits, P. and Pauker, S.G. 1978. Categorical and Probabilistic Reasoning in Medical Diagnosis. *Artificial Intelligence,* V.11, pp. 115-144

[37] Turner, Raymond, 1984. *Logics for Artificial Intelligence.* Chichester: Ellis Howard Limited.

[38] Weiss, S. , Kulikowski, C. , and Safir, A. 1977. A model-based consultation system for the long-term management of glaucoma. *IJCAI 5,* 826-832.

[39] Wesley, L.P. 1983. Reasoning about control: the investigation of an evidential approach. *Proceedings IJCAI-83,* pp. 203-206.

[40] Zadeh, L.A. 1975. Fuzzy logic and approximate reasoning, *Synthese,* Vol. 30, pp. 407-428.

PROGRESS IN REASONING WITH INCOMPLETE AND UNCERTAIN INFORMATION[*]

Part I: Uncertainty Calculi: How Many, When, and Why?
Part II: A Hierarchical Model Paradigm for Reasoning by Analogy
Part III: Theories of Non-Monotonic Reasoning and Reason Maintenance

Piero P. Bonissone, Gilbert B. Porter III, Allen L. Brown, Jr.
General Electric Company
Corporate Research and Development
P.O. Box 8
Schenectady, New York 12301

ABSTRACT

This paper summarizes our research efforts in the area of Reasoning with Incomplete and Uncertain Information, and is organized into three parts covering reasoning with uncertainty, reasoning by analogy, and reasoning with incompleteness. Part I, entitled *Uncertainty Calculi: How Many, When, and Why?*, is a collection of two papers describing the evolution of an architecture for reasoning with uncertainty. The first paper of this collection, entitled *Selecting Uncertainty Calculi and Granularity: An Experiment in Trading-off Precision and Complexity*, describes the experiments that led to the derivation of equivalence classes among the (apparently) different uncertainty calculi as a function of the input granularity. The second paper, entitled *Summarizing and Propagating Uncertain Information with Triangular Norms*, describes an architecture for reasoning with uncertainty, which is organized in three layers: representation, inference, and control. The representation layer describes the structure required to capture information used in the inference layer and meta-information used in the control layer. The inference layer defines uncertainty calculi based on Triangular norms (T-norms), intersection operators whose truth functionality entails low computational complexity. The control layer specifies the policy selection for the different calculi used in the inference layer, based on their meanings, properties, and contextual information. Conflicts and ignorance measurements are also proposed.

which is based on a multi-staged decomposition; the knowledge representation scheme which uses a hierarchy of models that are ordered by complexity; the search strategy for dynamically creating a domain model for the current goal, and the global control method for forming an analogy. The supporting model paradigm is then described in detail and a few preliminary results are noted.

Part III, entitled *Theories of Non-Monotonic Reasoning and Reason Maintenance*, is a collection of two papers describing the evolution of the theory and the algorithm for reasoning with incomplete information. The first paper of this collection, entitled *Modal Propositional Semantics for Reason Maintenance Systems*, defines a propositional dynamic logic of derivation (PDLD). PDLD is a specification logic in which to express declarative control. This is achieved by characterizing the mental states of a reasoning agent attempting to reason with respect to some logic theory. The second paper, entitled *Reason Maintenance from a Lattice-Theoretic Point of View*, provides a mathematical framework (lattice) in which assumption-based justifications (ATMS) and non-monotonic justifications can be directly and transparently described. From this formulation it is possible to derive algorithms that support efficient revision of beliefs, as a reasoning agent changes its assumptions and/or its constraints on beliefs.

[*] This work was partially supported by the Defense Advanced Research Projects Agency (DARPA) contract F30602-85-C-0033. Views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official opinion or policy of DARPA or the U.S. Government.

Part I: Uncertainty Calculi: How Many, When, and Why?

Table of Contents

Selecting Uncertainty Calculi and Granularity: An Experiment in Trading-off Precision and Complexity

Summarizing and Propagating Uncertain Information with Triangular Norms

Part II: MONAD: A Hierarchical Model Paradigm for Reasoning by Analogy

Table of Contents

Part III: Theories of Non-Monotonic Reasoning and Reason Maintenance

Table of Contents

Modal Propositional Semantics for Reason Maintenance Systems

Reason Maintenance from a Lattice-Theoretic Point of View

SELECTING UNCERTAINTY CALCULI AND GRANULARITY: AN EXPERIMENT IN TRADING-OFF PRECISION AND COMPLEXITY

Piero P. Bonissone and K.S. Decker

ABSTRACT

The management of uncertainty in expert systems has usually been left to *ad hoc* representations and rules of combinations lacking either a sound theory or clear semantics. The objective of this paper is to establish a theoretical basis for defining the syntax and semantics of a small subset of calculi of uncertainty operating on a given term set of linguistic statements of likelihood. Each calculus is defined by specifying a negation, a conjunction and a disjunction operator. Families of Triangular norms and conorms constitute the most general representations of conjunction and disjunction operators. These families provide us with a formalism for defining an infinite number of different calculi of uncertainty. The term set will define the uncertainty *granularity*, i.e. the finest level of distinction among different quantifications of uncertainty. This granularity will limit the ability to differentiate between two similar operators. Therefore, only a small finite subset of the infinite number of calculi will produce notably different results. This result is illustrated by two experiments where nine and eleven different calculi of uncertainty are used with three term sets containing five, nine, and thirteen elements, respectively. Finally, the use of context dependent rule set is proposed to select the most appropriate calculus for any given situation. Such a rule set will be relatively small since it must only describe the selection policies for a small number of calculi (resulting from the analyzed trade-off between complexity and precision).

INTRODUCTION

The aggregation of uncertain information (facts) is a recurrent need in the reasoning process of an expert system. Facts must be aggregated to determine the degree to which the premise of a given rule has been satisfied, to verify the extent to which external constraints have been met, to propagate the amount of uncertainty through the triggering of a given rule, to summarize the findings provided by various rules or knowledge sources or experts, to detect possible inconsistencies among the various sources, and to rank different alternatives or different goals.

In a recent survey of reasoning with uncertainty [1-3], it is noted that the presence of uncertainty in reasoning systems is due to a variety of sources: the *reliability* of the information, the inherent *imprecision* of the representation language in which the information is conveyed, the *incompleteness* of the information, and the *aggregation* or summarization of information from multiple sources.

The existing approaches surveyed in that study are divided into two classes numerical and symbolic representations. The numerical approaches generally tend to impose some restrictions upon the type and structure of the information, e.g. mutual exclusiveness of hypotheses, conditional independence of evidence, etc. These approaches represent uncertainty as a precise quantity (scalar or interval) on a given scale. They require the user or expert to provide a *precise* yet *consistent* numerical assessment of the uncertainty of the atomic data and of their relations. The output produced by these systems is the result of laborious computations, guided by well-defined calculi, and *appears* to be equally precise. However, given the difficulty in consistently eliciting such numerical values from the user, it is clear that these models of uncertainty require an unrealistic level of precision that does not actually represent a real assessment of the uncertainty.

Models based on symbolic representations, on the other hand, are mostly designed to handle the aspect of uncertainty derived from the *incompleteness* of the information. However, they are generally inadequate to handle the case of *imprecise* information, since they lack any measure to quantify confidence levels.

The objective of this paper is to examine the various calculi of uncertainty and to define a rationale for their selection. The number of calculi to be considered will be a function of the uncertainty granularity, i.e., the finest level of distinction among different quantifications of uncertainty that adequately represent the user's discriminating perception. To accomplish this objective we will establish the theoretical framework for defining the syntax of a small subset of calculi of uncertainty operating on a given term set of linguistic statements of likelihood.

In Section 2 of this paper, the negation, conjunction, and disjunction operators that form the various calculi of uncertainty are described in terms of their most generic representation: families of functions (Triangular norms and conorms) satisfying the basic axioms expected of set operations such as intersection and union.

In Section 3, linguistic variables defined on the [0,1] interval are interpreted as verbal probabilities and their semantics are represented by fuzzy numbers. The term set of linguistic variables defines the granularity of the confidence assessment values that can be consistently expressed by users or experts. A nine element term set is given as an example.

Section 4 describes two experiments, consisting of evaluating nine and eleven different T-norms with the elements of three different term sets containing five, nine, and thirteen elements, respectively. A review of the techniques required to implement the experiment is also provided. The review covers the implementation of the extension principle (a formalism that enables crisply defined functions to be evaluated with fuzzy-valued arguments) and describes linguistic approximation (a process required to map the result of the aggregation of two elements of the term set back into the term set).

Section 5 shows the results of computing the closures of selected operators on common term sets. An analysis of the results of these experiments shows the equivalence of some calculi of uncertainty that produce indistinguishable results within the granularity of a given term set. Possible interpretations for the calculi that produce notably different results are suggested in the last part of this section.

Section 6 illustrates the conclusions of this paper.

AGGREGATION OPERATORS

According to their characteristics, there are three basic classes of aggregation: *conjunctions, trade-offs*, and *disjunctions*. Dubois and Prade [4] have shown that Triangular norms (T-norms), averaging operators, and Triangular conorms (T-conorms) are the most general families of binary functions that respectively satisfy the requirements of the conjunction, trade-off, and disjunction operators. T-norms and T-conorms are two-place functions from [0,1]x[0,1] to [0,1] that are monotonic, commutative and associative. Their corresponding boundary conditions satisfy the truth tables of the logical AND and OR operators. Averaging operators are symmetric and idempotent but are not associative. They do not have a corresponding logical operator since, on the [0,1] interval, they are *located* between the conjunctions and the disjunctions.

The generalizations of conjunctions and disjunctions play a vital role in the management of uncertainty in expert systems: they are used in evaluating the satisfaction of premises, in propagating uncertainty through rule chaining, and in consolidating the same conclusion derived from different rules. More specifically, they provide the answers to the following questions:

— When the premise is composed of multiple clauses, how can we aggregate the degree of certainty x_i of the facts matching the clauses of the premise? i.e., what is the function $T(x_1,\ldots,x_n)$ that determines x_p, the degree of certainty of the premise?

— When a rule does not represent a logical implication, but rather an empirical association between premise and conclusion, how can we aggregate the degree of satisfaction of the premise x_p with the strength of the association s_r? i.e., what is the function $G(x_p,s_r)$ that propagates the uncertainty through the rule?

— When the same conclusion is established by multiple rules with various degrees of certainty $y_1,\ldots,y_m$, how can we aggregate these contributions into a final degree of certainty? i.e., what is the function $S(y_1,\ldots,y_m)$ that consolidates the certainty of that conclusion?

The following three subsections describe the axiomatic definitions of the conjunction, disjunction, and negation operators.

Conjunction and Propagation Using Triangular Norms

The function $T(a,b)$ aggregates the degree of certainty of two clauses in the same premise. This function performs an *intersection* operation and satisfies the conditions of a Triangular norm (T-norm):

$T(0,0) = 0$ [boundary]
$T(a,1) = T(1,a) = a$ [boundary]
$T(a,b) \leq T(c,d)$ if $a \leq c$ and $b \leq d$ [monotonicity]
$T(a,b) = T(b,a)$ [commutativity]
$T(a,T(b,c)) = T(T(a,b),c)$ [associativity]

Although defined as two-place functions, the T-norms can be used to represent the intersection of a larger number of clauses in a premise. Because of the associativity of the T-norms, it is possible to define recursively $T(x_1,\ldots,x_n,x_{n+1})$, for $x_1,\ldots,x_{n+1} \in [0,1]$, as:

$$T(x_1,\ldots,x_n,x_{n+1}) = T(T(x_1,\ldots,x_n),x_{n+1})$$

A special case of the conjunction is the *detachment* function $G(x_p,s_r)$, which attaches a certainty measure to the conclusion of a rule. This measure represents the aggregation of the certainty value of the premise of the rule x_p (indicating the degree of fulfillment of the premise) with the strength of the rule s_r (indicating the degree of causal implication or empirical association of the rule). This function satisfies the same conditions of the T-norm (although it does not need to be commutative.)

Disjunction Using Triangular Conorms

The function $S(a,b)$ aggregates the degree of certainty of the (same) conclusions derived from two rules. This function performs a *union* operation and satisfies the conditions of a Triangular conorm (T-conorm):

$S(1,1) = 1$ [boundary]
$S(0,a) = S(a,0) = a$ [boundary]
$S(a,b) \leq S(c,d)$ if $a \leq c$ and $b \leq d$ [monotonicity]
$S(a,b) = S(b,a)$ [commutativity]
$S(a,S(b,c)) = S(S(a,b),c)$ [associativity]

A T-conorm can be extended to operate on more than two arguments in a manner similar to the extension for the T-norms. By using a recursive definition, based on the associativity of the T-conorms, we can define:

$$S(y_1,\ldots,y_m,y_{m+1}) = S(S(y_1,\ldots,y_m),y_{m+1})$$

Relationships Between T-norms and T-conorms

For suitable negation operations $N(x)$, such as N(x)=1-x, T-norms T and T-conorms S are duals in the sense of the following generalization of DeMorgan's Law:

$$S(a,b) = N(T(N(a),N(b)))$$
$$T(a,b) = N(S(N(a),N(b)))$$

This duality implies that the extensions of the intersection and union operators cannot be independently defined and they should, therefore, be analyzed as DeMorgan triples $(T(.,.), S(.,.), N(.))$ or, for a common negation operator like $N(a) = 1-a$, as DeMorgan pairs $(T(.,.), S(.,.))$ [1] Some typical pairs of T-norms $T(a,b)$ and their dual T-conorms $S(a,b)$ are the following:

[1] Quinlan [32] raised a criticism regarding the use of the *min* operator, considered an *optimistic* intersection operator, and the *max* operator, considered a pessimistic union operator. The use of this pair of operators is actually not a contradiction, since they are their respective DeMorgan duals.

$$\begin{aligned}
T_0(a,b) &= \min(a,b) \text{ if } \max(a,b) = 1\\
&= 0 \text{ otherwise}\\
T_1(a,b) &= \max(0, a+b-1)\\
T_{1.5}(a,b) &= (a,b)/[2-(a+b-ab)]\\
T_2(a,b) &= ab\\
T_{2.5}(a,b) &= (a,b)/(a+b-ab)\\
T_3(a,b) &= \min(a,b)
\end{aligned}$$

$$\begin{aligned}
S_0(a,b) &= \max(a,b) \text{ if } \min(a,b) = 0\\
&= 1 \text{ otherwise}\\
S_1(a,b) &= \min(1, a+b)\\
S_{1.5}(a,b) &= (a+b)/(1+ab)\\
S_2(a,b) &= a + b - ab\\
S_{2.5}(a,b) &= (a+b-2ab)/(1-ab)\\
S_3(a,b) &= \max(a,b)
\end{aligned}$$

These operators are ordered as follows:

$$T \le T_1 \le T_{1.5} \le T_2 \le T_{2.5} \le T_3$$

$$S_3 \le S_{2.5} \le S_2 \le S_{1.5} \le S_1 \le S_0$$

An analysis of their properties can be found elsewhere [5]. The Appendix provides a summary of such properties.

Notice that any T-norm $T(a,b)$ and any T-conorm $S(a,b)$ are bounded by:

$$T_0(a,b) \le T(a,b) \le T_3(a,b)$$

$$S_3(a,b) \le S(a,b) \le S_0(a,b)$$

This set of boundaries implies that the averaging operators, used to represent trade-offs are located between the MIN operator T_3 (upper bound of T-norms) and the MAX operator S_3 (lower bound of T-conorms). These limits have a very intuitive explanation since, if compensations are allowed in the presence of conflicting goals, the resulting trade-off should lie between the most optimistic lower bound and the most pessimistic upper bound, i.e., the worst and best local estimates. Averaging operators are symmetric and idempotent, but, unlike T-norms and T-conorms, are not associative. A detailed description of averaging operators can be found elsewhere [4].

Negation Operators and Calculi of Uncertainty

The selection of a T-norm, Negation operator and T-conorm defines a particular *calculus* of uncertainty. The axioms for a Negation operator have been discussed by several researchers [6-8]. The axioms are:

$N(0) = 1$	[boundary]
$N(1) = 0$	[boundary]
$N(x) > N(y)$ if $x < y$	[strictly monotonic decreasing]
$N(a) = \lim_{x \to a} N(x)$	[continuity]
$N(N(x)) = x$	[involution]

Bellman and Giertz [6] have shown that the above axioms do not uniquely determine a negation operator. In addition to the abcve axioms they imposed a highly constraining *symmetry* condition, i.e., "...A certain change in the truth value of $\mu(S)$ of S [i.e., x] should have the same effect on the acceptance of "not S" [i.e., $N(x)$] regardless of the value of $\mu(S)$ [i.e., x]". Only with this (sometimes questionable) axiom is it possible to determine uniquely $N(x) = 1 - x$. Klement [9] provides an excellent summary of equivalences among the various sets of axiomatic definitions of conjunction, disjunction and negation operators.

It is important to notice that, like intuitionistic logic, most[2] multiple-valued logics defined by selecting the three operators (T (.,.), S (.,.), N(.)) disregard the *excluded middle* law and its DeMorgan's dual *law of non-contradiction*. The historic reason for this departure from classical logic goes back to Godel's proof of incompleteness: if it might not be possible to derive a true theorem from a given set of axioms, i.e., if it is possible for a theorem to be logically uncertain, it would then be necessary to consider at least three logic values: *true, false, unknown*. Therefore a statement could be something other than *true or false* and the excluded middle law does not apply.

The requirements of distributivity (or idempotency) *uniquely* determine the conjunction and disjunction operators to be the *min* (T_3) and *max* (S_3) operators [6,11]. This DeMorgan triple, (T_3,S_3,1 − ()), was first used in Łukasiewicz *Aleph-1* multiple-valued logics and has been widely adopted in fuzzy logic [12-13]. Dubois and Prade [14] have shown that the DeMorgan triple (T_1,S_1,1−()) satisfies[3] the excluded middle but is not distributive. They have also demonstrated that the distributivity property is mutually exclusive[4] with the axiom of the excluded middle.

[2] The only multiple-valued logics that satisfy the excluded middle are those defined by (T(.,.), S(.,.), N(.)), where the three operators were derived from the *same* generator. The additive generator of a T-norm is a function f that is continuous, strictly decreasing on [0,1], and satisfies the boundary conditions: $f(0)=b_0 \le \infty$ and $f(1) = 0$. Then any continuous Archimedean T-norm [10] $T(a,b)$ can be defined by

$$T(a,b) = f^*(f(a)+f(b))$$

where f^* is a function defined on $[0,\infty]$ by

$$\begin{aligned} f^*(x) &= f^{-1}(x) \text{ for } x \in [0,b_0]\\ &= 0 \text{ for } x \in [b_0,\infty] \end{aligned}$$

and f^{-1} is the inverse function of f. The generator of a negation operator is a function t that is continuous, increasing and satisfies the boundaries conditions: $t(0)=0$ and $t(1)<\infty$. Then any negation operator $N(x)$ can be defined by:

$$N(x) = t^{-1}(t(1) - t(x))$$

The T-norm will have the same generator if: $f(x) = t(1)-t(x)$. The T-conorm will have the same generator if derived from the T-norm using the DeMorgan duality condition [5,8].

[3] For this triple, the common generator is $t(x)=x$.

[4] The *min* and *max* operators, which form the only pair satisfying distributivity, *cannot* be defined by any additive generator. Thus there is no a DeMorgan triple, based on the these two operators and a negation operator, in which all three operators have a common generator

In most expert systems, a common selection of functions is:

$$CON\ UNCTION = T(a,b) = T_3(a,b) = min(a,b)$$

$$WEIGHTING = G(a,b) = T_2(a,b) = ab$$

$$DISJUNCTION = S(a,b) = S_3(a,b) = max(a,b)$$

$$NEGATION = N(a) = 1-a$$

Families of T-norms and T-conorms

Sometimes it is desirable to blend some of the previously described T-norm operators in order to smooth some of their effects. While it is always possible to generate a linear combination of two operators, in most cases this would imply giving up the associativity property. However, associativity is the most crucial property of the T-norms [10,15] since it allows the decomposition of multiple-place functions in terms of two-place functions. The correct solution is to find a family of T-norms that ranges over the desired operators. The proper selection of a parameter will then define the intermediate operator with the desired effect while still preserving associativity.

There are at least six families of T-norms $T_x(a,b,p)$ with their dual[5] T-conorms $S_x(a,b,p)$. The value of the subscript x will denote the family of norms; p, the third argument of each norm, will denote the parameter used by the corresponding family.

YAGER: $T_Y(a,b,q) = 1 - \text{MIN}\ \{1, [(1-a)^q + (1-b)^q]^{1/q}\}$	for $q > 0$
YAGER: $S_Y(a\ b,q) = \text{MIN}\ \{1, (a^q + b^q)^{1/q}\}$	for $q > 0$
DUBOIS: $T_D(a,b,\alpha) = (ab)/\text{MAX}\ \{a,b,\alpha\}$	for $\alpha \in [0,1]$
DUBOIS: $S_D(a,b,\alpha) = [a+b-ab - \text{MIN}\ \{a,b,(1-\alpha)\}]/\text{MAX}\ \{(1-a), (1-b),\alpha\}$	for $\alpha \in [0,1]$
HAMACHER: $T_H(a,b,\gamma) = (ab)/[\gamma+(1-\gamma)(a+b-ab)]$	for $\gamma \geq 0$
HAMACHER: $S_H(a,b,\gamma) = [a+b+(\gamma-2)ab]/[1+(\gamma-1)ab]$	for $\gamma \geq 0$
SCHWEIZER: $T_{Sc}(a,b,p) = \text{MAX}\ \{0, (a^{-p} + b^{-p}-1)\}^{-1/p}$	for $p \in [-\infty,\infty]$
SCHWEIZER: $S_{Sc}(a,b,p) = 1 - \text{MAX}\ \{0, [(1-a)^{-p}+(1-b)^{-p}-1]\}^{-1/p}$	for $p \in [-\infty,\infty]$
FRANK: $T_F(a,b,s) = \text{Log}_s\ [1+(s^a-1)(s^b-1)/(s\ -1)\ ]$	for $s > 0$
FRANK: $S_F(a,b,s) = 1 - \text{Log}_s\ [1+(s^{(1-a)}-1)(s^{1-b)}-1)/(s\ -1)\ ]$	for $s > 0$
SUGENO: $T_{Su}(a,b,\lambda) = \text{MAX}\ \{0, (\lambda+1)(a+b-1) -\lambda ab\}$	for $\lambda \geq -1$
SUGENO: $S_{Su}(a,b,\lambda) = \text{MIN}\ \{1, a+b-\lambda.a.b\}$	for $\lambda \geq -1$

The above families of T-norms and T-conorms are individually described in the literature [5,15-20].

The following table indicates the value of the parameter for which the above families of norms reproduce the most common T-norms $\{T_0, \ldots, T_3\}$.

Table 1

RANGES OF THE SIX PARAMETRIZED FAMILIES OF T-NORMS

$T_Y(a,b,q)$	$T_D(a,b,\alpha)$	$T_H(a,b,\gamma)$	$T_{Sc}(a,b,p)$	$T_F(a,b,s)$	$T_{Su}(a,b,\lambda)$	T-norm
q	α	γ	p	s	λ	
$\| \rightarrow 0^+$	.	$\| \rightarrow \infty$	$\| \rightarrow -\infty$	.	$\| \rightarrow \infty$	T_0
$\| 1$	.	$\| .$	$\| -1$	$\| \rightarrow \infty$	$\| 0$	T_1
$\| .$	.	$\| 2$	$\| .$	$\| .$	$\| .$	$T_{1.5}$
$\| .$	$\| 1$	$\| 1$	$\| \rightarrow 0$	$\| \rightarrow 1$	$\| -1$	T_2
$\| .$	$\| .$	$\| 0$	$\| .$	$\| .$	.	$T_{2.5}$
$\| \rightarrow \infty$	$\| 0$	.	$\| \rightarrow \infty$	$\| \rightarrow 0^+$	.	T_3

The vertical bars | used in Table 1 indicate the legal ranges of each parameter. The table for the T-conorms is identical to the above except for the header, where the families of T-norms are replaced by the corresponding families of T-conorms, and the last column, where the T-norms are replaced by their respective dual T-conorms, i.e., T_0 by S_0, etc.

LINGUISTIC VARIABLES DEFINED ON THE INTERVAL [0,1]

These families of norms can specify an infinite number of calculi that operate on arguments taking *real number* values on the [0,1] interval. This *fine-tuning* capability would be useful if we needed to compute, with a high degree of precision, the results of aggregating information characterized by very precise measures of its uncertainty. However, when users or experts must provide these measures, an assumption of *fake precision* must usually be made to satisfy the requirements of the selected calculus.

Szolovits and Pauker [21] noted that "...while people seem quite prepared to give qualitative estimates of likelihood, they are often notoriously unwilling to give precise numerical estimates to outcomes." This seems to indicate that any scheme that relies on the user providing *consistent*

[5] The dual T-conorms are obtained from the T-norm by using the generalized DeMorgan's Law with negation defined by $N(x)=1-x$. This negation operator, however, is not unique as illustrated by Lowen [7]

and *precise numerical* quantifications of the confidence level of his/her conditional or unconditional statements is bound to fail.

It is instead reasonable to expect the user to provide *linguistic* estimates of the likelihood of given statements. The experts and users would be presented with a verbal scale of certainty expressions that they could then use to describe their degree of certainty in a given rule or piece of evidence. Recent psychological studies have shown the feasibility of such an approach: "...A verbal scale of probability expressions is a compromise between people's resistance to the use of numbers and the necessity to have a common numerical scale" [22].

Linguistic probabilities offer another advantage. When dealing with subjective assessment of probability, it has been observed [23] that conservatism is consistently present among the suppliers of such assessments. The subjects of various experiments seem to stick to the original (a priori) assessments regardless of new amount of evidence that should cause a revision of their belief. In a recent experiment [24], linguistic probabilities have been compared with numerical probabilities to determine if the observed conservatism in the belief revision was a phenomenon intrinsic in the perception of the events or due to the type of representation (i.e., numerical rather than verbal expressions). The results indicate that people are much closer to the optimal Bayesian revision when they are allowed to use linguistic probabilities.

Each linguistic likelihood assessment is internally represented by fuzzy intervals, i.e., fuzzy numbers. A fuzzy number is a fuzzy set defined on the real line. In this case, the membership function of a fuzzy set defined on a truth space, i.e., the interval [0,1], could be interpreted as the *meaning* of a label describing the degree of certainty in a linguistic manner [25-26]. During the aggregation process, these fuzzy numbers will be modified according to given combination rules and will generate another membership distribution that could be mapped back into a linguistic term to the user's convenience or to maintain closure. This process, referred to as linguistic approximation, has been extensively studied [27-28] and will be briefly reviewed in Section 4.2.

Example of a Term Set of Linguistic Probabilities

Let us consider the following term set L_2:

{*impossible extremely_unlikely very_low_chance small_chance it_may meaningful_chance most_likely extremely_likely certain*}

Each element E_i in the above term set represents a statement of linguistic probability or likelihood. The semantics of each element E_i are provided by a fuzzy number N_i defined on the [0,1] interval. A fuzzy number N_i can be described by its continuous membership function $\mu_{N_i}(x)$, for $x \in [0,1]$.

A computationally more efficient way to characterize a fuzzy number is to use a parametric representation of its membership function. This parametric representation [26] is achieved by the 4-tuple $(a_i, b_i, \alpha_i, \beta_i)$. The first two parameters indicate the interval in which the membership value is 1.0; the third and fourth parameters indicate the left and right *width* of the distribution. Linear functions are used to define the slopes. Therefore, the membership function $\mu_{N_i}(x)$, of the fuzzy number $N_i = (a_i, b_i, \alpha_i, \beta_i)$ is defined as follows:

$$
\begin{aligned}
\mu_{N_i}(x) &= 0 && \text{for } x < (a_i - \alpha_i) \\
&= (1/\alpha_i)(x - a_i + \alpha_i) && \text{for } x \in [(a_i - \alpha_i), a_i] \\
&= 1 && \text{for } x \in [a_i, b_i] \\
&= (1/\beta_i)(b_i + \beta_i - x) && \text{for } x \in [b_i, (b_i + \beta_i)] \\
&= 0 && \text{for } x > (b_i + \beta_i)
\end{aligned}
$$

Figure 1 shows the membership distribution of the fuzzy number $N_i = (a_i, b_i, \alpha_i, \beta_i)$.

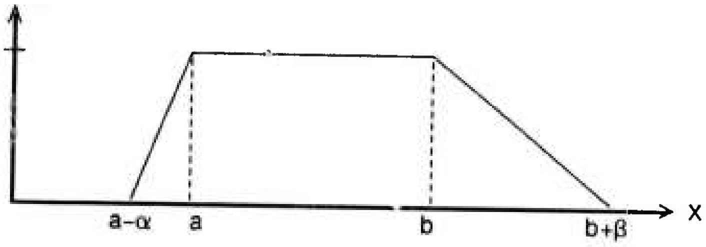


Figure 1. Membership Distributions of $N_i = (a_i, b_i, \alpha_i, \beta_i)$.

The following table indicates the semantics of the proposed term set L_2:

TABLE 2

THE NINE ELEMENT TERM SET L_2

impossible	(0 0 0 0)
extremely_unlikely	(.01 .02 .01 .05)
very_low_chance	(.1 .18 .06 .05)
small_chance	(.22 .36 .05 .06)
it_may	(.41 .58 .09 .07)
meaningful_chance	(.63 .80 .05 .06)
most_likely	(.78 .92 .06 .05)
extremely_likely	(.98 .99 .05 .01)
certain	(1 1 0 0)

The membership distributions of the term set elements are illustrated in Figure 2. The values of the fuzzy interval associated with each element in the proposed term set were derived from an adaptation of the results of psychological experiments on the use of linguistic probabilities [23]. For most of the elements in the term set, the two measures of dispersions used by Beyth-Marom, e.g., the interquartile range (C_{25}-C_{75}) and the 80 per cent range (C_{10}-C_{90}), were used to define respectively the intervals $[a_i, b_i]$ and $[(a_i - \alpha_i), (b_i - \beta_i)]$ of each fuzzy number N_i.

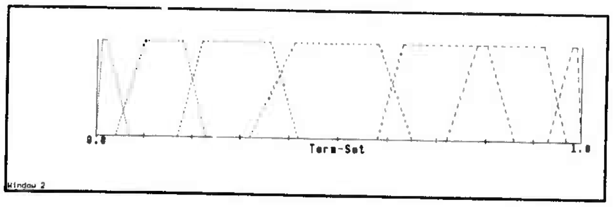


Figure 2. Membership Distributions of Elements in L_2.

DESCRIPTION OF THE EXPERIMENTS AND REQUIRED TECHNIQUES

The First Experiment

The first experiment consists in selecting nine different T-norms that, in combination with their DeMorgan dual T-conorms and a negation operator, define nine different calculi of uncertainty. Three different term sets--containing five, nine, and thirteen elements--provide three different levels of granularity for quantifying the uncertainty. For each of the three term sets, the T-norms will be evaluated on the crossproduct of the term set elements, thus generating the closure of each T-norm. Each closure will be compared with the closure of the *adjacent* T-norm and the number of differences will be computed. If there are no significant differences, the T-norms will be considered similar enough to be equivalent for any practical purpose. A threshold value will determine the maximum percentage of differences allowed among members of the same equivalence class. This concept is analogous to the hierarchical clustering technique typical of Pattern Recognition problems.

Selecting the Term Sets

The term sets used to provide the different levels of granularity in both experiments are: L_1, L_2, and L_3. L_2 contains seven elements, and was defined in Table 2. L_1 and L_3 contain five and thirteen elements, respectively. Their labels and semantics are defined in the following tables:

TABLE 3

THE FIVE ELEMENT TERM SET L_1

impossible	(0 0 0 0)
unlikely	(.01 .25 .01 .1)
maybe	(.4 .6 .1 .1)
likely	(.75 .99 .1 .01)
certain	(1 1 0 0)

Table 4

THE THIRTEEN ELEMENT TERM SET L_3

impossible	(0 0 0 0)
extremely_unlikely	(.01 .02 .01 .05)
not_likely	(.05 .15 .03 .03)
very_low_chance	(.1 .18 .06 .05)
small_chance	(.22 .36 .05 .06)
it_may	(.41 .58 .09 .07)
likely	(.53 .69 .09 .12)
meaningful_chance	(.63 .80 .05 .06)
high_chance	(.75 .87 .04 .04)
most_likely	(.78 .92 .06 .05)
very_high_chance	(.87 .96 .04 .03)
extremely_likely	([illegible] .99 .05 .01)
certain	(1 1 0 0)

Selecting the T-Norms

To select the T-norms for the experiment, we first took the three most important T-norms, i.e., T_1, [6] T_2, T_3, which provide the lower bound of the copulas,[7] an intermediate value, and the upper bound of the T-norms. We then used a parameterized family of T-norms capable of covering the entire spectrum between T_1 and T_3. Our choice fell on the family of T-norms proposed by Schweizer and Sklar, i.e., $T_{Sc}(a,b,p)$, described in Section 2.4. The selection of this particular family of T-norms was due to its full coverage of the spectrum and its numerical stability in the neighborhood of the origin. We then selected six values of the parameter p to probe the space between T_1 and T_2 ($p \in [-1,0]$), and between T_2 and T_3 ($p \in [0,\infty]$). The six T-norms instantiated from this family were: $T_{Sc}(a,b,-0.8)$, $T_{Sc}(a,b,-0.5)$, $T_{Sc}(a,b,-0.3)$, $T_{Sc}(a,b,0.5)$, $T_{Sc}(a,b,1)$, $T_{Sc}(a,b,2)$.

The selection of the parameter values was guided by the relative location of the six T-norms within the T-norm space bounded by T_1 and T_3. Figure 3 describes the space of T-norms $T_i(a,b) = K$ in the $[0,1]\times[0,1]$ universe of axb for K=0.25, 0.50, and 0.75. From this figure we can observe that, for small and medium values of K, the six T-norms instantiated from the parametric family proposed by Schweizer and Sklar, i.e., $T_{Sc}(a,b,p)$, provide a well distributed coverage[8] of the space between T_1, T_2, and T_3.

The Second Experiment

The second experiment was motivated by the behavior of the triangular conorms for high values of K, as illustrated in Figure 3. It was noted that the area of the triangular spaces corresponding to the various Ks decreases as K increases in value, i.e., Area = $(1\text{-}K)^2/2$. This can be explained by the saturation effect that most T-norms have for low values of K (and T-conorms for high values of K). However, it was also

[6] T_0, the lower bound of the T-norms, is rather uninteresting since its discontinuous and extreme behavior limits its applicability.

[7] A *copula* is a continuous 2 place function T: $[0,1]\times[0,1] \rightarrow [0,1]$ that satisfies the boundary and monotonicity conditions of the T-norms plus the following condition:

$$T(a,d) + T(c,b) \leq T(a,b) + T(c,d)$$
$$when\ a \leq c,\ b \leq d$$

Schweizer and Sklar [15] have shown that if a T-norm has an additive generator, the T-norm is a copula if and only if the additive generator is a *convex* function. With this more restrictive condition, we have that any copula $T(a,b)$ is bounded by:

$$T_1(a,b) \leq T(a,b) \leq T_3(a,b)$$

This is the more familiar set of boundaries used for the probability (and for the belief function) of the intersection of events.

[8] The nine T-norms considered in this experiment (six instances of the Schweizer and Sklar family in addition to T_1, T_2, and T_3 are maximally separated at the point $a=b$. The coordinates of the points in which the line $a=b$ intersects the six T-norms $T_{Sc}(a,b,p) = 0.25$ can be obtained from the expression:

$$a = [0.5(1 + (K)^{-p})]^{-1/p}$$

The values of the coordinate a for the intersection points of the nine T-norms ($T_1(a,b)$, $T_{Sc}(a,b,-.8)$, $T_{Sc}(a,b,-.5)$, $T_{Sc}(a,b,-.3)$, $T_2(a,b)$, $T_{Sc}(a,b,.5)$, $T_{Sc}(a,b,1)$, $T_{Sc}(a,b,2)$, $T_3(a,b)$) with the line $a=b$ are:

0.250 0.342 0.400 0.444 0.5000.573 0.562 0.600 0.625

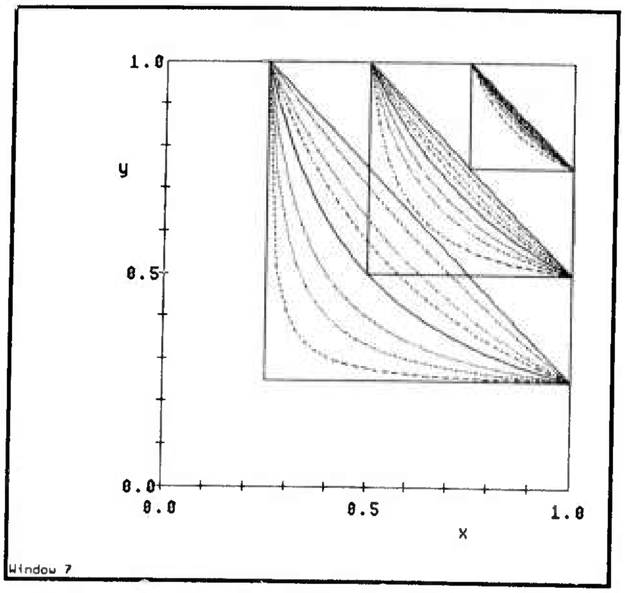


Figure 3. Space of T-norms $T_i(a,b) = K$, for K = 0.25, 0.50, and 0.75.

noted that for large values of K, most T-norms (all but T_3) seemed to converge toward T_1, therefore the space between $T_{Sc}(a,b,2)$ and T_3 was much larger than the space between any other T-norm. Figure 4 shows a plot of the nine T-norms $T_i(a,b)$, evaluated on the plane a=b. This figure illustrates both the saturation effect for small values of K and the convergency effect for high values of K.

For the sake of completeness, a second experiment was designed to provide a better sample of the space between $T_{Sc}(a,b,2)$ and T_3. Two more T-norms were instantiated from the same family of T-norms, namely $T_{Sc}(a,b,5)$ and $T_{Sc}(a,b,8)$, and added to the original nine, for a total of eleven T-norms. The same three term sets used in the first experiment were also used in this second experiment to define the input granularity. The objective of the second experiment was to verify if the first experiment had overlooked any relevant calculus requiring its own equivalence class.

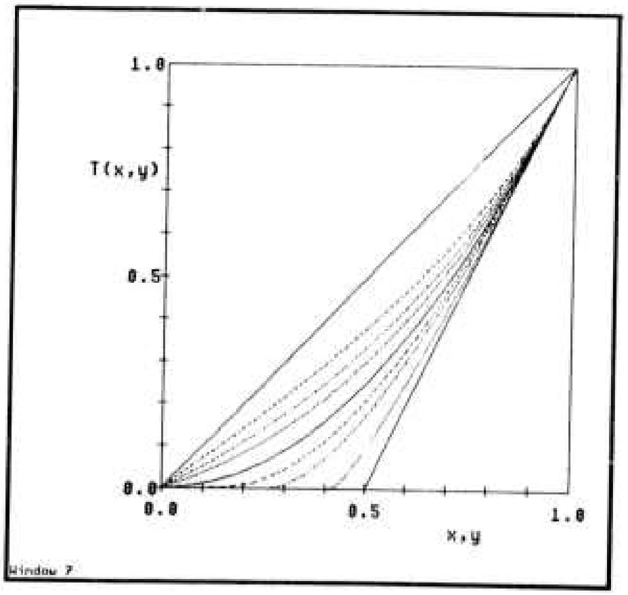


Figure 4. Space of T-norms $T_i(x,y)$ plotted for x=y.

Computational Techniques

The above experiments can be performed only if some particular computational techniques are used. It is necessary to evaluate the selected T-norms (crisply defined functions) with the elements of the term sets (linguistic variables with fuzzy-valued semantics). Furthermore, the result of this evaluation must be another element of the term set. This implies that closure must be maintained under the application of each T-norm. The following two subsections describe the techniques necessary to satisfy these requirements.

The Extension Principle

The extension principle [26] allows any non-fuzzy function to be fuzzified in the sense that if the function arguments are made fuzzy sets, then the function value is also a fuzzy set whose membership function is uniquely specified. The extension principle states that if the scalar function, f, takes n arguments $(x_1, x_2, \ldots, x_n)$, denoted by $\mathbf{X}$ and if the membership functions of these arguments are denoted by $\mu_1(x_1), \mu_2(x_2), \ldots, \mu_1(x_n)$, then

$$\mu_{f(X)}(y) = \underset{X}{SUP} \quad [\overset{n}{\underset{i=1}{INF}}\ \mu_i(x_i)]$$

$$s.t.\ f(X) = y$$

where SUP and INF denote the *Supremum* and *Infimum* operators.

The use of this formal definition entails various types of computational difficulties [26]. The solution to these difficulties is based on the parametric representation of the membership distribution of a fuzzy number,[9] i.e N_i = $(a_i, b_i, \alpha_i, \beta_i)$, described in Section 3.1. Such a representation allows one to describe uniformly a *crisp number*, e.g., $(a_i, a_i, 0, 0)$; a *crisp interval*, e.g., $(a_i, b_i, 0, 0)$; a *fuzzy number*, e.g., $(a_i, a_i, \alpha_i, \beta_i)$; and a *fuzzy interval* $(a_i, b_i, \alpha_i, \beta_i)$.

The adopted solution consists of deriving the *closed-form* parametric representation of the result. This solution is a very good approximation of the result obtained from using the extension principle to evaluate arithmetic functions with fuzzy numbers, and has a much more limited computational overhead. Table 5 shows the formulae providing the closed form solution for inverse, logarithm, addition, subtraction, multiplication, division, and power. The scope of each for-

[9] Two restrictions are imposed on the shape of the membership function of the fuzzy number represented by this parametric representation: *normality* and *convexity*. All the fuzzy numbers used to define the semantics of the proposed term sets satisfy this condition. Furthermore--except for *impossible*, the first element of each term set L_1, L_2, L_3, corresponding to a *crisp zero*--all the other elements are *positive normal convex* fuzzy numbers. They are the *only* type of fuzzy numbers that form a *commutative semi-group* [33]. They do not form a group since they lack the inverse elements for addition and multiplication. All other fuzzy numbers either do not satisfy the closure condition under some operation or do not satisfy the distributivity law.

Table 5

FORMULAE FOR ARITHMETIC OPERATIONS WITH FUZZY NUMBERS

Operation	Result	Conditions	Formula No.
$-\tilde{n}$	$(-d, -c, \delta, \gamma)$	all $\tilde{n}$	(1)
$\dfrac{1}{\tilde{n}}$	$\left[\frac{1}{d}, \frac{1}{c}, \frac{\delta}{d(d+\delta)}, \frac{\gamma}{c(c-\gamma)}\right]$	$\tilde{n} > 0, \tilde{n} < 0$	(2)
$e^{\tilde{n}}$	$(e^{c}, e^{d}, e^{c}(1-e^{-\gamma}), e^{d}(e^{\delta}-1))$	$\tilde{n} > 0$	(3)
$\log \tilde{n}$	$\left[\log c, \log d, \log \frac{c}{(c-\gamma)}, \log \frac{(d+\delta)}{d}\right]$	$\tilde{n} > 0$	(4)
$\tilde{m} + \tilde{n}$	$(a+c, b+d, \alpha+\gamma, \beta+\delta)$	all $\tilde{m}, \tilde{n}$	(5)
$\tilde{m} - \tilde{n}$	$(a-d, b-c, \alpha+\delta, \beta+\gamma)$	all $\tilde{m}, \tilde{n}$	(6)
$\tilde{m} \times \tilde{n}$	$(ac, bd, a\gamma+c\alpha-\alpha\gamma, b\delta+d\beta+\beta\delta)$	$\tilde{m} > 0, \tilde{n} > 0$	(7)
	$(ad, bc, d\alpha-a\delta+\alpha\delta, -b\gamma+c\beta-\beta\gamma)$	$\tilde{m} < 0, \tilde{n} > 0$	(8)
	$(bc, ad, b\gamma-c\beta+\beta\gamma, -d\alpha+a\delta-\alpha\delta)$	$\tilde{m} > 0, \tilde{n} < 0$	(9)
	$(bd, ac, -b\delta-d\beta-\beta\delta, -a\gamma-c\alpha+\alpha\gamma)$	$\tilde{m} < 0, \tilde{n} < 0$	(10)
$\tilde{m} \div \tilde{n}$	$\left[\frac{a}{d}, \frac{b}{c}, \frac{a\delta+d\alpha}{d(d+\delta)}, \frac{b\gamma+c\beta}{c(c-\gamma)}\right]$	$\tilde{m} > 0, \tilde{n} > 0$	(11)
	$\left[\frac{a}{c}, \frac{b}{d}, \frac{c\alpha-a\gamma}{c(c-\gamma)}, \frac{d\beta-b\delta}{d(d+\delta)}\right]$	$\tilde{m} < 0, \tilde{n} > 0$	(12)
	$\left[\frac{b}{d}, \frac{a}{c}, \frac{b\delta-d\beta}{d(d+\delta)}, \frac{a\gamma-c\alpha}{c(c-\gamma)}\right]$	$\tilde{m} > 0, \tilde{n} < 0$	(13)
	$\left[\frac{b}{c}, \frac{a}{d}, \frac{-b\gamma-c\beta}{c(c-\gamma)}, \frac{-a\delta-d\alpha}{d(d+\delta)}\right]$	$\tilde{m} < 0, \tilde{n} < 0$	(14)
$\tilde{m}^{\tilde{n}}$	$\left[a^{c}\cdot b^{d}, a^{c},-(a-\alpha)^{c-\gamma}, (b+\beta)^{d+\delta}-b^{d}\right]$	$\tilde{m}\in[1,\infty)$ $\tilde{n} > 0$	(15)
	$\left[b^{c}, a^{d}\cdot b^{c}-(b+\beta)^{c-\gamma}, (a-\alpha)^{d+\delta}-a^{d}\right]$	$\tilde{m}\in[1,\infty)$ $\tilde{n} < 0$	(16)
	$\left[a^{d}\cdot b^{c}, a^{d},-(a-\alpha)^{d+\delta}, (b+\beta)^{c-\gamma}-b^{c}\right]$	$\tilde{m}\in[0,1]$ $\tilde{n} > 0$	(17)
	$\left[b^{d}\cdot a^{c}, b^{d}-(b+\beta)^{d+\delta}, (a-\alpha)^{c-\gamma}-a^{c}\right]$	$\tilde{m}\in[0,1]$ $\tilde{n} < 0$	(18)

where $\tilde{m} \triangleq (a, b, \alpha, \beta)$ and $\tilde{n} \triangleq (c, d, \gamma, \delta)$

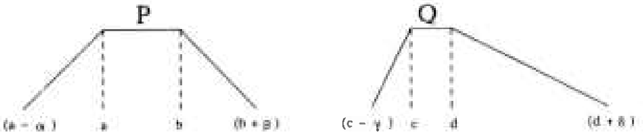
mula is defined by its attached condition[10] on the third column of Table 5. Table 6 shows the formulae for evaluating the minimum and maximum of two normal convex fuzzy numbers. All these formulae were used in the implementation of the experiments described in Sections 4.1 and 4.2.

Linguistic Approximation

The process of *linguistic approximation* consists of finding a *label* whose meaning is the same or the closest (according to some metric) to the meaning of an unlabelled membership function generated by some computational model. Bonissone [27-28] has discussed the general solution to this problem.

Table 6

FORMULAE FOR MINIMUM AND MAXIMUM OPERATORS WITH FUZZY NUMBERS



$$\mathrm{MAX}(P,Q) = (\max(a,c), \max(b,d), l, r)$$

$$\begin{array}{ll}
\text{if } (b+\beta) > (d+\delta) & r = (b+\beta) - \max(b,d) \\
\text{if } (b+\beta) < (d+\delta) & r = (d+\delta) - \max(b,d) \\
\text{if } (b+\beta) = (d+\delta) & r = \begin{cases} \beta & \text{if } b > d \\ \delta & \text{if } b < d \\ \beta=\delta & \text{if } b = d \end{cases} \\
 & \\
\text{if } (a-\alpha) > (c-\gamma) & l = (c+\alpha) - \min(a,c) \\
\text{if } (a-\alpha) < (c-\gamma) & l = (a+\gamma) - \min(a,c) \\
\text{if } (a-\alpha) = (c-\gamma) & l = \begin{cases} \alpha & \text{if } a > c \\ \gamma & \text{if } a < c \\ \alpha=\gamma & \text{if } a = c \end{cases}
\end{array}$$

$$\mathrm{MIN}(P,Q) = (\min(a,c), \min(b,d), l, r)$$

$$\begin{array}{ll}
\text{if } (b+\beta) > (d+\delta) & r = (d+\delta) - \min(b,d) \\
\text{if } (b+\beta) < (d+\delta) & r = (b+\beta) - \min(b,d) \\
\text{if } (b+\beta) = (d+\delta) & r = \begin{cases} \delta & \text{if } b > d \\ \beta & \text{if } b < d \\ \delta=\beta & \text{if } b = d \end{cases} \\
 & \\
\text{if } (a-\alpha) > (c-\gamma) & l = (a+\gamma) - \max(a,c) \\
\text{if } (a-\alpha) < (c-\gamma) & l = (c+\alpha) - \max(a,c) \\
\text{if } (a-\alpha) = (c-\gamma) & l = \begin{cases} \gamma & \text{if } a > c \\ \alpha & \text{if } a < c \\ \gamma=\alpha & \text{if } a = c \end{cases}
\end{array}$$

For our experiments, this process was simplified by the small cardinality of the term sets. Therefore, a simplified solution was adopted. From each element of the term set and from the unlabelled membership function representing the result of some arithmetic operation, two features were extracted: the first moment of the distribution and the area under the curve. A weighted Euclidean distance, where the weights reflected the relevance of the two parameters in determining semantic similarity, provided the metric required to select the element of the term set that more closely represented the result.

This process was used in the experiments described in Sections 4.1 and 4.2 to provide *closure* under the application of the various T-norms. The closure requirement is required by any calculus of uncertainty to maintain the form and meaning of the linguistic confidence measures throughout the rule chaining and aggregation process.

EXPERIMENT RESULTS AND ANALYSIS

Tabulated Results

Selected results of the experiments are shown in tabular form in Tables 7, 8, and 9. Each table illustrates the effects of applying T_1, T_2, and T_3 to the elements of a particular term set. Because of the commutativity property of the T-norms, the tables are symmetric.

Analysis of the Results of the Experiment

The three previous tables graphically illustrate the different behaviors of T_1, T_2 and T_3 when applied to a common term set. As expected, T_1 was the strictest operator and T_3 was the most liberal operator. However, the interesting aspect of the experiment was not rediscovering the behavior of the two extremes but determining how many different variations of behavior we had to consider from the operators located between T_1 and T_3.

[10] The conditions described in the third column of Table 5 refer to the sign of a fuzzy number. A fuzzy number $N_i = (a_i, b_i, \alpha_i, \beta_i)$ is positive, i.e., $N_i > 0$, iff its support is positive (i.e., $a-\alpha \geq 0$ if $\alpha \neq 0$ or $a-\alpha > 0$ if $\alpha = 0$). Analogously, $N_i < 0$ implies that its support is negative (i.e., $b+\beta \leq 0$ if $\beta \neq 0$ or $b+\beta < 0$ if $\beta = 0$).

Table 7

CLOSURE OF T_1, T_2, T_3, ON L_1

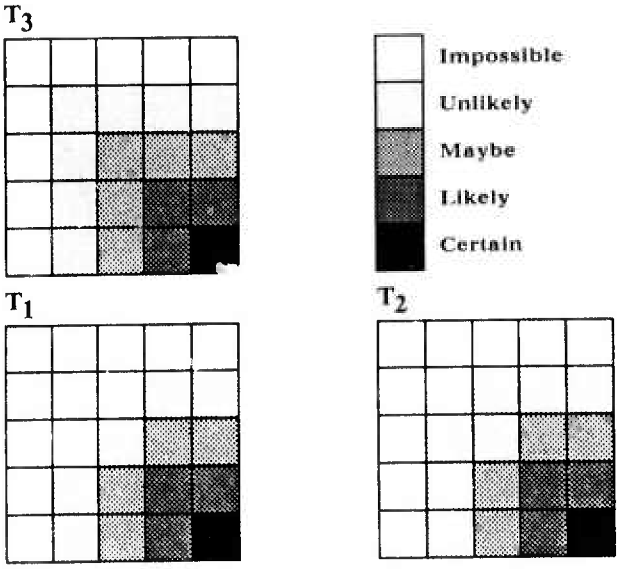


Table 8

CLOSURE OF T_1, T_2, T_3, ON L_2

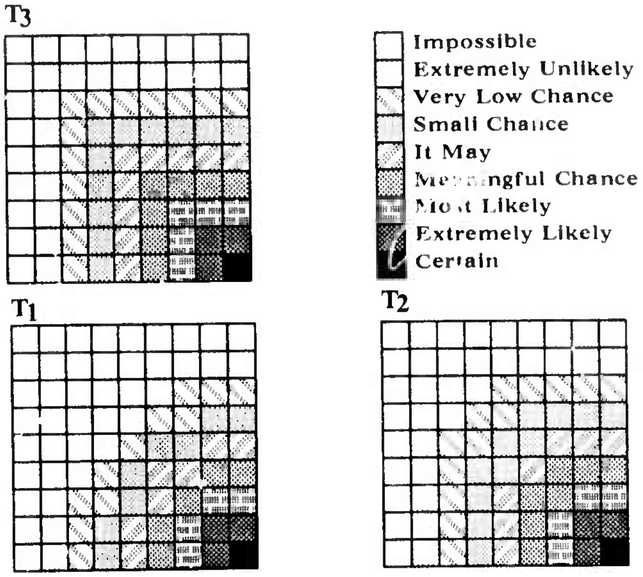


Table 9

CLOSURE OF T_1, T_2, T_3, ON L_3

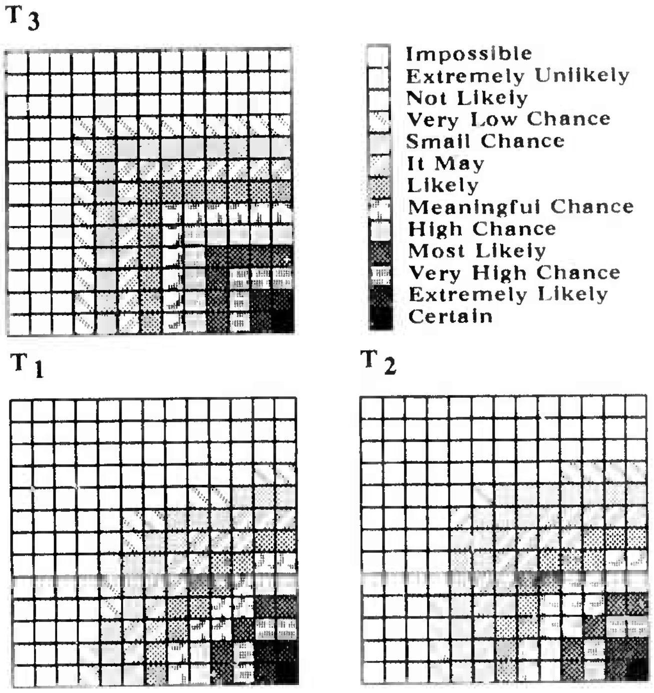
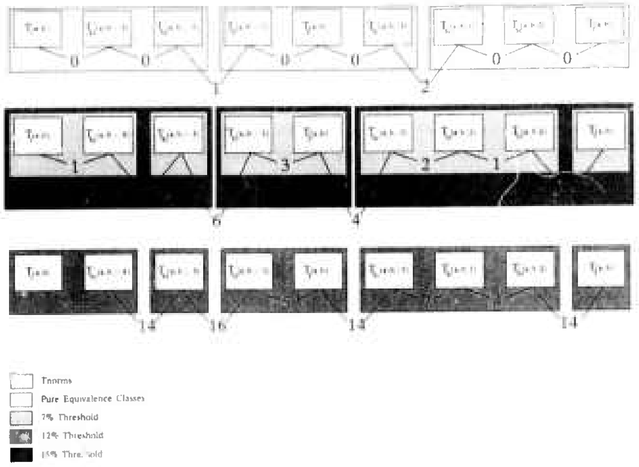


Table 10

NUMBER OF DIFFERENCES AMONG THE NINE T-NORMS APPLIED TO L_1, L_2, AND L_3.

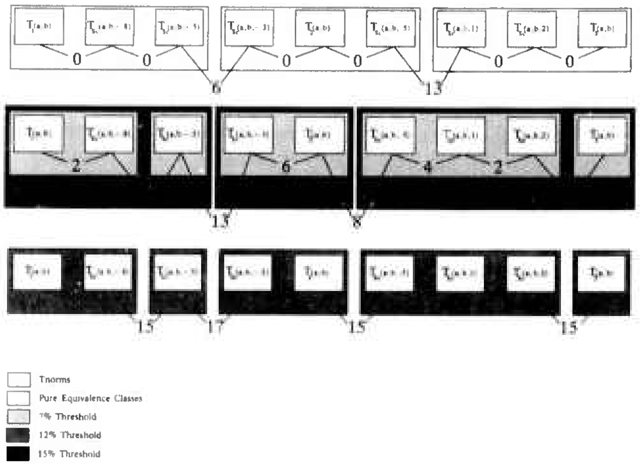


In the first experiment, the closures of seven T-norms, bounded by T_1 from below and by T_3 from above, were computed and compared with the closures of the two extremes. For each of the three term sets, each element in the closure of a given T-norm, i.e., $T_x(E_i, E_j)$, was compared with the same element in the closure of a different T-norm, i.e., $T_y(E_i, E_j)$. The number of differences found by moving from one T-norm to the next was tabulated for each term set and the results shown in Table 10. The percentages of the differences shown in Table 10 were computed as the ratio of the number of changes divided by the cardinality of the closure for each term set. Since the closures were symmetric due to the commutativity property of the T-norms, the cardinality of the closure for a term set with n elements was considered to be $n(n-1)/2$. The percentage differences are shown in Table 11.

By analyzing Table 10, it is evident that for L_1, *no differences* were found among the intermediate T-norms. There are indeed three equivalence classes of T-norms producing

Table 11

PERCENTAGE DIFFERENCES AMONG THE NINE T-NORMS APPLIED TO L_1, L_2, AND L_3



different results when applied to elements of L_1. These classes of equivalence are:

$$T_1(a,b), T_{Sc}(a,b,-0.8), T_{Sc}(a,b,-0.5)$$

$$T_{Sc}(a,b,-0.3), T_2(a,b), T_{Sc}(a,b,0.5),$$

$$T_{Sc}(a,b,1), T_{Sc}(a,b,2), T_3(a,b)$$

From the same Table 10, we can observe that *few significant differences* were found among the intermediate T-norms when applied to elements of L_2. To create equivalence classes among the T-norms, we need to establish a threshold value indicating the maximum percentage of differences that we are willing to tolerate among T-norms of the same class of equivalence. With a threshold of 7%, using Table 11 we find five classes:

$$T_1(a,b), T_{Sc}(a,b,-0.8),$$

$$T_{Sc}(a,b,-0.5),$$

$$T_{Sc}(a,b,-0.3), T_2(a,b),$$

$$T_{Sc}(a,b,0.5), T_{Sc}(a,b,1),$$

$$T_{Sc}(a,b,2), T_3(a,b)$$

With a threshold of 15% we find three classes:

$$T_1(a,b), T_{Sc}(a,b,-0.8), T_{Sc}(a,b,-0.5),$$

$$T_{Sc}(a,b,-0.3), T_2(a,b),$$

$$T_{Sc}(a,b,0.5), T_{Sc}(a,b,1),$$

$$T_{Sc}(a,b,2), T_3(a,b)$$

Finally, we can observe that for L_3 a *larger number of differences* were found among the intermediate T-norms. Using a threshold of 12% we find five classes of equivalence:

$$T_1(a,b), T_{Sc}(a,b,-0.8),$$

$$T_{Sc}(a,b,-0.5),$$

$$T_{Sc}(a,b,-0.3), T_2(a,b),$$

$$T_{Sc}(a,b,0.5), T_{Sc}(a,b,1), T_{Sc}(a,b,2),$$

$$T_3(a,b)$$

In the second experiment, the closures of nine T-norms, also bounded by T_1 from below and by T_3 from above, were computed and compared with the closures of the two extremes. For each of the same three term sets, each element in the closure of a given T-norm was compared with the same element in the closure of another, different T-norm. The number of differences found by moving from one T-norm to the next was tabulated for each term set and the results shown in Table 12. The percentages of the differences shown in Table 12 were computed as before. The percentage differences are shown in Table 13.

Table 12

NUMBER OF DIFFERENCES AMONG THE ELEVEN T-NORMS APPLIED TO L_1, L_2, AND L_3.

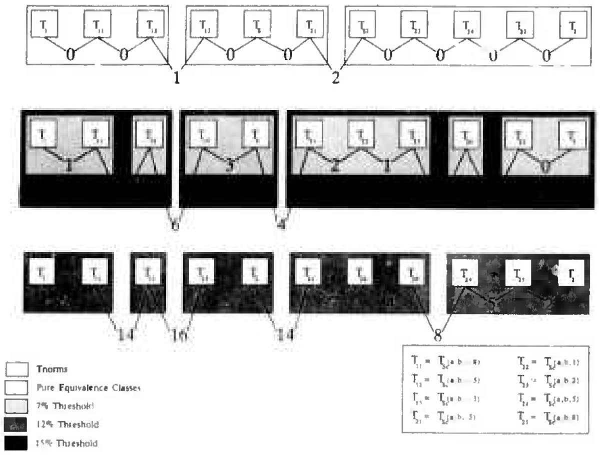
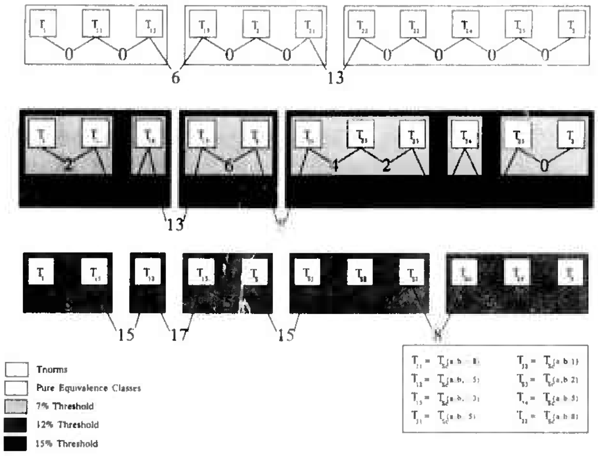


Table 13

PERCENTAGE DIFFERENCES AMONG THE ELEVEN T-NORMS APPLIED TO L_1, L_2, AND L_3



By analyzing Table 12, it is again evident that for L_1, *no differences* were found among the intermediate T-norms. The three equivalence classes of T-norms producing different results when applied to elements of L_1 are:

$$T_1(a,b), T_{Sc}(a,b,-0.8), T_{Sc}(a,b,-0.5)$$

$$T_{Sc}(a,b,-0.3), T_2(a,b), T_{Sc}(a,b,0.5),$$

$$T_{Sc}(a,b,1), T_{Sc}(a,b,2), T_{Sc}(a,b,5), T_{Sc}(a,b,8), T_3(a,b)$$

From the same Table 12, we can still observe that *few significant differences* were found among the intermediate T-norms when applied to elements of L_2. After establishing a threshold of 7% and using Table 13 we find six classes (rather than the five obtained in the first experiment):

$$T_1(a,b), T_{Sc}(a,b,-0.8),$$

$$T_{Sc}(a,b,-0.5),$$

$$T_{Sc}(a,b,-0.3), T_2(a,b),$$

$$T_{Sc}(a,b,0.5), T_{Sc}(a,b,1), T_{Sc}(a,b,2),$$

$$T_{Sc}(a,b,5),$$

$$T_{Sc}(a,b,8), T_3(a,b)$$

However, with a threshold of 8%, the last two classes of equivalence collapse into one, represented by T_3. This indicate that, for a slightly larger threshold (8% instead of 7%) the additional two T-norms added in the second experiment are not significantly different from T_3.

With a threshold of 15% we find three classes (the same as in the first experiment):

$$T_1(a,b), T_{Sc}(a,b,-0.8), T_{Sc}(a,b,-0.5),$$

$$T_{Sc}(a,b,-0.3), T_2(a,b),$$

$$T_{Sc}(a,b,0.5), T_{Sc}(a,b,1), T_{Sc}(a,b,2), T_{Sc}(a,b,5), T_{Sc}(a,b,8), T_3(a,b)$$

Finally, we can observe that for L_3 a *larger number of differences* were still found among the intermediate T-norms. Using a threshold of 12% we again find five classes of equivalence:

$$T_1(a,b), T_{Sc}(a,b,-0.8),$$

$$T_{Sc}(a,b,-0.5),$$

$$T_{Sc}(a,b,-0.3), T_2(a,b),$$

$$T_{Sc}(a,b,0.5), T_{Sc}(a,b,1), T_{Sc}(a,b,2),$$

$$T_{Sc}(a,b,5), T_{Sc}(a,b,8), T_3(a,b)$$

In summary, we can see that three T-norms are sufficient to define the relevant calculi using the five element term set L_1; five T-norms are required to represent (88% of the time) the variations in relevant calculi for the thirteen element term set L_3. For the case of L_2, the same three T-norms used for L_1 will suffice if we are willing to accept results that might be slightly[11] different 15% of the time. Otherwise, we will have to use five T-norms, as for L_3, to reduce the number of slight differences to 8%. These results hold for both experiments.

For any practical purpose, the three classes of equivalence represented by T_1, T_2, and T_3 more than adequately represent the variations of calculi that can produce different results when applied to elements of term sets with at most nine elements.

The results of both experiments hold for the T-conorms as well. The elements of each term set are almost symmetric with respect to the middle point of the scale, 0.5. Therefore, by using the Linguistic Approximation, the closure of the negation operator can be simply computed by *reversing* the order of the elements in each term set. The closures for the T-conorms can then be computed from the closures of the T-norms and the closure of the negation operator, using DeMorgan's identity. The classes of equivalence obtained for the T-norms are the same as those obtained for their dual T-conorms.

The appropriate selection of uncertainty granularity (i.e., the term set cardinality) is still a matter of subjective judgement. However, if we use the very well-known results on the *span of absolute judgement* [29], it seems unlikely that any expert or user could consistently quantify uncertainty using more than nine different values.

Meaning of T_1, T_2, T_3

T_1, T_2, and T_3 were the three operators that produced notably different results for L_1 and L_2. A challenging task is to establish the meaning of each T-norm, i.e., the rationale for selecting one T-norm over the other two.

A first interpretation indicates that T_1 seems appropriate to perform the intersection of lower probability bounds [30]. Similarly, T_3 is appropriate to represent the intersection of upper probability bounds. T_2 is the classical probabilistic operator that assumes *independence* of the arguments; its dual T-conorm, S_2, is the usual *additive* measure for the union.

To provide a better understanding of these T-norms, we will paraphrase an example introduced by Zadeh [31]:

If 30% of the students in a college are engineers, and 80% of the students are male, how many students are both male and engineers?

Although we started with numerical quantifiers, the answer is no longer a number, but is given by the interval [10%, 30%]

The lower bound of the answer is provided by $T_1]$(0.3, 0.8); T_3(0.3, 0.8) generates its upper bound. T_2(0.3, 0.8) gives a somewhat arbitrary estimate of the answer, based on the independence of the two pieces of evidence.

In Figure 5, we try to describe geometrically the meaning of the three T-norms. The figure illustrates the result of T_1(0.3, 0.8), T_2(0.3, 0.8), and T_3(0.3, 0.8). T_1 captures the notion of *worst case*, where the two arguments are considered as *mutually exclusive* as possible (the dimensions on which they are measured are 180° apart). T_2 captures the notion of *independence* of the arguments (their dimensions are 90° apart). T_3 captures the notion of *best case*, where one of the arguments attempts to *subsume* the other one (their dimensions are collinear, i.e., 0° apart).

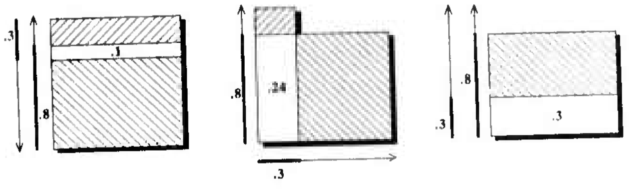


Figure 5. Geometrical Interpretation of T_1(0.3, 0.8), T_2(0.3, 0.8), and T_3(0.3, 0.8).

[11] The *slight* difference in the result implies that sometimes the result will be an element of the term set that is *adjacent* to the correct one.

Meaning of $T_{Sc}(a,b,-0.5)$ and $T_{Sc}(a,b,1)$

There are two cases in which we will need to deal with five calculi instead of three. In the first case, we want to decrease the input granularity by using a term set with a finer resolution than L_2 (e.g., L_3). In the second case, within the granularity provided by L_2, we want to decrease the percentage of differences within an equivalence class by lowering the tolerance threshold from 15% to 8%. In either case, we must provide an interpretation for the meaning of the two additional T-norms, i.e., $T_{Sc}(a,b,-0.5)$ and $T_{Sc}(a,b,1)$.

A rather straightforward interpretation of $T_{Sc}(a,b,-0.5)$ and $T_{Sc}(a,b,1)$ is to consider them the intersection operators for pieces of evidence that exhibit *mild* negative or positive correlation, respectively. This is in contrast with T_1 and T_3 that represent the *extreme* cases of negative and positive correlation, respectively.

CONCLUSIONS

Summary of the Results

In this paper we have presented a formalism to represent any truth functional calculus of uncertainty in terms of a selection of a negation operator and two elements from families of T-norms and T-conorms. Because of our skepticism regarding the realism of the *fake precision* assumption required by most existing numerical approaches, we proposed the use of a term set that determines the finest level of specificity, i.e., the *granularity*, of the measure of certainty that the user/expert can *consistently* provide. The suggested semantics for the elements of the term set are given by fuzzy numbers on the [0,1] interval. The values of the fuzzy numbers were determined on the basis of the results of a psychological experiment aimed at the consistent use of linguistic probabilities.

We then proceeded to perform two experiments to test the required level of discrimination among the various calculi, given a fixed uncertainty granularity. We reviewed the techniques required to implement the experiments, such as the extension principle (that permits the evaluation of crisply defined function with fuzzy arguments), a parametric representation of fuzzy numbers (that allows closed form solutions for arithmetical operations), and the process of *linguistic approximation* of a fuzzy number (that guarantees *closure* of the term set under the various calculi of uncertainty).

We computed the closure of nine and eleven T-norm operators applied to three different term sets. We analyzed the sensitivity of each operator with respect to the granularity of the elements in the term set; and we finally determined that only three T-norms — T_1, T_2, and T_3 — generated sufficiently distinct results for those term sets that contain no more than nine elements.

Impact of the Results to Expert System Technology

In our final conclusions, we would like to establish an explicit link between the results of this paper and the problem of reasoning with uncertainty in expert systems. In building expert systems architectures three distinct layers must be defined: *representation*, *inference*, and *control* layers. The treatment of uncertainty in expert systems must address each of these layers. The characterization of uncertainty measures as linguistic variables with fuzzy-valued semantics and the use of a given uncertainty calculus address the representation and inference layers, respectively. The selection of the most appropriate calculus to be used must be addressed by the control layer.

However, in most expert systems, the control layer has been procedurally embedded in the inference engine, thus preventing any opportunistic and dynamic change in ordering inferences and in aggregating uncertainty. Usually, the same type of aggregation operators, i.e., the same uncertainty calculus, is selected a priori and is used uniformly for any inference made by the expert system. The most recent trend in building expert systems is moving toward having a declarative representation for the control layer.

As an integral part of this layer, we suggest to define a set of context dependent rules that will select the most appropriate calculus for any given situation. Such a rule set will be relatively small since it must describe only the selection policies for a small number of calculi. The reduced number of calculi is the result of the analyzed trade-off between complexity and precision. These rules will rely on contextual information -- such as the nature, reliability, and characteristics of the evidence sources -- as well as on the meanings of the three or five analyzed calculi that will be used in the inference layer.

REFERENCES

[1] Bonissone, P.P. & Tong, R.M., (1985). Editorial: Reasoning with Uncertainty in Expert Systems, *International Journal of Man-Machine Studies*, Vol. 22, No. 3, March 1985.

[2] Bonissone, P.P. (1985). Reasoning with Uncertainty in Expert Systems: Past, Present, and Future, KBS Working Paper, General Electric Corporate Research and Development Center, Schenectady, New York, presented at the International Fuzzy Systems Association (IFSA) 1985, Mallorca, Spain July 1-5, 1985.

[3] Bonissone, P.P. & Brown, A.L. (1985). Expanding the Horizons of Expert Systems To appear in the *Proceedings of the Second International Conference on Artificial Intelligence Technologies, Expert Systems and Knowledge Engineering*, Ruschlikon, Switzerland, 25-26 April 1985.

[4] Dubois, D. & Prade, H. (1984). Criteria Aggregation and Ranking of Alternatives in the Framework of Fuzzy Set Theory, *TIMS/Studies in the Management Science*, H.J. Zimmerman, L.A. Zadeh, B.R. Gaines (eds.), Vol. 20, pp. 209-240, Elsevier Science Publishers.

[5] Dubois, D. & Prade, H., (1982). A Class of Fuzzy Measures Based on Triangular Norms, *International Journal of General Systems*, Vol. 8, 1.

[6] Bellman, R. & Giertz, M., (1973). On the analytic formalism of the theory of fuzzy sets, *Information Sciences*, Vol. 5, pp. 149-156.

[7] Lowen, R., (1978). On Fuzzy Complements, *Information Science*, Vol. 14, pp. 107-113.

[8] Trillas, E., (1979). Sobre funciones de negacion en la teoria de conjuntos difusos, *Stochastica*, (Polytechnic University of Barcelona, Spain), Vol. III, No.1, pp. 47-60

[9] Klement, E.P., (1981). Operations on Fuzzy Sets and Fuzzy Numbers Related to Triangular Norms, *Proceedings of the 11th International Symposium on Multiple-Valued Logic*, IEEE Computer Society Press, pp. 218-225, May 27-29, 1981, Oklahoma City, Oklahoma.

[10] Ling, C-H, (1965). Representation of Associative Functions, *Publicationes Mathematicae Debrecen*, Vol. 12, pp.189-212.

[11] Fung, L.W. & Fu, K.S., (1975). An axiomatic approach to rational decision-making in a fuzzy environment, in *Fuzzy sets and their applications to cognitive and decision processes*, Zadeh, L.A., Fu, K.S., Tanaka, K. and Shimura, M. (eds.), pp. 227-256, Academic Press, New York.

[12] Zadeh, L.A. (1965). Fuzzy Sets, *Information and Control*, Vol. 8, pp. 338-353.

[13] Zadeh, L.A., (1975) Fuzzy logic and approximate reasoning (in memory of Grigor Moisil),*Synthese*, Vol. 30, pp. 407-428.

[14] Dubois, D. & Prade, H., (1980). New Results about Properties and semantics of fuzzy set-theoretic operators, in *Fuzzy Sets: Theory and Applications to Policy Analysis and Information Systems*, P.P. Wang & S.K. Chang (Eds.), pp. 59-75, Plenum Press, New York.

[15] Schweizer, B., Sklar, A. (1963). Associative Functions and Abstract Semi-Groups, *Publicationes Mathematicae Debrecen*, Vol. 10, pp. 69-81.

[16] Yager, R., (1980). On a General Class of Fuzzy Connectives, *Fuzzy Sets and Systems*, Vol. 4, pp. 235-242.

[17] Hamacher, H, (1975). Uber logische Verknupfungen unscharfer Aussagen und deren zugehorige Bewertungs-funktionen, in *Progress in Cybernetics and Systems Research, Vol. II*, R. Trappl & F. de P. Hanica (Eds.), pp. 276-287, Hemisphere Pub. Corp., New York.

[18] Frank, M.J., (1979). On the simultaneous associativity of F(x,y) and x+y-F(x,y), *Aequationes Mathematicae*, Vol. 19, pp. 194-226.

[19] Sugeno, M., (1974). Theory of Fuzzy Integrals and its Applications, Ph. D. dissertation, Tokyo Institute of Technology.

[20] Sugeno, M., (1977). Fuzzy Measures and Fuzzy Integrals: a Survey, in *Fuzzy Automata and Decision Processes*, M.M. Gupta, G.N. Saridis & B.R. Gaines (Eds.), pp. 89-102, North Holland, New York.

[21] Szolovits, P. & Pauker, S.G., (1978). Categorical and probabilistic reasoning in medical diagnosis, *Artificial Intelligence Journal*, Vol. 11, pp. 115-144

[22] Beyth-Marom, R., (1982). How Probable is Probable? A Numerical Taxonomy Translation of Verbal Probability Expressions, *Journal of Forecasting*, Vol. 1, pp. 257-269.

[23] Phillips, L. & Edwards, W. (1966). Conservatism in a simple probability inference task, *Journal of Experimental Psychology*, Vol. 72, pp. 346-354.

[24] Zimmer, A.C. (1985). The Estimation of Subjective Probabilities vai Categorical Judgments of Uncertainty, *Proceedings of the Workshop on Uncertainty and Probability in Artificial Intelligence*, pp. 217-224, UCLA, Los Angeles, California, August 14-16, 1985.

[25] Zadeh, L.A., (1975) The concept of a linguistic variable and its application to approximate reasoning, Part I, *Information Sciences*, Vol. 8, pp. 199-249; Part II, *Information Sciences*, Vol. 8, pp. 301-357; Part III, *Information Sciences*, Vol. 9, pp. 43-80.

[26] Bonissone, P.P., (1980). A Fuzzy Sets Based Linguistic Approach: Theory and Applications, *Proceedings of the 1980 Winter Simulation Conference*, edited by T.I. Oren, C.M. Shub, P.F. Roth, pp. 99-111, Orlando, December 1980. Also in *Approximate Reasoning in Decision Analysis*, edited by M.M. Gupta, E. Sanchez, pp. 329-339, North Holland Publishing Co., New York, 1982.

[27] Bonissone, P.P., (1979). The problem of Linguistic Approximation in System Analysis, Ph. D. dissertation, Dept. EECS, University of California, Berkeley, 1979. Also in University Microfilms International Publications #80-14,618, Ann Arbor, Michigan.

[28] Bonissone, P.P., (1979). A Pattern Recognition Approach to the Problem of Linguistic Approximation in System Analysis, *Proceedings of the IEEE International Conference on Cybernetics and Society*, pp. 793-798, Denver, October 1979.

[29] Miller, G. A. (1967). The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information, in *The Psychology of Communication*, Penguin Books, Inc.

[30] Dempster, A.P. (1967). Upper and Lower Probabilities Induced by a Multivalued Mapping, *The Annals of Mathematical Statistics*, 38-2, pp. 325-339.

[31] Zadeh, L.A. (1983). A Computational Approach to Fuzzy Quantifiers in Natural Languages, *Computer & Mathematics with Applications*, Vol. 9, No. 1, pp. 149-184.

[32] Quinlan, J.R. (1983). INFERNO: A Cautious Approach to Uncertain Inference, *Computer Journal*, Vol. 26.

[33] Mizumoto, M. & Tanaka, K. (1979). Some Properties of Fuzzy Numbers, in *Advances in Fuzzy Set Theory and Applications*, M.M. Gupta, R.K. Ragade, R.R. Yager (Eds.), pp. 153-164, North-Holland Publishing Co.

APPENDIX: PROPERTIES OF T-NORM OPERATORS

The subset of properties satisfied by a given T-norm operator succinctly defines its behavior. The properties that capture the most salient features of such an operator are:

Continuous: an infinitesimal change in one of the arguments cannot cause a noticeable change in the result

Archimedean: continuous and satisfying the following conditions:

$T(x,x) < x \quad S(x,x) > x \quad \text{for all } x \in (0,1)$

Idempotent: $T(x,x) = x \quad S(x,x) = x \quad \text{for all } x \in [0,1]$

Strict: continuous and strictly increasing in both places, i.e, satisfying the the following conditions:

$T(x,y) < T(x,y')$ and

$T(y,x) < T(y',x)$

for $x>0$, $y<y'$, and

$T(a,b)=\lim_{c \to a} T(c,b)=\lim_{d \to b} T(a,d)$

Nilpotent: Given a sequence $\{x_1,\ldots,x_n\}$ of numbers in (0,1), there is a finite number n for which:

$T(x_1,\ldots,x_n) = 0$ and $\sum_{i=1}^{n} f(x_i) > f(0)$, where $f(x)$

is the additive generator of the T-norm [10,15].

The T-norm operators used in the last column of Table 1 satisfy the following properties:

	T_0	T_1	$T_{1.5}$	T_2	$T_{2.5}$	T_3
Continuous	NO	YES	YES	YES	YES	YES
Archimedean	NO	YES	YES	YES	YES	NO
Idempotent	NO	NO	NO	NO	NO	YES
Strict	NO	NO	YES	YES	YES	NO
Nilpotent	—[12]	YES	NO	NO	NO	—[12]

Any continuous Archimedean T-norms is either *strict* or *nilpotent*. Its classification can be obtained by analyzing the T-norm's additive generator:

Continuous Archimedean *Strict* T-norms have an additive generator $f(x)$ such that:

$$f(0) = \infty \text{ and } f(1) = 0$$

Continuous Archimedean *Nilpotent* T-norms have an additive generator $f(x)$ such that:

$$f(0) < \infty \text{ and } f(1) = 0$$

It is worth noting that the three T-norms analyzed in the conclusions, i.e., T_1, T_2, and T_3, are nilpotent, strict, and idempotent, respectively.

[12] The nilpotent property is defined in terms of the T-norm's additive generator Both T_0 and T_3 do not have any additive generator (T_0 is not continuous, T_3 is nor Archimedean).

SUMMARIZING AND PROPAGATING UNCERTAIN INFORMATION WITH TRIANGULAR NORMS

Piero P. Bonissone

ABSTRACT

A large variety of numerical or symbolic approaches to reasoning with uncertainty have been proposed in the AI literature. In this paper we postulate a desiderata that any such formalism should attempt to satisfy. We then propose a new formalism for reasoning with uncertainty, which is organized in three layers: the *representation, inference*, and *control* layer. In the representation layer we describe the structure required to capture information used in the inference layer and meta-information used in the control layer. In this structure, numerical slots take values on linguistic term sets with fuzzy-valued semantics. These term sets capture the *input granularity* usually provided by human experts or users. In the inference layer we describe a large number of uncertainty calculi based on Triangular norms (T-norms), intersection operators whose *truth functionality* entails low computational complexity. We show that, for a common negation operator, the selection of a T-norm uniquely and completely describes an uncertainty calculus. From previous experiments we have determined the existence of a small number of equivalence classes among the uncertainty calculi (as a function of the input granularity). This property drastically reduces the number of *different* combining rules to be considered. In the control layer we specify the policy selection for the different calculi used in the inference layer, based on their meanings, properties, and contextual information. Conflicts and ignorance measurements are also proposed.

INTRODUCTION TO REASONING WITH UNCERTAINTY

In most realistic situations, the information available to the decision maker is incomplete and uncertain. In automated reasoning systems, these two facets of the information have usually been treated independently. Theories and techniques for dealing with incomplete (but precise) information have evolved into the development of non-monotonic logics [17-18], Truth Maintenance Systems (TMS) [16], and Reason Maintenance Systems (RMS) [4,7]. Theories and techniques for dealing with uncertain (but complete) information have been either adapted from other fields, such as probability theory, by accepting unrealistic *global* assumptions, or proposed as an ad hoc solution without formal justifications [6].

In this paper we want to analyze the problem of reasoning with uncertainty *within the context* of automated reasoning. This implies that the formalism for reasoning with uncertainty must exhibit the same structural (layered) decomposition typical of other automated reasoning methodologies. The formalism must be based on sound theoretical foundations to guarantee its general applicability to a variety of reasoning tasks. The proposed layered approach will be suitable to integration with Reason Maintenance Systems that provide a distinction between the object logic theory (inference layer) and the meta logic theory (control layer).

Three Layers Organization

In building expert systems architectures three distinct layers must be defined: *representation, inference*, and *control* layers. It is our claim that the treatment of uncertainty in expert systems must address each of these layers.

The majority of the approaches to reasoning with uncertainty do not properly cover these issues. Some approaches lack expressiveness in their representation paradigm. Other approaches require unrealistic assumptions to provide uniform combining rules defining the plausible inferences.

Specifically, the non-numerical approaches [8-10], are inadequate to represent and summarize measures of uncertainty. The numerical approaches generally tend to impose some restrictions upon the type and structure of the information (e.g., mutual exclusiveness of hypotheses, conditional independence of evidence). Most numerical approaches represent uncertainty as a precise quantity (scalar or interval) on a given scale. They require the user or expert to provide a *precise* yet *consistent* numerical assessment of the uncertainty of the atomic data and of their relations. The output produced by these systems is the result of laborious computations, guided by well-defined calculi, and *appears* to be equally precise. However, given the difficulty in consistently eliciting such numerical values from the user, it is clear that these models of uncertainty require an unrealistic level of precision that does not actually represent a real assessment of the uncertainty.

With few exceptions, such as MRS [14], the control of the inference process in most expert systems has been *procedurally* embedded in the inference engine, thus preventing any opportunistic and dynamic change in ordering inferences and in aggregating uncertainty. Usually, the same type of aggregation operators (i.e., the same uncertainty calculus) is selected a priori and is used uniformly for any inference made by the expert system. In the few numerical approaches where conflictive information is detected [22] its handling is done in the inference layer, where the conflict resolution procedure is embedded in the same combining rules. This procedure consists of removing the conflictive part of the information. The non-conflictive portion is then normalized and propagated as if the conflict never existed.

In this paper we describe an alternative paradigm, where some of the above shortcomings will be avoided. In Section 1, we postulate a desiderata that specifies the most important requirements for each of the three layers of representation, inference, and control. We then propose an approach to reasoning, with uncertainty, organizing its description around the three layers structure. In Section 2, we discuss the *representation* layer that determines issues such as the appropriate data structure for the uncertainty information (used in the inference layer) and meta-information (used by the control layer), the input granularity selection, and the term set calibration. In Section 3, we illustrate the *inference* layer that determines the uncertainty calculi to perform the intersection, detachment, union, and pooling of the information. In Section 4, we analyze the *control* layer that determines the calculi selection, the conflict measurement and resolution, the ignorance measurement, and the resource allocation.

Desiderata for Reasoning with Uncertainty

The following *desiderata* represents a list of requirements to be satisfied by the ideal formalism for representing uncertainty and making inference with uncertainty. A comparative evaluation of existing approaches to reasoning with uncertainty against a subset of this requirements list can be found in [6]. To be consistent with the organizing principle described in the Section 2, the desiderata is subdivided into the same three layers of Representation, Inference, and Control.

Representation Layer

1. There should be an explicit representation of the *amount* of evidence for *supporting* and for *refuting* any given hypothesis.
2. There should be an explicit representation of the *reasons* for *supporting* and for *refuting* any given hypothesis, to be used for conflict resolution by the control layer.
3. The representation should allow the user to describe the uncertainty of information at the available level of detail (i.e., allowing *heterogeneous information granularity*).
4. There should be an explicit representation of *consistency*. Some measure of consistency or compatibility should be available to detect trends of potential conflicts and to identify essential contributing factors in the conflict.
5. There should be an explicit representation of *ignorance* to allow the user to make *non-committing* statements, i.e., to express the user's lack of conviction about the certainty of *any* of the available choices or events. Some measure of ignorance, similar to the concept of entropy, should be available to guide the gathering of discriminant information.
6. The representation must be, or at least must appear to be natural to the user to enable him/her to *describe uncertain input* and to *interpret uncertain output*. The representation must also be natural to the expert to enable him/her to elicit *consistent* weights representing the strength of the implication of each rule.

Inference Layer

7. The combining rules should not be based on global assumptions of *evidence independence*.
8. The combining rules should not be based on global assumptions of *hypotheses exhaustiveness* and *exclusiveness*.
9. The combining rules should maintain the *closure* of the syntax and semantics of the representation of uncertainty.
10. Any function used to propagate and summarize uncertainty should have clear semantics. This is needed both to maintain the semantic closure of the representation and to allow the control layer to *select* the most appropriate combining rules.

Control Layer

11. There should be a clear distinction between a *conflict* in the information (i.e., violation of consistency), and *ignorance* about the information.
12. The *traceability* of the aggregation and propagation of uncertainty through the reasoning process must be available to resolve conflicts or contradictions, to explain the support of conclusions, and to perform meta-reasoning for control.
13. It should be possible to make pairwise comparisons of uncertainty since the induced *ordinal or cardinal ranking* is needed for performing any kind of decision-making activities.
14. There should be a second order measure of uncertainty. It is important to measure the uncertainty of the information as well as the *uncertainty of the measure* itself.
15. It should be possible to *select* the most appropriate combination rule by using a declarative form of control (i.e., by using a set of context dependent rules that specify the selection policies).

REPRESENTATION LAYER

Representing Uncertainty Information and Meta-Information

In a previous paper [3], we noticed that "...the uncertainty of some type of evidence or facts is a complex object, and it is unlikely that a single, uniform representation will ever be sufficient to model it. An intriguing approach is that of attempting to combine, whenever possible, the symbolic information provided by a complex data structure (frame-like), as in the theory of endorsements, with some of the quantitative representations previously described, such as the theory of necessity and possibility."

This suggestion has evolved into the development of a representation that captures uncertainty information, used in the inference layer, and meta-information, used in the control layer. This representation is a certainty-frame (or unit) with a set of associated slots. Some of these slots contain numerical values, such as the amount of *confirmation* and the amount of *refutation* of evidence A, denoted by $N(A)$ and $N(\neg A)$, respectively, that will be used and combined by the uncertainty calculi. $N(A)$ represents the lower bound of the degree of confirmation of evidence A. As in the case of Dempster's (or Shafer's) lower and upper probability bounds, the following identity holds: $N(\neg A) = 1\text{-}Pl(A)$, where $Pl(A)$ denotes the upper bound of the certainty in A, and is interpreted as the amount of failure to refute A.

Other numerical slots contain the evaluation of the measure's uncertainty (a second order measure analogous to the concept of variance), the evaluation of an entropy function defining the quality of the given information, and a measure of the (potential) conflict. These slots will quickly provide the control layer with a numerical summary to assess the presence and amount of ignorance and conflict. A description of these slots is given in Section 4.

The non-numerical slots provide further information to the control layer allowing it to reason *about* the evidence's uncertainty, rather than *with* the evidence's uncertainty.

The selection of the appropriate uncertainty calculus must be determined in the control layer on the basis of the calculi's characteristics and the contextual information captured by these slots. Such contextual information is described by slots such as the evidence's source, the source's prior credibility in providing that type of evidence, the (environmental or operational) conditions under which the source obtained such information.

Defining Input Granularity for Numerically Valued Slots

Szolovits and Pauker [24] noted that "...while people seem quite prepared to give qualitative estimates of likelihood, they are often notoriously unwilling to give precise numerical estimates to outcomes." This seems to indicate that any scheme that relies on the user providing *consistent* and *precise numerical* quantifications of the confidence level of his/her conditional or unconditional statements is bound to fail.

It is instead reasonable to expect the user to provide *linguistic* estimates of the likelihood of given statements. The experts and users would be presented with a verbal scale of certainty expressions that they could then use to describe their degree of certainty in a given rule or piece of evidence. Recent psychological studies have shown the feasibility of such an approach: "...A verbal scale of probability expressions is a compromise between people's resistance to the use of numbers and the necessity to have a common numerical scale" [1].

Linguistic probabilities offer another advantage. When dealing with subjective assessment of probability, it has been observed [19] that conservatism is consistently present among the suppliers of such assessments. The subjects of various experiments seem to stick to the original (a priori) assessments regardless of new amount of evidence that should cause a revision of their belief. In a recent experiment [27], linguistic probabilities have been compared with numerical probabilities to determine if the observed conservatism in the belief revision was a phenomenon intrinsic in the perception of the events or due to the type of representation (i.e., numerical rather than verbal expressions). The results indicate that people are much closer to the optimal Bayesian revision when they are allowed to use linguistic probabilities.

The use of three different term sets, with five, nine and thirteen elements, respectively, has been proposed in a previous paper [5]. Each term set defines a different verbal scale of certainty, by providing a different set of *linguistic* estimates of the likelihood of any given statement. Thus, the selection of a term set determines the uncertainty granularity (i.e., the finest level of distinction among different quantifications of uncertainty). The semantics for the elements of each term set are given by fuzzy numbers on the [0,1] interval. A fuzzy number is a fuzzy set defined on the real line. In this case, the membership function of a fuzzy set defined on a truth space, i.e. the interval [0,1], could be interpreted as the *meaning* of a label describing the degree of certainty in a linguistic manner [2,26]. The values of the fuzzy numbers have been determined from the results of a psychological experiment aimed at the consistent use of linguistic probabilities [1].

The triangular norms, which form the basis for the various uncertainty calculi discussed in Section 3, take as arguments *real number* values on the [0,1] interval, which must be initially provided by the user or the expert. Their applicability is extended to fuzzy numbers by using a parametric representation for fuzzy numbers that allows closed form solutions for arithmetical operation.

INFERENCE LAYER

This section summarizes the functionalities and axiomatic definitions of the operators that form an uncertainty calculus. A detailed discussions of these operators can be found in a previous paper [5], except for the detachment operators. These operators, not discussed in reference 5, are examined in Section 3.1.4.

Defining the Uncertainty Calculi

The generalizations of conjunctions and disjunctions play a vital role in the management of uncertainty in expert systems: they are used in evaluating the satisfaction of premises, in propagating uncertainty through rule chaining, and in consolidating the same conclusion derived from different rules. More specifically, they provide the answers to the following questions:

- When the premise is composed of multiple clauses, how can we aggregate the degree of certainty x_i of the facts matching the clauses of the premise? (i.e., what is the function $T(x_1, \ldots, x_n)$ that determines x_p, the degree of certainty of the premise?).
- When a rule does not represent a logical implication, but rather an empirical association between premise and conclusion, how can we aggregate the degree of satisfaction of the premise x_p with the strength of the association s_r? (i.e., what is the function $G(x_p, s_r)$ that propagates the uncertainty through the rule?).
- When the same conclusion is established by multiple rules with various degrees of certainty $y_1, \ldots, y_m$, how can we aggregate these contributions into a final degree of certainty? (i.e., what is the function $S(y_1, \ldots, y_m)$ that consolidates the certainty of that conclusion?).

Triangular norms (T-norms) and Triangular conorms (T-conorms) are the most general families of binary functions that satisfy the requirements of the conjunction and disjunction operators, respectively. T-norms and T-conorms are two-place functions from [0,1]x[0,1] to [0,1] that are monotonic, commutative and associative. Their corresponding boundary conditions satisfy the truth tables of the logical AND and OR operators.

Conjunction Operators

The function *T(a,b)* aggregates the degree of certainty of two clauses in the same premise. This function is a *conjunction* operator and satisfies the conditions of a Triangular norm (T-norm):

$T(0,0) = 0$	[boundary]
$T(a,1) = T(1,a) = a$	[boundary]
$T(a,b) \leq T(c,d)$ if $a \leq c$ and $b \leq d$	[monotonicity]
$T(a,b) = T(b,a)$	[commutativity]
$T(a, T(b,c)) = T(T(a,b),c)$	[associativity]

Although defined as a two-place function, a T-norm can be used to represent the intersection of a larger number of clauses in a premise. Because of the associativity of the T-norms, it is possible to define recursively $T(x_1, \ldots, x_n, x_{n+1})$, for $x_1, \ldots, x_{n+1} \in [0,1]$, as:

$$T(x_1, \ldots, x_n, x_{n+1}) = T(T(x_1, \ldots, x_n), x_{n+1})$$

Disjunction Operators

The function $S(a,b)$ aggregates the degree of certainty of the (same) conclusions derived from two rules. This function is a *disjunction* operator and satisfies the conditions of a Triangular conorm (T-conorm):

$S(1,1) = 1$	[boundary]
$S(0,a) = S(a,0) = a$	[boundary]
$S(a,b) \leq S(c,d)$ if $a \leq c$ and $b \leq d$	[monotonicity]
$S(a,b) = S(b,a)$	[commutativity]
$S(a,S(b,c)) = S(S(a,b),c)$	[associativity]

A T-conorm can be extended to operate on more than two arguments in a manner similar to the extension for the T-norms. By using a recursive definition, based on the associativity of the T-conorms, it is possible to define:

$$S(y_1, \ldots, y_m, y_{m+1}) = S(S(y_1, \ldots, y_m), y_{m+1})$$

DeMorgan's Duality

For suitable negation operations $N(a)$, such as $N(a)=1-a$x, T-norms $T(.,.)$ and T-conorms $S(.,.)$ are duals in the sense of the following generalization of DeMorgan's Law:

$$S(a,b) = N(\ T\ (N(a),\ N(b)\ ))$$

$$T(a,b) = N(\ S\ (N(a),\ N(b)\ ))$$

This duality implies that the extensions of the intersection and union operators cannot be independently defined and they should, therefore, be analyzed as DeMorgan triples $(T(.,.),\ S(.,.),\ N(.))$. Given a common negation operator like $N(a) = 1-a$, the selection of a T-norm $T(.,.)$ uniquely constrains the selection of the T-conorm $S(.,.)$.

Some typical T-norms $T(a,b)$ and their dual T-conorms $S(a,b)$ are the following:

$T_0(a,b)$	$= \min(a,b)$ if $\max(a,b)=1$ $= 0$ otherwise	$S_0(a,b)$	$= \max(a,b)$ if $\min(a,b)=0$ $= 1$ otherwise
$T_1(a,b)$	$= \max(0,\ a+b-1)$	$S_1(a,b)$	$= \min(1,a+b)$
$T_{1.5}(a,b)$	$= (ab)/[2-(a+b-ab)]$	$S_{1.5}(a,b)$	$= (a+b)/(1+ab)$
$T_2(a,b)$	$= ab$	$S_2(a,b)$	$= a+b-ab$
$T_{2.5}(a,b)$	$= (ab)/(a+b-ab)$	$S_{2.5}(a,b)$	$= (a+b-2ab)/(1-ab)$
$T_3(a,b)$	$= \min(a,b)$	$S_3(a,b)$	$= \max(a,b)$

These operators are ordered as following:

$$T_0 \leq T_1 \leq T_{1.5} \leq T_2 \leq T_{2.5} \leq T_3$$

$$S_3 \leq S_{2.5} \leq S_2 \leq S_{1.5} \leq S_1 \leq S_0$$

Detachment Operators

$S_3(a,b) = \max(a,b)$ Given a statement P, whose certainty value is located in the interval $[b,B]$, and an inference rule $P \rightarrow Q$, whose lower bounds for sufficiency and necessity are s and n, respectively, one can derive the boundaries for the certainty value of the conclusion Q by using the detachment operator. Such boundaries, denoted by $[N(Q), Pl(Q)]$, are represented by the interval $[T(s,b), S((1-n),B)]$, where $T(.,.)$ and $S(.,.)$ stand for any T-norm and its dual T-conorm. By using DeMorgan's identity, this interval can be rewritten as $[N(Q), Pl(Q)] = [T(s,b), 1-T(n,(1-B))]$. Therefore, the detachment operator can be uniquely defined by specifying a T-norm $T(.,.)$.

$P \qquad (P \rightarrow Q) \qquad (Q \rightarrow P) \equiv (\neg P \rightarrow \neg Q)$
$[b,B] \quad [s, \,] \qquad [n, \,]$

Q
$[T(s,b), S(B,1-n)]$

Proof:

Let: $b = N(P) \qquad B = Pl(P) = 1 - N(\neg P)$
$s = N(P \rightarrow Q) \qquad n = N(\neg P \rightarrow \neg Q)$

where N(A) and Pl(A) indicate the lower and upper bounds of the A's certainty, respectively.

The lower bound N(Q) can be obtained by applying Modus Ponens to the minor premise and the *sufficient* part of the inference rule:

$$P \text{ AND } (P \rightarrow Q) \Rightarrow Q$$

By using any T-norm $T(.,.)$ to represent the AND operator, we have

$$T(N(P), N(P \rightarrow Q)) = N(Q)$$
$$N(Q) = T(b,s)$$

The upper Bound Pl(Q) can be obtained by applying Modus Tollens to the minor premise and the *necessary* part of the inference rule:

$$\neg P \text{ AND } (\neg P \rightarrow \neg Q) \Rightarrow \neg Q$$

By using any T-norm $T(.,.)$ to represent the AND operator, we have:

$$T(N(\neg P), N(\neg P \rightarrow \neg Q)) = N(\neg Q)$$

Using the identity $N(\neg Q) = 1 - Pl(Q)$:

$$Pl(Q) = 1 - T(N(\neg P), N(\neg P \rightarrow \neg Q))$$

Again, using the identity $N(\neg P) = 1 - Pl(P)$:

$$Pl(Q) = 1 - T(1-Pl(P), N(\neg P \rightarrow \neg Q))$$

Using DeMorgan's identity $S(x,y) = 1- T((1-x),(1-y))$:

$$Pl(Q) = S(Pl(P), 1 - N(\neg P \rightarrow \neg Q))$$
$$Pl(Q) = S(B, (1-n))$$

The upper bound $Pl(Q) = S((1-n),B)$ correspond to the *implication* operator used in multiple-valued logics [20].
For $S(x,y) = S_3(x,y) = Max(x,y)$, the upper bound Pl(Q) becomes $Max\ (1-n,\ B)$, the Kleene-Dienes implication operator.
For $S(x,y) = S_2(x,y) = x+y-xy$, the upper bound Pl(Q) becomes $1-n+nB$, which has been called the Kleene-Dienes/Łukasiewicz implication operator.
For $S(x,y) = S_1(x,y) = Min(1, x+y)$, the upper bound Pl(Q) becomes $Min(1,\ 1-n+B)$, the Łukasiewicz implication operator.

Clearly, the interval $[T_1(s,b),\ 1-T_1(n,(1-B))]$ subsumes $[T_2(s,b),\ 1-T_2(n,(1-B))]$, which, in turn, contains the interval $[T_3(s,b),\ 1-T_3(n,(1-B))]$. The selection of the T-norm (and therefore the selection of the detachment operator) will determine the amount of ignorance (*width* of the interval) associated with the conclusion by the detachment operator.

The previous analysis of the detachment operator assumed that the conclusion is inferred from the minor and major premise by applying *modus ponens*. The symbol "→" that is present in the major premise $P \rightarrow Q$ (sufficiency) and $Q \rightarrow P$ (neccssity) represents the *material implication*.

An alternative interpretation of "→" is that of *conditioning*. Under this assumption, if the certainties of statements P and Q are given a probabilistic interpretation, then the boundaries for the certainty of Q is derived from a perturbation analysis of the probability formula:

$$p(Q) = p(Q \mid P) + p(Q \mid \neg P)\, p(\neg P)$$

Let: $b = N(P)$
$B = Pl(P) = 1 - N(\neg P)$
$s = N(Q \mid P)$
$S = Pl(Q \mid P)$
$r = N(Q \mid \neg P)$
$R = Pl(Q \mid \neg P)$

where N(A) and Pl(A) indicate the lower and upper bounds of p(A), the probability of A. Then:

$P \qquad P \rightarrow Q \text{ (i.e., } Q \mid P) \qquad \neg P \rightarrow Q \text{ (i.e, } Q \mid \neg P)$
$[b,B] \quad [s,S[\quad [r,R]$

Q
$[\min((sb+r(1-b)),(sB+r(1-B)),\ \ \max((Sb+R(1-b)),(SB+R(1-B))[$

When $p(Q \mid \neg P)$ is unknown (i.e., $[r,R[= [0,1[$), then:

Q
$[\min(sb,\ sB),\ \ \max((1- b +Sb),(1- B +SB))]$
$[s(\min(b,B)),\ \max(1- b(1-S)),(1- B(1-S))]$

since: $b \leq B$ and $\max(1\text{-}a,1\text{-}b) = 1\text{-}\min(a,b)$

$$[sb,\ 1\text{-}\min(b(1\text{-}S)),(B(1\text{-}S))]$$
$$[sb,\ 1\text{-}(1\text{-}S)(\min(b,B))]$$
$$[sb,\ 1\text{-}\ b(1\text{-}S)]$$
$$[sb,\ (1 - b + Sb)]$$

This result was reported by Ginsberg [29-31] and by Dubois and Prade [28].

Notice that under the assumption of ignorance about p(Q | ¬P) (i.e., [r,R]=[0,1]), the boundaries for the probability of Q are defined by $[T_2(s,b),\ S_2((1\text{-}b),S)]$ or equivalently by $[T_2(s,b),\ 1\text{-}T_2(b,(1\text{-}S))]$.

Parametrized Families of T-norms

The T-norms described in previous sections have different properties and characteristics. It is sometimes desirable to *blend* some of these operators, in order to smooth some of their effects. While it is always possible to generate a linear combination of two operators, this would imply giving up the associativity property. However, associativity is the most crucial property of the T-norms [21] since it allows the decomposition of multiple-place functions in terms of two-place functions. The correct solution is to find a family of T-norms that ranges over the desired operators. The proper selection of a parameter will then define the intermediate operator with the desired effect while still preserving associativity.

In a previous paper [5], six parametrized families of T-norms and dual T-conorms, originally proposed by Yager [25], Dubois and Prade [11], Hamacher [15], Schweizer and Sklar [21], Frank [12], and Sugeno [23], were discussed and analyzed. Of the six parametrized families, one family has been selected due to its broad coverage and numerical stability. This family, proposed by Schweizer & Sklar, is denoted by $T_{Sc}(a,b,p)$, where p is the parameter that spans the space of T-norms From T_0 to T_3. More specifically:

$$T_{Sc}(a,b,p) \quad = \text{MAX}\ \{0,\ (a^{-p} + b^{-p} - 1)\}^{-1/p} \quad \text{for } p \in [-\infty,\infty]$$

$$S_{Sc}(a,b,p) \quad = 1 - \text{MAX}\ \{0,\ [(1\text{-}a)^{-p} + (1\text{-}b)^{-p} - 1]\}^{-1/p} \quad \text{for } p \in [-\infty,\infty]$$

The following table indicates the value of the parameter for which this family reproduce the most common T-norms $\{T_0, \ldots, T_3\}$.

TABLE 1:
Ranges of values of parameter p
for $T_{Sc}(a,b,p)$

$T_{Sc}(a,b,p)$	T-norm
p	
$\rightarrow -\infty$	T_0
-1	T_1
.	$T_{1.5}$
$\rightarrow 0$	T_2
.	$T_{2.5}$
$\rightarrow \infty$	T_3

The table for the T-conorms is identical to the above except for the header, where the families of T-norms are replaced by the corresponding families of T-conorms, and the last column, where the T-norms are replaced by their respective dual T-conorms, i.e., T_0 by S_0, etc.

These families of norms can specify an infinite number of calculi that operate on arguments taking *real number* values on the [0,1] interval. This *fine-tuning* capability would be useful if we needed to compute, with a high degree of precision, the results of aggregating information characterized by very precise measures of its uncertainty. However, when users or experts must provide these measures, an assumption of *fake precision* must usually be made to satisfy the requirements of the selected calculus.

Equivalence Classes Among T-norms

Because of the difficulties in eliciting precise and yet consistent numerical values from the user or expert, the use of term sets has been proposed. Each term set determines the finest level of specificity (i.e., the *granularity*) of the measure of certainty that the user/expert can *consistently* provide. This granularity limits the ability to differentiate between two similar calculi. Therefore, only a small finite subset of the infinite number of calculi produces notably different results. The number of calculi to be considered is a function of the uncertainty granularity.

This result has been confirmed by an experiment [5] where eleven different calculi of uncertainty, represented by their corresponding T-norms, were analyzed. Figure 1 illustrates a plot of the eleven T-norms, where the parameter p in Schweizer's family has been given the following values: -1, -0.8, -0.5, -0.3, 0 (in the limit), 0.5, 1, 2, 5, 8, ∞ (in the limit). This plot shows the space of T-norms that produce the same result K, for K=0.25, 0.5, 0.75.

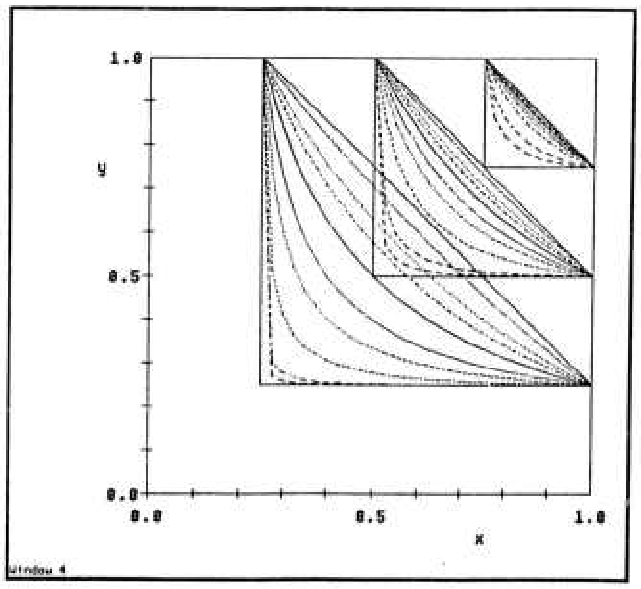


FIGURE 1: Space of T-norms
$T_i(a,b) = K$, for K=0.25, 0.50, and 0.75

The eleven calculi were used with three term sets containing five, nine, and thirteen elements, respectively. For each of the three term sets, the T-norms were evaluated on the crossproduct of the term set elements, generating the closure of each T-norm. Each closure was compared with the closure of the *adjacent* T-norm and the number of differences were computed. The T-norms that did not exhibit significant differences were considered similar enough to be equivalent for any practical purpose. A threshold value determined the maximum percentage of differences allowed among members of the same equivalence class. Only three calculi generated sufficiently distinct results for those term sets that contained no more than nine elements. Five calculi were required when a larger term set (containing thirteen elements) was used.

The three calculi required in the first case were defined by the following operators:

$$(T_1(a,b),\ S_1(a,b),\ N(a))$$
$$(T_2(a,b),\ S_2(a,b),\ N(a))$$
$$(T_3(a,b),\ S_3(a,b),\ N(a))$$

where $N(a)$ is the negation operator $N(a) = 1-a$, and $T_i(a,b)$ $(S_i(a,b))$ are the T-norms (DeMorgan duals T-conorms) defined by the Schweizer family $T_{Sc}(a,b,p)$ $(S_{Sc}(a,b,p))$ for the following three values of p:

$$
\begin{array}{lll}
p = -1 & T_1(a,b) = max(0,a+b-1) \\
 & S_1(a,b) = min(1,a+b) \\
p \to 0 & T_2(a,b) = ab \\
 & S_2(a,b) = a+b-ab \\
p \to \infty & T_3(a,b) = min(a,b) \\
 & S_3(a,b) = max(a,b)
\end{array}
$$

In addition to the three operators defined above, the five calculi (required in the second case) need the following T-norms:

$$
\begin{array}{ll}
p = -0.5 & T_{Sc}(a,b,-0.5) = \max(0, a^{0.5}+b^{0.5}-1)^2 \\
 & S_{Sc}(a,b,-0.5) = 1\text{-}\max\{0, [(1-a)^{0.5}+(1-b)^{0.5}-1]\}^2 \\
p = 1 & T_{Sc}(a,b,1) = \max(0, a^{-1}+b^{-1}-1)^{-1} \\
 & S_{Sc}(a,b,1) = 1\text{-}\max\{0, [(1-a)^{-1}+(1-b)^{-1}-1]\}^{-1}
\end{array}
$$

Table 2 illustrates the equivalence classes.

CONTROL LAYER

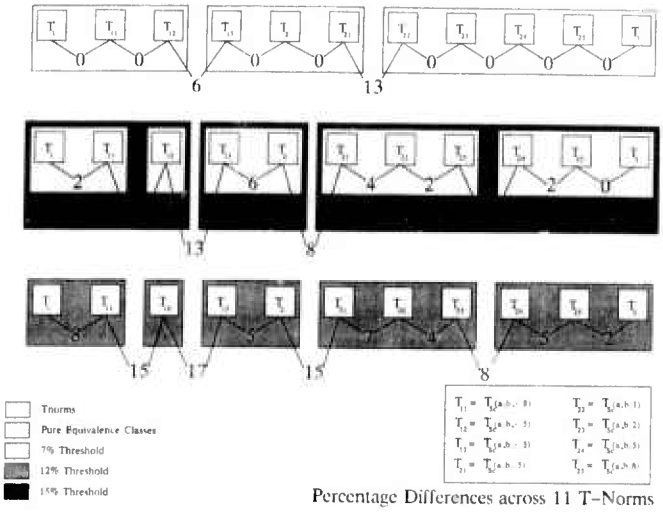
Selecting Uncertainty Calculi

The selection of the most appropriate uncertainty calculus depends on how well the calculus characteristics fit the local assumptions described by the context information. To accomplish this, it is essential to analyze the properties of the calculi used in the inference layer.

Since T-conorms and detachment operators can be expressed as functions of the negation operator and the T-norms, to understand the meaning of each calculus it is enough to analyze its underlying T-norm operator. A first interpretation suggests that T_1 is appropriate to perform the intersection of lower probability bounds. T_3 is appropriate to represent the intersection of upper probability bounds. T_2 is the classical probabilistic operator that assumes *independence* of the arguments; its dual T-conorm, S_2, is the usual *additive* measure for the union.

TABLE 2:

Percentage Differences Among the Eleven T-norms Applied to the Three Term Sets



Percentage Differences across 11 T–Norms

Figure 2 provides a geometric description of the meaning of the three T-norms. The figure illustrates the result of T_1 (0.3, 0.8), T_2 (0.3, 0.8), and T_3 (0.3, 0.8). T_1 captures the notion of *worst case*, where the two arguments are considered as *mutually exclusive* as possible (the dimensions on which they are measured are 180° apart). T_2 captures the notion of *independence* of the arguments (their dimensions are 90° apart). T_3 captures the notion of *best case*, where one argument attempts to *subsume* the other one (their dimensions are collinear, i.e., 0° apart).

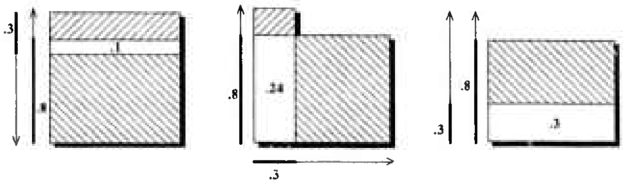


FIGURE 2: Geometrical Interpretation of $T_1(0.3, 0.8)$, $T_2(0.3, 0.8)$, and $T_3(0.3, 0.8)$

The other two T-norms, $T_{Sc}(a,b,-0.5)$ and $T_{Sc}(a,b,1)$, can be used when the information is known to be *mildly* negative or positive correlated, without requiring the drastic extremes of mutually exclusiveness or subsumption. The two additional calculi provide intermediate degrees of pessimism and optimism in the range of worst case/best case analysis.

Measuring Ignorance and Consistency

The numerical slots that provide control information are: the *measure's uncertainty*, the *entropy function*, and the *inconsistency measure*.

The measure's uncertainty is defined as the area under the curve delimited by the (fuzzy) interval [N(A),Pl(A)]. When N(A) and Pl(A) are crisp numbers such measure is simply the difference Pl(A)-N(A) [13].

The entropy function is defined as: $f(x) = -K(x \log(x) + (1-x) \log(1-x))$ where K is a normalizing constant (e.g., $K = 1/\log(2)$ normalizes the range of $f(x)$ to the interval [0,1]). The evaluation of the quality of the information is given by the interval [f(N(A)), f(Pl(A))]. When N(A) and Pl(A) are fuzzy numbers, a set of closed-form formulae [2,5], based on the extension principle [26], can be used to evaluate such a function.

The detection of inconsistency occurs when N(A) > Pl(A). A measure of such inconsistency is given by the difference N(A)-Pl(A).

CONCLUSIONS

We have proposed a layered architecture to define the representation, inference, and control of uncertain information. This architecture is summarized in Figure 3.

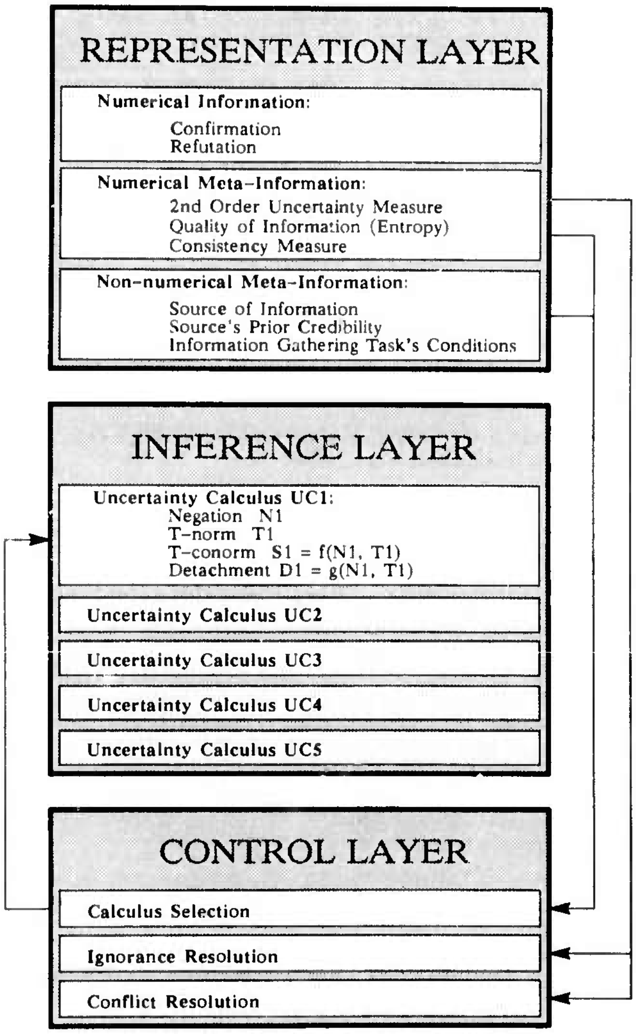


FIGURE 3: Three Layer Architecture

In the *representation* layer we have advocated the use of frame-like structures, capturing uncertainty information, such as the degrees of confirmation and refutation, as well as uncertainty meta-information such as the information quality and measure's precision. The uncertainty information is used and combined in the inference layer by an appropriate uncertainty calculus. The uncertainty meta-information is used in the control layer to select the appropriate uncertainty calculus, based on *local* (i.e., contextual), rather than *global* assumptions. We have proposed the use of linguistic term sets of likelihood statements to anchor the input granularity for the numerically valued slots.

In the *inference* layer, we have shown that any truth functional uncertainty calculus can be represented (and analyzed) in terms of its underlying T-norm, an associative, commutative operator that extends the concept of set intersection to multiple-valued logics.

The truth functionality of the calculi used in this layer entails low computational complexity: the aggregated certainty of any logic expression can be computed *directly* from the certainty of the individual components. The associativity of the calculi guarantees the recursive decomposition of multiple-arguments aggregation into two-argument aggregations. This property is extremely useful when, by decomposing large problems into smaller sub-problems, we can then make use of special hardware (custom VLSI chips) to concurrently evaluate the sub-expressions ad aggregate the partial results.

We have shown that, for a fixed input granularity, the infinite number of uncertainty calculi (T-norms) can be reduced to at most five distinct equivalence classes. This fact allows us to individually study the calculi characteristics and to understand the assumptions that the use of each calculus would entail (mutually exclusiveness, uncorrelation, subsumption).

In the *control* layer, we have proposed to select the appropriate calculus based on each calculus' properties (context independent information) and on the available meta-information describing the situation (context dependent information). Unlike the theory of endorsements, where a combinatorial problem occurs when the semantic rules (determining how endorsements are aggregated) must be defined for *every value combination*, the selection policies set (meta-rules) to be defined in this layer is relatively small. The selection policies set must only determine which of the three (or five) calculi, defined in the inference layer, is the appropriate one for any given case. Usually these cases are grouped in hierarchical contexts (subclasses) so that the selection policies can be assigned to the context nodes and inheritance methods can be used to pass the assignment to the rule instances. Once a calculus has been selected, the combining rules for every value combination are uniquely determined.

Rather than embedding conflict resolution in the inference layer, as it is the case for other approaches, we have proposed to perform conflict detection and resolution in the control layer. This is motivated by the fact that resolving conflicts or ignorance is part of the resource allocation problem which is best done at this layer: deciding if, when, and how to eliminate conflicting information depends on various factors, such as the magnitude of the conflict, on the goal's sensitivity to the information, on the cost of gathering further information, on the likelihood of succeeding in gathering such information, and on the cost of failing in such a task. The cleaner solution is to declaratively express these contextually-scoped conflict policies in the control layer.

REFERENCES

[1] Beyth-Marom, R., (1982). How Probable is Probable? A Numerical Taxonomy Translation of Verbal Probability Expressions, *Journal of Forecasting*, Vol. 1, pp. 257-269.

[2] Bonissone, P.P., (1980). A Fuzzy Sets Based Linguistic Approach: Theory and Applications, *Proceedings of the 1980 Winter Simulation Conference*, edited by T.I. Oren, C.M. Shub, P.F. Roth, pp. 99-111, Orlando, December 1980. Also in *Approximate Reasoning in Decision Analysis* edited by M.M. Gupta, E. Sanchez, pp. 329-339, North Holland Publishing Co., New York, 1982.

[3] Bonissone, P.P. & Tong, R.M., (1985). Editorial: Reasoning with Uncertainty in Expert Systems, *International Journal of Man-Machine Studies*, Vol. 22, No. 3, pp. 241-250, March 1985.

[4] Bonissone, P.P. & Brown, A.L. (1985). Expanding the Horizons of Expert Systems To appear in the *Proceedings of the Second International Conference on Artificial Intelligence Technologies, Expert Systems and Knowledge Engineering*, Ruschlikon, Switzerland, 25-26 April 1985.

[5] Bonissone, P.P. & Decker, K.S., (1985). Selecting Uncertainty Calculi and Granularity: An Experiment in Trading-off Precision and Complexity, *Proceedings of the Workshop on Uncertainty and Probability in Artificial Intelligence*, pp. 57-66, University of California, Los Angeles, August 14-16, 1985. To appear in *Uncertainty in Artificial Intelligence*, L. Kanal & J. Lemmer (Eds.), North-Holland, 1986.

[6] Bonissone, P.P., (1986). Plausible Reasoning: Coping with Uncertainty in Expert Systems, to appear in the *Encyclopedia of Artificial Intelligence*, Stuart C. Shapiro (Editor), John Wiley & Sons Publishing Co., New York, New York, 1986.

[7] Brown, A.L., (1985). Modal Propositional Semantics for Reason Maintenance Systems. *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, California, August, 1985.

[8] Cohen, P.R. & Grinberg, M.R., (1983). A Theory of Heuristics Reasoning about Uncertainty, *The AI Magazine*, pp. 17-23, Summer 1983.

[9] Cohen, P.R. & Grinberg, M.R. (1983). A Framework for Heuristics Reasoning about Uncertainty, *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pp. 355-357, Karlsruhe, West Germany, 1983.

[10] Doyle, J., (1983). Methodological Simplicity in Expert System Construction: The Case of Judgements and Reasoned Assumptions, *The AI Magazine*, Summer 1983, Vol. 4, No. 2, pp.39-43, 1983.

[11] Dubois, D. & Prade, H., (1980). New Results about Properties and semantics of fuzzy set-theoretic operators, in *Fuzzy Sets: Theory and Applications to Policy Analysis and Information Systems*, P.P. Wang & S.K. Chang (Eds.), pp. 59-75, Plenum Press, New York.

[12] Frank, M.J., (1979). On the simultaneous associativity of F(x,y) and x+y-F(x,y), *Aequationes Mathematicae*, Vol. 19, pp. 194-226.

[13] Garvey, T.D., Lowrance, J.D. & Fischler, M.A. (1981). An Inference Technique for Integrating Knowledge from Disparate Sources, *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pp. 319-325, Vancouver, B.C , Canada, (1981).

[14] Genesereth, M.R (1982). An Overview of MRS for AI Experts, Stanford Heuristic Programming Project Memo HPP-82-27, Dept. of Computer Science, Stanford University.

[15] Hamacher, H, (1975). Uber logische Verknupfungen unscharfer Aussagen und deren zugehorige Bewertungs-funktionen, in *Progress in Cybernetics and Systems Research, Vol. II*, R. Trappl & F. de P. Hanica (Eds.), pp. 276-287, Hemisphere Pub. Corp., New York.

[16] McAllester, D.A., (1980). An Outlook on Truth Maintenance. MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts.

[17] McDermott, D. & Doyle, J. (1980). Non-Monotonic Logic I, *Artificial Intelligence*, Vol. 13, pp. 133-170.

[18] McDermott, D. (1982). Non-Monotonic Logic II: Non-Monotonic Modal Theories, *Journal of the Association for Computing Machinery*, Vol. 29, pp. 33-57.

[19] Phillips, L. & Edwards, W. (1966). Conservatism in a simple probability inference task, *Journal of Experimental Psychology*, Vol. 72, pp. 346-354.

[20] Rescher, N., (1969). *Many-valued logics*, Mc-Graw Hill, New York, New York, 1969.

[21] Schweizer, B., Sklar, A., (1963). Associative Functions and Abstract Semi-Groups, *Publicationes Mathematicae Debrecen*, Vol. 10, pp. 69-81.

[22] Shafer, G. (1976). *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, NJ.

[23] Sugeno, M., (1977). Fuzzy Measures and Fuzzy Integrals: a Survey, in *Fuzzy Automata and Decision Processes*, M.M. Gupta, G.N. Saridis & B.R. Gaines (Eds.), pp. 89-102, North Holland, New York.

[24] Szolovits, P. & Pauker, S.G., (1978). Categorical and probabilistic reasoning in medical diagnosis, *Artificial Intelligence Journal*, Vol. 11, pp. 115-144

[25] Yager, R., (1980). On a General Class of Fuzzy Connectives, *Fuzzy Sets and Systems*, Vol. 4, pp. 235-242.

[26] Zadeh, L.A., (1975) The concept of a linguistic variable and its application to approximate reasoning, Part I, *Information Sciences*, Vol. 8, pp. 199-249; Part II, *Information Sciences*, Vol. 8, pp. 301-357; Part III, *Information Sciences*, Vol. 9, pp. 43-80.

[27] Zimmer, A.C. (1985). The Estimation of Subjective Probabilities via Categorical Judgments of Uncertainty, *Proceedings of the Workshop on Uncertainty and Probability in Artificial Intelligence*, pp. 217-224, UCLA, Los Angeles, California, August 14-16, 1985. To appear in *Uncertainty in Artificial Intelligence*, L. Kanal & J. Lemmer (Eds.), North-Holland, 1986.

[28] Dubois, D. & Prade, H (1985). Combination and Propagation of Uncertainty with Belief Functions - A Reexamination, *Proceedings of the Ninth Internationa Joint Conference on Artificial Intelligence*, IJCAI85, pp. 111-113, August 18-23, 1985, Los Angeles, California.

[29] GINSBERG, M.L. (1984). Non-Monotonic Reasoning Using Dempster's Rule, *Proceedings of the National Conference on Artificial Intelligence*, pp. 126-129, Austin, Texas, August 6-10, 1984.

[30] GINSBERG, M.L. (1984). Analyzing Incomplete Information, Heuristic Programming Project Report No. HPP 84-17, June 1984.

[31] GINSBERG, M.L. (1984). Implementing probabilistic reasoning, Heuristic Programming Project Report No. HPP 84-31, June 1984.

MONAD*

A HIERARCHICAL MODEL PARADIGM FOR REASONING BY ANALOGY

Interim Report †

Gilbert B. Porter, III

General Electric Company
Corporate Research and Development
Schenectady, NY 12345
ARPANET: GBPorter@GE-CRD

INTRODUCTION

Reasoning by Analogy is a two edged sword: on one hand it attempts to solve problems that are beyond the scope of the knowledge contained in the Knowledge Base; while on the other, it provokes the insidious problem of searching a universe of potential candidate matches constructed under the guise of similarity. This interim report summarizes some of the work in progress to strike a balance between these two opposing forces. First, we will describe the philosophy behind the decisions relating to the overall architecture of the system: its knowledge representation scheme, its search strategy, and its analogical method. Then we will offer a few preliminary results and a status report.

PHILOSOPHY

The class of problems which are addressed by the method of reasoning by analogy described in this report have the characteristic that their solutions are not directly contained in the Knowledge Base either in the form of a fact, or as a belief which is directly deducible from a set of rules applied to the facts. We refer to these problems as **novel** with respect to the knowledge base meaning that the solution must be derived from available solutions by one or more applications of what we may loosely refer to as a **substitution**. In performing a substitution, the reasoning system must hypothesize from uncertain evidence that, in deriving the required goal, a known reasoning step can be modified to produce a new step which is only weakly justified by the hypothesis and the knowledge base.

To perform reasoning in which modifications may be made during the deductive process, the reasoning system must very carefully address the problem of search. The main objective of this work is to devise a reasoning method which can derive solutions to these novel problems by constructing near miss solutions contained in the knowledge base in order to confine the search. We will address three issues regarding the machinery required to reason in this manner: the knowledge representation scheme, the search strategy used by the model building scheme, and the analogical method. In this section on philosophy, we informally describe the approach and requirements for analogical reasoning, an overview of the problem solving approach, followed by a discussion of the goals and requirements of the knowledge representation and search strategies. Although work is well along in producing an implementation of the system we will describe, the supporting ideas are, by no means, immutable: it is quite likely that difficulties will inspire alterations.

Motivation

In our work, the term analogy will be used in a fairly broad sense: the comparison of problem solutions based on a notion of similarity for the purpose of recognition of solutions or synthesis of new solutions. As we have said, our work on reasoning by analogy will concentrate on finding solutions to problems which are not directly contained in the knowledge base. The motivation for this type of reasoning is fairly simple: as expert system technology is applied to more complex problems, it becomes less practical to develop complete and consistent knowledge bases for these problems. One alternative is to build multiple cooperating expert systems that share in the solution of a multidisciplined problem. Each expert could be quite complex but restricted to its specialty. We believe that this approach requires careful consideration of the communication between the systems to make them functional and implies a design coupling between the systems that would have hopefully been avoided by the choice of a multi-expert architecture. The alternative we have chosen is to build an abstract problem solver which produces solutions from approximate information.

Much of the previous work on analogical reasoning has been based on the method of matching the structures representing the problem to a representation of a candidate solution. Associated with the matching procedure is some sort of measure of similarity which is used rank the goodness of a solution and, perhaps, to order the solution search. One serious problem which has been encountered attempting this sort of reasoning is that the representation is very important in evaluating the similarity. It is hard to devise a general representation scheme in which an often poorly understood problem/solution can be uniformly

* This work was partially supported by the Defense Advanced Research Projects Agency (DARPA) contract F30602-85-C-0033. Views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official opinion or policy of DARPA or the U.S. Government.

† This report summarizes a forthcoming paper describing the details of this work.

expressed. Additionally, as the problems become complex, there can be an excess of unimportant information in the representation which can cause the search for a solution to be unduly complicated.

By building a system which uses approximate information, it is possible to span a larger class of solvable problems with less overall information. A major dra:vback to this type of system is that it is nearly impossible to know how to debug or expand a very large system due to the tenuous coupling between facts and solutions. To address this issue, we propose a representation which intends to capture the conceptual underpinnings of the facts in the knowledge base. We have focused on the problem of making the representation scheme flexible and highly tuned to each specific problem.

Constructing a problem specific model is done dynamically as reasoning proceeds. The basis for the specific problem model is a **hierarchical model** definition which captures many levels of detail from various points of view. The intent of dynamic model construction is to provide a simple model of the problem from which analogies may be drawn. Creating a very simple, uncluttered model reduces irrelevant details that can hopelessly confuse the search. In a corresponding manner, the matching procedure avoids using detailed differences to measure similarity. Instead, it tries to move to the maximum level of abstraction before making a comparison. This has the benefit of making concepts important while prohibiting "un-semantic" comparisons. An un-semantic comparison is one for which there is no conceptual founding in the knowledge base. The most blatant human example is the pun, but there are also many more subtle and purposeful kinds of associations such as rhyming for poetry, thesaural inference, and seemingly unconnected insight, all of which may occur, initially, by chance but may be learned and practiced. When it is desirable to make such undirected comparisons, a mechanism is provided for creating arbitrary associations but at a much higher cost (as we believe it should be).

In the next sections, we will describe the overall strategy embodied in the reasoning system. Much of the discussion pertains to the modeling scheme which is the backbone of the system and deserves the majority of the attention.

The Problem Solving Strategy

The analogical method described here is embedded in a problem solving system. In order to restrict the scope of this work, we have chosen to bypass some issues and give only cursory mention to others. The important supporting philosophies are those regarding the overall architecture, the construction of the working representation for performing analogy, and the search strategy. As a side issue, the philosophy regarding model content has raised some interesting questions which we will report.

The approach embodied in the problem solver is a **multi-staged decomposition** procedure using a hierarchical model paradigm as the representation scheme. The reason for choosing a multi-staged decomposition is based on several observations. First, we observe that the known solutions to this class of problem are few and, usually, quite complex. If we consider the notion that a solution might be composed from a common, flexible set of techniques rather than a collection of new insights, then the multi-staged approach appears to be a more facile method for combining ill-mated techniques than a more tightly coupled integration method.

Secondly, we observe that complex **recognition** problems are often solved by starting with a set of observables, which we shall call **features**. Features are of two kinds: natural features, which are usually associated with the physical characteristics of the involved objects, and process derived features, which have no observable correspondence but are essential to the implementation of the associated recognition process. The primitive features are combined and recombined into more complex process derived features in a staged sequence which reduces irrelevant information. The development cf these features is usually ordered due to the nested feature composition. Additionally, due to the explosion of feature combinations, it is necessary to restrict the number of features which may be composed within a given stage. From the number of typically computed features the staged approach is again suggested.

And finally, we observe that even if we were to try to use an unstaged solution, the potential connectivity of the various modules required to express the solution would be very large without some restriction which we propose as a function of the staging.

As a result of this philosophy, we have chosen to use a relatively simple reasoning strategy which relies on complexity of the model structure for richness. The strategy may easily be repeatedly applied at each stage to create new sets of features. Preliminary results have indicated that a useful class of problems is solvable by this specialized method (this is a good sign since we are trying to devise a programming paradigm for analogical reasoning). It is encouraging that the same method appears to have application to a variety of problem categories such as planning, design, and diagnosis. Differences in the approach to these problems is controlled using the notion of **point of view** which orders the way in which information is portrayed by the knowledge base rather than a difference of method. The related, but orthogonal notion of context, meaning the semantics of a particular problem specification, will not be addressed here as it would have little bearing on the details of our method.

And so, due to this choice of architecture, the notion of building a general problem solver need not be addressed, rather, we will concentrate on techniques of using a specialized method to solve a variety of problems. If the method is to be simplistic, then to gain the necessary variety in solution capability, it must be applied successively under very select conditions imposed by the goals. An example will illustrate our point thus far.

Consider the problem of analyzing a visual scene: the image understanding problem. This is a hard class of problems and has achieved the most success for very constrained or restricted problems. Traditionally image processing has been based on the notion of extracting features from the image data, combining these, and repeating the process with higher level features until a high level representation of the scene is obtained. The high level representation may be matched against some models to derive the scene content. This is an admittedly terse, but not terribly inaccurate summary of one kind of image processing process. As we would expect, many possible feature representations may be derived. For example, the natural features could be objects, subparts, collections, and other geometrically related features, while the process related features would be edges, corners, edge direction, intensity, intensity derivatives, regions, boundaries, and the like. To avoid the combinatorics, a decoupling of the separable processes and the associated data is necessary. By this we mean that the interfaces between the various processes must be organized around the features as the communications symbology, and the chosen features must adequately represent the content of the data. Further, the mechanisms which drive the extraction of features such as convolutions, grammars, and the like, must be very efficient and closely tuned to the expected image behavior to be effective in any real implementation. We would like to dynamically construct a model which is carefully matched to the required observabies and internal states as we proceed. Then we must ask from what basis the model is constructed and how it is possible to derive the model for a problem which is initially unknown to the system? To understand this process, we must first describe the knowledge representation scheme in some detail.

Knowledge Representation

As we have said, knowledge is represented using a hierarchical model paradigm. Models are constructed and used from a specific **point of view** which may differ from construction to use. For example, a set of models for an object may be constructed from the point of view which represents the conceptual notion of how the object might be designed or constructed. In actual use, the models may be more effective if viewed from a different point of view. We wish to recognize and understand this issue before proposing a solution. Thus, the current philosophy is to directly encode the point of view information in order to make it explicit both to the reasoning program, and to ourselves for further examination.

From a specific point of view, then, the set of models representing an object may be viewed as a succession of vertical layers that are ordered such that, in some sense, each layer is a more complete or complex description of the expected behavior of the object from the stated point of view. The top layer depicts the normative state and the confirming observables. For example, if the behavior of a lead-acid storage battery (whose function is to supply power to a specific electrical system in a tank) is to produce a certain voltage at the output terminals, then the corresponding top level model of the battery is one in which the required voltage is present - perhaps completely independent of the load, the charge condition, the electrolyte condition, the ambient temperature, and many other important, but secondary parameters. The observable is the voltage and is represented either by a procedure for obtaining its measure, or a pointer to another model for devising such a procedure.

It is our philosophy that the model scheme should portray the expected behavior and perhaps some embedded functionality at a particular level of the model and from a specific point of view. If the observed or desired behavior is different than that predicted by the model, then, either the model is insufficient in detail, or the model has been incorrectly constructed. At any particular level, these two faults are indistinguishable and the reasoning system goes about trying to construct a more detailed model which predicts the correct behavior. In answering a query about what is known by the knowledge base, the reasoning system never alters the model. It is always assumed to be correct at its own level of detail. This is an important issue in that it is the foundation of the constructive procedure for a known solution. The precise relation between models is being formalized and will appear as a future result.

So, for our example of the lead-acid battery, at the top level, we simply expect the voltage to be present at the output terminals. If we are designing a circuit in which no further information is required, then the query would only access the top level. Similarly, if we are diagnosing a failure, then if the voltage is not present, there is no explanation for the fault at this level, and the model is invalid (we will explain later what is to be done). In a planning situation, if we wish to install a new battery in the tank, then observing the output voltage may obviate any further steps to validate performance. Specifications involving other observables which are not included in the top level model, simply invalidate the utility of the top level model: we re-iterate, the current level model is considered to perfectly explain the expected behavior until a model failure is determined. The search strategy decides how to correct the failed model.

In describing the search strategy, we will address two issues: the local creation of a model during a stage, and the global process of forming an analogical solution to a problem.

Search Strategy - Forming a Local Problem Model

The process of forming a local problem model is intended to construct a very tightly tuned representation of **only** the information required to solve the immediate problem - at least from the standpoint of search. The algorithm is intent on being very frugal about adding new information and, thus, the overly complex model paradigm.

Continuing with our example, let us examine what occurs when a model failure is discovered. Suppose we are diagnosing the tank electrical system and find that the voltage on the output terminals is out of specification. Our top level model does not predict this behavior and, to proceed, we must construct a more detailed model to account for this performance. Assume that the next level model contains information on the voltage-current behavior of the battery. Simplistically, we might just add this knowledge to our current understanding of the battery to conclude that some check of the loading conditions is relevant. There are two difficulties with this approach. First, this may not be the most likely fault; perhaps checking the charge condition is a better diagnostic method. Secondly, a more serious flaw is that the new information may be in conflict with the previously expected behavior portrayed by the top level model.

To deal with the first problem, we cast each problem class in the framework of a specific point of view. As a matter of choice, we could attempt to deduce the point of view from some sort of specification, but, for our work, this appears to be off the track. Hence, as we have said, we have specifically coded the point of view for each class. In addition, we have attempted to code the point of view implicit in the model structure as it is presented. We will evaluate this method for its facility in guiding the interpretation of a design oriented model to be used for a planning and a diagnosis task. The utility of this approach remains to be shown as the implementation proceeds. As a matter of philosophy, we feel it is an important area which should not fail to be addressed.

The second difficulty appears to have deep implications. It is certainly predictable that model conflicts might creep into the model code as a matter of course. The issue we are addressing here is that we may choose to create conflicting models at different levels simply to hide irrelevant details at the higher levels. Thus far, we have found this to be a valuable asset. For example, the voltage/current relation above could have been modeled at the top level as allowing infinite current with no drop in voltage - no internal battery impedance. These simplifying assumptions such as ignoring complex impedance, ignoring friction, ignoring inertia, an a whole host of others, have been used effectively in problem solving by humans. Since it is unreasonable for the system to constantly check for model consistency, we have chosen to sidestep the problem by annotating the differences in the models to avoid complex inheritance methods. For our first implementation, any information replicated at a lower level subsumes the inherited information. Complex subsumptions will be directly noted and not deduced. At first glance, this does not seem to be the right approach, but it will serve to create instances of the problem until a more correct approach is understood.

Global Control - Finding Known Solutions

Let us, for a moment, step back from our example and look at the overall search process. Each stage of the solution is a reversible process which builds a local model of the problem domain using two strategies for search which are oriented toward a specific **point of view**. The search strategies are the equivalent of forward and backward chaining and represent **recognition** and **synthesis**. The point of view allows the application of these two strategies to be ordered in such a way that particular goal methods are observed.

Now suppose that, in the process of searching for a problem solution in the knowledge base, the available information fails to satisfy the given goal. The results of the search leave us, if the notion of point of view is successful, with a near miss solution embedded somewhere in the history of the search. Two questions arise at this point: 1) how do we identify the closest or set of closest misses, and 2) how can we modify one of the members of this set to produce an acceptable solution? In order to consider these notions, we will first describe some details of the model scheme.

A STRUCTURED MODEL REPRESENTATION PARADIGM

A **model** is an abstraction for representing the class of its instances. It is internally consistent but may portray conflicting beliefs that are differentiated by their **context**. For a given **point of view**, a model contains four specification components. As we have said, the *function* defines the intentional purpose of the model including side effects. This component of a model is the primary link between associated ideas. Function is predominantly represented in a hierarchy for genetically related classes, along with cross links which depict associations. Associations may be at different levels since the functional information propagated across any of the links must undergo a transformation before it can be used to form an analogy. For example, the top level model of the battery model would depict it as a device for the storage of electrical energy and as a device for supplying electrical energy. In the same hierarchy, other kinds of energy storage devices would also be linked. Cross links would account for less conceptual associations such as other things that use lead or whatever. Not all intents and purposes can be accounted and thus the need for analogical methods. It may be necessary to perform fairly wide ranging search to form (previously) unlinked associations. We intend that these should be strictly confined by the search procedure. As an example, clearly the battery could be used as a door stop, a boat anchor, or as a flower planter with a little ingenuity. These are not expected uses and would not be accounted. On the other hand, the model for an (electrical) resistor would usefully include the electrical definition as well as the side effect of producing heat.

The model *interface* defines the inputs, outputs, states, and properties exhibited by its instances. This is a, more or less, conventional black box representation which also serves the purpose of identifying the observables. By this we mean those quantities which are somehow measurable by the enquirer. The plan for measuring these is contained in the behavior description, described shortly, and may be either a procedure or a complex plan which invokes other models. We see the need to supply an ordering mechanism for acquiring this information but it is not clear how it will be represented. Currently the notion of an additional model for each process is favored. For the battery, the voltage characteristics would be described along with a procedure or pointer to another model which describes how to measure it. Incidently, other properties, such as physical characteristics, fall under another point of view and so it is possible to deduce the idea of using the battery as a boat anchor within the same model. Similarly, for the resistor, the notion of using it as a heater can be gotten from a side effect directly included in the functional specification. More complex associations use the functional association links.

The model *composition* defines the internal structure of the model in terms of other models or components and their interconnections. This is the block diagram of the model internal structure. It is the vehicle which allows the search process to find more detail when the model fails to explain the current information. Here, we will highlight the distinction between our definition of a *model* and a *component*. A *component* has exactly the same composition as a model but it is designated to be able to act as a primitive concept in that it is self-sufficient without reference to its composition. In a crude way, components delineate the natural modularity present in the structure of the physical world (i.e. large separation of physical effects) which we mentioned earlier. This type of definition provides the means to avoid using "quantum mechanics" when analyzing a macroscopic physical situation; or to avoid using Maxwell's equations when evaluating a simple electrical circuit. It alerts the reasoning system to the natural separation of treatment which occurs in most scientific disciplines.

Finally, the *behavior* defines the relationships between the states, inputs, outputs, and properties which define the external appearance of the model in the interface specification above. It defines any necessary action procedures either within the model or within another model to determine how the terminal action processes get actual work done.

SIMILARITY - FORMING AN ANALOGY

Let us return, now, to the two questions which arise after we have failed to find an acceptable solution in the knowledge base. We must be able to determine how to identify the closest or set of closest misses, and how to modify one of the members of this set to produce an acceptable solution.

We have chosen the strategy that each failure will be traced along the chain of supporting models to the point that is most abstract, but still embodies the failure. We will refer to the chain of models from this most abstract point to the leaf model as the **failure chain.** Notice that the failure may not stem from the top level since new details may be introduced at any point. Contained in the function definition of the top model in the failure chain is the abstract description of the conceptual functionality of the object of the model. From this model down the failure chain to the leaf model, is an ordered set of disrupting concepts: the top model being the most desirable since it has the strongest conceptual theory for the failure and also the fewest potential search nodes. Each of the function definitions in the models along the chain points to the ISA hierarchy of related models. This ranking is the first factor in the measure of similarity. The second factor relates to the use of cross association links from each model along the failure chain. It is not yet clear how to choose between proceeding down the failure chain and/or across the association links to propose new avenues for search.

Once we have decided to evaluate a new model to replace part or all of the failure chain, we must decide how to modify it to effect a solution. We have approached this issue by assuming that the symbolic terms in the proposed model will not be directly compatible with the models in the failure chain. An equivalent of strong typing of these terms are described in auxiliary models which provide methods to perform translations. So, for example, if we were trying to find a mechanical component to perform a desired function by analogy of mechanical to electrical systems, then the required translation of terms would be based on the models for these term equivalences for a given point of view. Latitude for proposed modifications is not arbitrary and must be deduced from the term models.

TEMPORAL DEDUCTIVE MAINTENANCE

Since we allow modifications to be made (locally) to the facts during the reasoning process, the (local) appearance of the knowledge base is non-monotonic. Thus, the reasoning steps are not reflexive and the implication is that justifications for facts can vanish. Since we wish to allow this type of reasoning in order to perform analogy, then to deal with this problem, we define the notion of a weak justification as one which is grounded in a time (i.e. event frame) prior to the current one and not grounded in the current one.

A utility called the **temporal deductive maintenance system**, (TDMS), manages the state of the knowledge base to track these changes over time and maintain (relatively) efficient updates. It is a poor man's reason (or truth) maintenance system along with the appropriate machinery to automatically provide the reasoning system with this service. No interest in efficiency is pretended: the intent of this mechanism is strictly for its facility since it allows the automatic return to a specific reasoning system state without programming overhead. In addition, it maintains temporal event frames which will be useful for future projects.

STATUS

The system described here is currently in the preliminary stages of design and implementation. The deductive retrieval mechanism is in place along with the model building search mechanism. Experiments have been performed on a simple planning problem which does not require analogy, in order to evaluate the model building strategy. Next, a simple diagnosis problem will be used to develop the point of view mechanism. Finally, a simple analogical problem will be constructed to provide a well understood test case. Each of these examples will use the same knowledge base which will be augmented as the implementation proceeds.

RESULTS

Although it is far too early to draw any conclusions, the chosen hierarchical model structure has uncovered some interesting issues regarding the distribution of functional information within the models. The philosophy described earlier departs from some previous work in causal reasoning in that there seems to be little penalty for mixing general laws about object behavior along with specific functionality since the two are separated within the model. It seems acceptable to even describe specific instance data as the expected behavior at one level with the assurance that it can be rescinded at another.

MODAL PROPOSITIONAL SEMANTICS FOR REASON MAINTENANCE SYSTEMS

Allen L. Brown, Jr.

ABSTRACT

Non-monotonic logics are examined and found to be inadequate as descriptions of reason maintenance systems (sometimes called truth maintenance systems). A logic is proposed that directly addresses the problem of characterizing the mental states of a reasoning agent attempting to reason with respect to some object theory. The proposed logic, propositional dynamic logic of derivation (PDLD), is given a semantics, and a sound and complete axiomatization. The descriptive power of PDLD is demonstrated by expressing various inferential control policies as PDLD formulae.

INTRODUCTION

In this note we will elaborate the propositional fragment of an axiomatic semantics of reason maintenance systems (RMS's) [3]. The development of such a semantics stems from the desire to provide a declarative specification language for RMS's with particular emphasis on the description of the control of their reasoning processes, and to serve as a formal setting within which to compare and contrast the properties of different RMS's.

There is considerable ongoing research activity in the realm of non-monotonic reasoning [14]. The avowed aim of this research is to capture in a logical formalism some of the non-monotonic processes (e.g., default reasoning and defeasible reasoning) that are clearly part of the common sense reasoning repertoire enjoyed by humans. Implicit or explicit in many of these formalisms is the notion that the formalism in some sense describes the process carried out by the reasoning agent. In [11] McDermott and Doyle analyze Doyle's TMS [4] in terms of the non-monotonic logic that they elaborate in [11]. Their analysis suggests that the logic of TMS is a fragment of their non-monotonic logic. I believe that their analysis confuses the logic practiced by the reasoning agent (the TMS) with the particular object theory that the agent reasons about. A reasoning agent should be viewed as a finitary computing entity. The computations that it carries out have the express aim of mechanizing some object theory. Depending on the nature of the object theory or the reasoning agent's grasp of the theory, the mechanization may turn out to be imperfect. With respect to logics like that of [10] and [16], because there cannot be, in general, a recursive enumeration of the theorems of the object theory, a reasoning agent's mechanization of such theories is bound to be imperfect. In summary, the relation that obtains between an object theory and a reasoning agent is that the theory is an ideal object that the agent might hope to compute.

The sense in which many of the non-monotonic logics that have been studied might be descriptions of RMS's, or reasoning agents more generally, is roughly the sense in which a formalization of recursive function theory might be the description of a programming language, say PASCAL. Recursive function theory can be taken as an ideal object that a PASCAL implementation attempts to mechanize. However, recursive function theory has little to say about the actual semantics of PASCAL programs. Inevitably, a formal semantics of PASCAL would include recursive function theory, but most of the meat in axiomatizing PASCAL is the formalization of the states of the abstract machine that is interpreting PASCAL.

There are some researchers who have attempted to address the issue of describing the reasoning agent and its mental states. Weyhrauch's FOL system [18] has an explicit notion of object theory and meta-theory. (Indeed, FOL permits the construction of arbitrary hierarchies of such object/meta pairs.) FOL is an axiomatic system, specifically, a first-order system with types. From my perspective, FOL's main defect is that a FOL meta-theory, if taken as an attempt to formalize the properties of reasoning agents, has no explicit notion of the agent's mental state. We believe that an explicit notion of mental state is key to many representations and control issues.

Doyle [3] develops a very powerful functional semantics for theories of reasoned assumptions. His semantics, in the guise of an admissible set, has a definite notion of the mental state of a reasoning agent. He elaborates his functional semantics so as to be able give taxonomic structure to a wide range of reasoning formalisms. He focuses primarily on giving an account of what inferential theories are sanctioned by different formal notions of reasoned assumptions. Our interest, in contrast, is in describing the behavior of a reasoning agent when constrained to adhere to particular object theories. We should also mention that we prefer axiomatic to functional specifications as we think there is much more available technology for compiling operational RMS's from axiomatic descriptions.

Goodwin recently introduced [5] a new inferential formalism, logics of current proof (LCP's). His intent is to capture the dynamic reasoning processes of finite reasoning agents. LCP's are not logics in the usual sense as they have no proof theory or model theory. Goodwin's formal account of LCP's is functional in nature. The principal appeal of LCP's is that they explicitly encode the development of the deductive process. It was in attempting to give a first-order logic account of LCP's, having models that suitably interpreted the sequence of databases in an LCP that we happened upon the idea of a dynamic logic of derivation.

The proximal technical inspiration of the dynamic logic of derivation (DLD) is the dynamic logic (DL) formalism introduced by Pratt and elaborated by Fischer, Harel, Ladner, Meyer, and others [6,7]. DL gives axiomatic meaning to programs by means of a first-order language augmented with a collection of modal operators corresponding to those programs. Formulae in the language are used to characterize the states of computational processes before and after the execution of some computational step(s). DL's model theory is a collection of Kripke-style worlds [8] connected by binary relations corresponding to various possible programs. Just as the worlds of DL's semantics capture the states of a computational machine, the states of a DLD model will capture the mental states of a rational agent. The approach that we shall be taking is presaged by Pratt in [15] where he uses variants of DL to formalize individual actions, sequences of actions (processes), and their effects. The remainder of this paper is devoted to elucidating propositional dynamic logic of derivation (PDLD).

SYNTAX

Let L be a first-order language equipped with functions, predicates, connectives, quantifiers, and perhaps even modalities. L has the usual formation rules for first-order languages. The details of L will not concern me very much here. Let T be a theory over the language L. T is assumed to be axiomatizable with a set of axioms and rules of inference. $'L$, the language of PDLD, can to some extent be considered a meta-language for tor theories over L. Formulae over $'L$ will typically be used to specify how the formulae of T are actually derived from T's axioms and rules of inference. This specification will be in the form of an axiomatized theory $'T$. We will call $'T$ the mechanization of T. In effect $'T$, when so elaborated, will (partially) specify a reason maintenance system for the theory T.[1]

$'L$ has two sets of symbols: the atomic formulae and the atomic derivations, collectively denoted as I_0 and j_0, respectively. The atomic formulae are further subdivided into two classes, the proper atomic formulae and the reified atomic formulae. $'\Phi$ is a reified atomic formula of $'L$ if, and only if, Φ is a formula of L. We will use (possibly subscripted) ϕ, ψ, and χ to denote formula variables of L, Φ, Ψ, and χ to denote instances of formulae of L; p, q and r to denote formula variables of $'L$; P, Q, and R to denote instances of atomic formulae of $'L$; α and β to denote derivation variables; and a and b to denote instances of named atomic derivations. There is also the anonymous atomic derivation, $\vdash$. The proper atomic formulae are meant to behave like the truth value bearing constants of ordinary propositional logic. Intuitively reified atomic formulae are formulae that are asserted as deduced after some instance of a rule of inference in T has been applied.[2] Similarly atomic derivations are specific instances of inference rules. The PDLD-wffs and PDLD-derivations are defined by simultaneous induction:

1. an atomic formula is a PDLD-wff,
2. an atomic derivation is a PDLD-derivation,
3. for any PDLD-derivations α and β $(\alpha;\beta)$, $(\alpha\bigcup\beta)$, α^*, and α^{-1} are PDLD-derivations,
4. for any PDLD-wffs p and q and PDLD-derivation α, $\neg p$, $p\vee q$, and $<\alpha>p$ are PDLD-wffs.

We will abbreviate $\neg(\neg p\vee\neg q)$ to $p\wedge q$; $\neg p\vee q$ to $p\rightarrow q$; $(p\rightarrow q)\wedge(q\rightarrow p)$ to $p\equiv q$; $<\alpha;\alpha^{n-1}>p$ $(n>0)$ to $<\alpha^n>p$; $\neg<\alpha>\neg p$ to $[\alpha]p$; and $<\alpha^0>p$ to p.

SEMANTICS

Let W be a non-empty universe of states, elements of which are denoted by s and t (possibly with subscripts). A PDLD interpretation determines whether or not an PDLD-wff P is true in a state s (or s satisfies P). Atomic derivations can be viewed as binary relations on W. Accordingly an interpretation is defined to be a triple $<W,\pi,m>$,[3] where W is a non-empty set, π: $I_0\rightarrow 2^W$ and m: $j_0\rightarrow 2^{W\times W}$. π and m provide meaning for atomic formulae and derivations, and are extended inductively to the rest of $'L$:

$$m(\alpha;\beta) = \{<s,t>|\exists u <s,u>\in(a)\wedge<u,t>\in m(\beta)\}$$

$$m(\alpha\cup\beta) = m(\alpha)\cup m(\beta),$$

$$m(\alpha^*) = (m(\alpha)),$$

$$m(\alpha^{-1}) = \{<s,t>|<t,s>\in m(\alpha)\},$$

$$m(\vdash) \supseteq \bigcup_{\alpha\in j0} m(\alpha)$$

$$\pi(P\vee Q) = \pi(P)\cup\pi(Q),$$

$$\pi(\neg P) = W-\pi(P),$$

$$\pi(<a>P) = \{s|\exists t <s,t>\in m(a)\wedge t\in\pi(P)\}$$

$$\pi(<\alpha;\beta>P) = \{s|\exists t<s,t>\in m(\alpha;\beta)\wedge t\in\pi(P)\}$$

$$\pi(<\alpha\cup\beta>P) = \{s|\exists t<s,t>\in m(\alpha\cup\beta)\wedge t\in\pi(P)\}$$

$$\pi(<\alpha^*>) = \{s|\exists t<s,t>\in m(\alpha^*)\wedge t\in\pi(P)\}$$

$$\pi(<\alpha^{-1}>) = \{s|\exists t<s,t>\in m(\alpha^{-1})\wedge t\in\pi(P)\}.$$

Denoting $s\in\pi(\Phi)$ by $s\models\Phi$ and $<s,t>\in m(a)$ by *sat* and adopting free usage of conventional logical symbols, one may write for a fixed interpretation $<W,\pi,m>$ that $s\models<a>\Phi$ if and only if there is a t such that *sat* and $t\models\Phi$. Given an interpretation $I=<W,\pi,m>$, a PDLD-wff P is I-valid (written $\models_I P$) if for every $s\in Ws\models P$. A PDLD-wff P will be said to be PDLD-valid (written $\models P$) if for every I, it is I-valid. P will be said to be I-satisfiable if there is an s such that $s\models P$ and satisfiable if there is an I such that $\models_I P$.

[1] We wish to distinguish PDLD (and the first-order dynamic logic of derivation) from the dynamic logics of programs investigated by Pratt et al. The distinction is not grounded so much in their respective model theories or proof theories, but rather in the fact that the model-theoretic worlds of the former are related by program statements while in the latter they are related by inferential steps.

[2] The distinction between proper and reified atomic formulae will play no role in the development of PDLD proper. The distinction becomes important when the axioms that describe particular RMS's are adjoined to the axiomatization of PDLD.

[3] We will identify the three constituents of an interpretation with a particular interpretation I by suing the notation W^I, π^I, m^I.

A COMPLETE AXIOMATIZATION OF PDLD

The system **P** will constitute an axiomatization of PDLP. The axioms for **P** are the tautologies of propositional calculus together with:

$$[\alpha](p \to q) \to ([\alpha]p \to [\alpha]q) \tag{1}$$

$$[\alpha \bigcup \beta]p \equiv ([\alpha]p \wedge [\beta]p) \tag{2}$$

$$[\alpha;\beta]p \equiv [\alpha][\beta]p \tag{3}$$

$$[\alpha^*]p \to [\alpha]p \tag{4}$$

$$[\alpha^*]p \to p \tag{5}$$

$$[\alpha^*]p \to [\alpha^*][\alpha^*]p \tag{6}$$

$$p \to [\alpha]<\alpha^{-1}>p \tag{7}$$

$$p \to [\alpha^{-1}]<\alpha>p \tag{8}$$

$$(p \wedge [\alpha^*](p \to [\alpha]p)) \to [\alpha^*]p \tag{9}$$

$$[\vdash^z]p \to [\alpha^z]p \tag{10}$$

where z is an integer or '*' and $\alpha \in I_0$

The rules of inference for **P** are:

$$\text{if } \vdash_P p \to q \text{ and } \vdash_P p \text{ then } \vdash_P q \tag{11}$$

$$\text{if } \vdash_P p \text{ then } \vdash_P [\alpha]p \tag{12}$$

The following two theorems are straightforward consequences of the syntax, semantics, and axiomatization above:

Theorem 4.1 *The axioms (1) through (10) are PDLD-valid.*

Theorem 4.2 *The rules of inference (11) and (12) are sound with respect to PDLD interpretations.*

Parikh's completeness proof for propositional dynamic logic of programs [13] can be adapted to PDLD to obtain:

Theorem 4.3 *Every* PDLD-*valid formula is in the deductive closure of the system* **P**.[4]

DESCRIPTIVE POWER

General Considerations on Monotonic Theories

Thus far we have done nothing that connects any particular object theory T with a mechanization $'T$. In order to make that connection and to exhibit the descriptive power of $'L$, we will augment **P** with proper axioms that characterize a monotonic theory T. Assume that T includes the first-order predicate calculus. For each axiom Φ of T, there is an axiom $'\Phi$ of $'T$. Consider an instance of modus ponens in T:

$$\text{if } \vdash_T \Phi \text{ and } \vdash_T \Phi \to \Psi \text{ then } \vdash_T \Psi \tag{13}$$

This suggests an axiom for $'T$ of the form:

$$'\Phi \wedge '(\Phi \to \Psi) \to <MP>'\Psi^5 \tag{14}$$

The second observation to be made about modus ponens is that it is "belief conserving." That is, anything that is believed before the application of modus ponens should continue to be believed afterward. Conservation of belief (and non-belief) is a property inherent in monotonic rules of inference. To generalize then from the case of modus ponens, for each inference instance (of T) represented by the atomic derivation a, with antecedents $\Phi_1,\ldots,\Phi_n$ and consequent Ψ there is an axiom of $'T$ of the form

$$'\Phi_1 \wedge \cdots \wedge '\Phi_n \to <a>'\Psi \tag{15}$$

Given that T is monotonic, it seems natural to require the following frame axiom schema to enforce belief conservation relative to each atomic derivation: a

$$'\phi \to [a]'\phi \text{ where } \phi \text{ is any } L-wff \tag{16}$$

$$\neg'\phi \to [a]\neg'\phi \text{ where} \phi \text{ is any } L-wff \neq \Psi \tag{17}$$

It can be shown that $< \vdash^* >'\Phi$ can be proved from $'T$ (keeping in mind that $'T$ mechanizes the first-order predicate calculus), an augmentation of **P**, whenever Φ is a theorem of T. Indeed, **P** augmented with axioms corresponding to an object theory T together with derivation and frame axioms as above will be termed the natural mechanization of T. This leads to asserting that a PDLD theory $'T$ completely mechanizes T just in case

$$\vdash_T \Phi \text{ if and only if } \vdash_{'T} <\vdash^*>(\Phi \wedge [\vdash^*]'\Phi.^6 \tag{18}$$

Needless to say, if the object theory T to be mechanized happened to be the pure first-order predicate calculus, formulae such as $\neg <\vdash^*>'\Phi$ cannot generally be proven in the natural mechanizing theory $'T$ [1]. This observation has important consequences *vis a' vis* the proof theory of non-monotonic theories [10] and their mechanizations (see below).

Notice that for an object theory T and mechanizing theory $'T$, We have been implicitly taking $<\vdash^*>'\Phi$ to mean that $'T$ "believes" $'\Phi$ to be a consequence of believing the object theory T. Suppose **P** were taken as the object theory of $'T$.[7] $'T$ can be constructed in such a way that Φ is a theorem of $'T$ if, and only if, $<a_1;\ldots;a_n>'\Phi$ is a theorem of $'T$ for some sequence $a_1,\ldots,a_n$ of (reified) atomic derivations. On the other hand, it can also be demonstrated for $'T$ that Φ is not a theorem of $'T$ if, and only if, $<a_1;\ldots;a_n>'\Phi$ is *not* a theorem of $'T$ for any sequence $a_1,\ldots,a_n$ of atomic derivations. In fact, Φ is not a theorem of $'T$ if, and only if, $\neg'\Phi \to <a_1;\ldots;a_n> \neg'\Phi$ is a theorem of $'T$ for every sequence $a_1,\ldots,a_n$ of reified atomic derivations. The situation that appears to obtain in $'T$ then is the PDLD analogue of what

[4] The principal technical hurdles in adapting Parikh's complete proof to **P** are in validating certain claims that Parikh makes for "pseudo-models" and "closed sets" when applied to **P**.

[5] For a complete characterization of a monotonic inference rule such as *modus ponens*, one should also add the axiom $'\Phi \wedge '(\Phi \to \Psi) \to [MP]'\Psi$ since the rule is entirely deterministic in its consequent.

[6] The assertion $<\vdash^*>'\Phi$ does not suffice on the right hand side of the "if, and only if" as T might be a non-monotonic theory. The second clause is necessary in order to assure that once $'T$ derives $'\Phi$ it "sticks" and that $'T$ does not oscillate, believing and disbelieving $'\Phi$, owing to some belief revision policy.

[7] To show how such a thing is possible, let $'L$ be the language L together with the formula $'\Phi$ whenever Φ is a formula of $'L$. Consider $'$**P**, the system **P** taken over $'L$ together with an axiom $'\Phi$ whenever Φ is an axiom of **P**, the natural axiomatic encodings of the rules of inference (modus ponens and necessitation) of **P**, and the frame axioms for those rules of inference. In the same spirit as reified atomic formulae, atomic derivations that are instances of modus ponens and necessitation of the system **P** will be called reified.

Moore [12] calls autoepistemic stability of an ideally rational agent. Loosely speaking, Φ is a theorem of $'T$ if, and only if, from every mental state (wherein Φ may or may not believed) there is a derivation leading to a mental state in which Φ is believed. Conversely, Φ is not a theorem of $'T$ if, and only if, (dis-)belief in Φ is invariant under derivation.

Specifying Breadth-first Search

An explicit derivation of Φ is a formula of the form $<a_1,\ldots,a_k>'\Phi$. $'T$ enumerates the theorems of T in a breadth-first fashion if and only if

1. for each theorem Φ of T, there is an explicit derivation of Φ that is a theorem of $'T$,
2. the sequence of named atomic derivations that appears in the prefix of $'\Phi$ corresponds to the sequence of inference rules applied in the proofs of the theorems of T when enumerating them in breadth-first order,
3. if Ψ_1 precedes Ψ_2 in the breadth-first ordering, then the derivation of Ψ_2 cannot be proved as a theorem of $'T$ until Ψ_1 has been proved.

A formula T is said to be of rank n if the shortest proof of that formula is of length n. Then axioms of T are of rank 0. Let Δ_n be an ordered list of the last atomic derivations applied in the proofs of each of the formulae of rank n.[8] Breadth-first enumeration is achieved by replacing axiom (15) above with (19,20) below:

$$C_{1,1} \tag{19}$$

$$C_{n,m} \wedge '\Phi_{n,m,1} \wedge \cdots \wedge '\Phi_{n,m,k} \rightarrow <a_{n,m}>'\Psi_{n,m} \wedge D_{n,m} \tag{20}$$

and adding boundary conditions

$$D_{n,m} \rightarrow [a_{n,m}]C_{n,m+1} \text{ if there exists } a_{n,m+1} \tag{21}$$

$$D_{n,m} \rightarrow [a_{n,m}]C_{n+1,1} \text{ otherwise} \tag{22}$$

where the $a_{n,m}$ is the m'th atomic derivation on the list Δ_n, and the $\Phi_{n,m}$'s and $\Psi_{n,m}$ are, respectively, the antecedents and consequent of the atomic derivation $a_{n,m}$. The interaction of the C's and D's prevents $\Psi_{n,m+1}$ from being derived before $\Psi_{n,m}$ is derived. Indeed, no formula of rank n is derived before every formula of lesser rank is derived. The Ψ's are thereby forced to be produced in breadth-first order. Of course it must be verified that a theory $'T$ that mechanizes T completely, when modified with the breadth-first axioms, continues to mechanize T completely. To that end the following holds:

Theorem 5.1 *If $'T$ is the natural mechanization of T with axiom (15), and if $'T'$ is the breadth-first mechanization of T with axioms (19,20,21,22) replacing (15), and if $\vdash_{T} < \vdash^* > '\Phi$, then $\vdash_{T}' < \vdash^* > '\Phi$.*

With a different set of boundary conditions, a depth-first enumeration of the theorems of T could have been achieved. That is, there is a set of boundary conditions such that

1. for each theorem Φ of T there is an explicit derivation of Φ that is a theorem of $'T$,
2. the sequence of named atomic derivations that appears in the prefix of $'\Phi$ corresponds to the sequence of inference rules applied in the proofs of the theorems of T when enumerating them in depth-first order,
3. if Ψ_1 precedes Ψ_2 in the depth-first ordering, then the derivation of Ψ_2 cannot be proved as a theorem of $'T$ until Ψ_1 has been proved.

The interaction between the axioms (19,20) and boundary conditions suggests a general "programming" methodology for controlling the application of derivations. The propositional constants $D_{n,m}$ and $C_{n,m}$ should be viewed as "enabling" and "completion" flags for the firing of the atomic derivation $a_{n,m}$. These constants indicate respectively that a derivation can be used and that a derivation has been used. Programming then consists of designing systems of boundary conditions to achieve the desired sequencing of inferences by suitably controlling the truth values of enabling flags in various mental states.

Goodwin [5] (and McDermott before him in [10]) cites a number of problems in using deduction to control deduction. He remarks that attempts at controlling inferences by deductive methods have typically resulted in invalidating particular inferences altogether, or alternatively resulted in RMS states that assert that some proposition has been proven if and only if it has not been proven. It should be clear from the discussion of programming above that atomic derivations are enabled with respect to particular states. As a consequence, an inference can be temporarily en-(dis)-abled, and there is no problem whatsoever in having some proposition $'\Psi$ be derived by some derivation that has since become disabled. The axiom schemata (21,22) could just as well have been written

$$D_{n,m} \rightarrow [a_{n,m}]C_{n,m+1} \wedge \neg D_{n,m} \text{ if there exists } a_{n,m+1} \tag{23}$$

$$D_{n,m} \rightarrow [a_{n,m}]C_{n+1,1} \wedge \neg D_{n,m} \text{ otherwise} \tag{24}$$

which have the effect of disabling each of the (19,20) after use.

Finite Reasoning Agents

At the outset of this note we proclaimed PDLD as a mechanism for describing the behavior of finite reasoning agents. Careful scrutiny of PDLD interpretations will reveal that PDLD theories admit interpretations which are at odds with any reasonable notion of a finite agent. Consider the following observations. For one thing, of a sequence of mental states related by various atomic derivations and corresponding to the flow of some form of mental time, that time can extend infinitely into the past and future. Moreover, a mental state can be immediately preceded by multiple states. Finally, states can be "dense." That is, PDLD interpretations can be such that for an atomic derivation a whenever $<s,t> \in m(a)$ there is a u such that $<s,u> \in m(a)$ and $<u,t> \in m(a)$.

[8] It could be that the formulae of rank n are infinite in number. In that case the enumeration will never get beyond the formulae of rank n.

As it turns out, all of these anomalies can be legislated away with appropriate axioms. Tense logics [17] that impose various topologies on the ordering of time provide much of what is needed. To focus on one of the anomalies, consider the infinite extension into the past. This can be eliminated with:

$$<\vdash^{*}>\neg<\vdash^{-1}>p\vee\neg p. \tag{25}$$

This last formula says that every state either is, or is preceded by, a state which is *not* immediately preceded by a state that satisfies $p\vee\neg p$. But since every state satisfies $p\vee\neg p$, this formula can be satisfied if, and only if, every state is either immediately preceded by no state at all, or is preceded by some state which is in turn preceded by no state. This axiom prevents infinitely long (receding) chains of states. On the other hand, it does not prevent interpretations having a particular state from which there is a receding chain of any given finite length. More axiomatic machinery still is required to prevent that.

Non-monotonic Theories

In considering the descriptive power of PDLD with respect to non-monotonic theories it should first be noted that the intuitive statement of the rule of possibilitation introduced in [11] is directly expressible in PDLD. Recall that McDermott and Doyle first gave an informal definition of their non-monotonic rule of inference which stated that if a proposition were not provable in a theory T, then the negation of the proposition is provably possible. Though the intent of this rule is clear, it is unfortunately circular. McDermott and Doyle had to appeal to an indirect technical device to capture possibilitation. In the PDLD mechanization of T, however, their original notion of possibilitation can be expressed as:

$$\neg<\vdash^{*}>'\neg\phi\rightarrow<\vdash>'<>\phi \tag{26}$$

where "$<>$" is the consistency modality of [10,11]. Possibilitation is well defined but, unfortunately, not effectively computable in general. Since there is no magic, a non-monotonic theory T that is not recursively enumerable, cannot have a complete mechanization that is recursively enumerable. If a (partial) mechanization $'T$ is to remain r.e., such mechanizations cannot in general have the formulae $\neg<\vdash^{*}>'\neg\phi$ (on the antecedent side of 26) as theorems.

The whole point of a non-monotonic logic is to formalize the default and defeasible inferences that are evident in common sense reasoning and practiced by various RMS's. It should be evident that PDLD provides a mechanism for directly formalizing such reasoning without necessarily resorting to the sorts of infinitary processes implicit in McDermott and Doyle's rule of possibilitation. In order to realize defeasible inferences, a PDLD theory cannot have the general frame axioms (16,17); not all atomic derivations will be belief conserving. A default introducing axiom scheme might be:

$$\neg'\neg\phi\rightarrow<\vdash>'\phi \tag{27}$$

which says that if $\neg\phi$ is not currently believed then ϕ can be believed. Of course, it might be the case that $\vdash_{T}<\vdash^{*}>'\neg\phi$. Thus, the simple notion of default reasoning supported by (27) would admit states to interpretations of $'T$ that sanctioned inconsistent beliefs. Now for $'T$ to have inconsistent beliefs is not the same as $'T$'s being inconsistent. On the other hand, states that have $'\phi\wedge'\neg\phi$ true are irrational, and to have $<\vdash^{*}>'\phi\wedge'\neg\phi$ as a theorem of $'T$ makes $'T$ irrational. RMS's generally have backtracking mechanisms to revise the set of current beliefs so that consistency of beliefs is restored. Although PDLD as presented here is not expressive enough to describe all the details of those mechanisms, it can describe the general policies that are typically enforced by those mechanisms. A weak policy might be:

$$('\phi\wedge'\neg\phi)\rightarrow[\vdash]\neg('\phi\wedge'\neg\phi) \tag{28}$$

which says that if the reasoning agent is in a state that is irrational with respect to a particular formula ϕ, all states immediately reachable from that state should be rationalized. A much stronger (and typically unenforceable by effective computation) policy is stated by:

$$('\phi\wedge'\neg\phi)\rightarrow[\vdash]\neg<\vdash^{*}>('\phi\wedge'\neg\phi) \tag{29}$$

This schema says that if the reasoning agent is in a state that is irrational with respect to a particular formula ϕ, the agent should do something (e.g., withdraw sufficient premises or hypotheses in which the irrational state is grounded) such that at no future time can the agent be in a state irrational with respect to ϕ. These examples of deduction and premise control policies seem to respond directly to McAllester's [9] objections to non-standard logics:

> The problem with non-monotonic logics is that they bring in non-traditional formalisms too early, muddying deduction, justifications, and backtracking. The aspect of truth maintenance which cannot be formalized in a traditional framework is premise control...

Dynamic logics of derivation offer an opportunity to make the various issues explicit.

CONCLUSIONS

In the foregoing we have developed the syntax and semantics of the propositional dynamic logic of derivation, and presented a complete axiomatization for the logic. By way of examples we have illustrated some of the expressive power available in PDLD for specifying and analyzing the behavior of reason maintenance systems. Finally we have offered dynamic logic as an alternative to the sorts of non-monotonic logics investigated heretofore as a means for giving a formal account of some aspects of common sense reasoning.

PDLD obviously cannot be completely expressive of all properties that might be ascribed to an RMS. For that, one requires the first-order dynamic logic of derivation [2]. In the latter formalism one can not only give a complete first-order account of control protocols, but also of the collateral data structures (viz. "no-good" lists, hypothesis contexts, dependency relations, etc.) that RMS's utilize in the belief revision process. Between PDLD without the $\vdash^{*}$ derivation and full first-order dynamic logic of derivation there are many alternative logics having different powers of expressiveness. The analogous dynamic logics of programs have been extensively investigated. We believe that those investigations will offer a good starting point for developing an RMS specification logic which is suitably expressive, while being deductively tractable.

REFERENCES

[1] Boolos, G., *The Unprovability of Consistency: An Essay in Modal Logic.* Cambridge: Cambridge University Press, 1979.

[2] Brown, A.L. *Considerations on the Semantics of Reason Maintenance Systems: A Modal Propositional Theory,* General Electric Research and Development Center technical report. Schenectady, New York. (forthcoming)

[3] Doyle, J., "A truth maintenance system," *Artificial Intelligence* 12:(1979)231-72.

[4] Doyle, J., *Some Theories of Reasoned Assumptions*, Carnegie Mellon University Computer Science Department technical report no. CMU CS-83-125. Pittsburgh, 1983.

[5] Goodwin, J.W., "WATSON: A dependency directed inference system," *Proceedings of the AAAI workshop on non-monotonic reasoning*, ed. R. Reiter et al., pp. 103-14, 1984.

[6] Harel, D., "Dynamic logic," *Extensions of Classical Logic,* eds. D. Gabbay and F. Guenthner, pp. 497-604. D. Reidel Publishing Company, Dordrecht, Netherlands: 1984.

[7] Harel, D., *First-order dynamic logic,* Lecture Notes in Computer Science, vol. 68. Berlin: Springer-Verlag, 1979.

[8] Hughes, G.E. and M.J. Cresswell, *An Introduction to Modal Logic,* Methuen, London: 1968.

[9] McAllester, D.A., *An Outlook on Truth Maintenance,* MIT Artificial Intelligence Laboratory memorandum no. 551. Cambridge, Massachusetts, 1980.

[10] McDermott, D.V., "Non-monotonic logic II: non-monotonic modal theories," *Journal of the Association for Computing Machinery* 29:(1982)33-57.

[11] McDermott, D.V., and J. Doyle, "Non-monotonic logic I," *Artificial Intelligence* 13:(1980)133-70.

[12] Moore, R.C., Semantical considerations on nonmonotonic logic, *Artificial Intelligence* 25, :75-94 January 1985.

[13] Parikh, R., *A Completeness Result for a Propositional Dynamic Logic.* MIT Laboratory for Computer Science technical memorandum no. 106. Cambridge, Massachusetts, 1978.

[14] Perlis, D., Bibliography of literature of non-monotonic reasoning, *Proceedings of the AAAI workshop on non-monotonic reasoning*, ed. R. Reiter et al., pp. 396-401, 1984.

[15] Pratt, V.R., *Six Lectures on Dynamic Logic*, MIT Laboratory for Computer Science technical memorandum no. 117. Cambridge, Massachusetts, 1978.

[16] Reiter, R., "A logic for default reasoning," *Artificial Intelligence,* 13:(1980)81-132.

[17] Rescher, N. and A. Urquhart, *Temporal Logic. Library of Exact Philosophy*, Springer-Verlag, New York, 1971.

[18] Weyhrauch, R.W., "Prolegomena to a theory of mechanized formal reasoning," *Artificial Intelligence* 13:(1980)133-70.

REASON MAINTENANCE
FROM A LATTICE-THEORETIC POINT OF VIEW

Dan Benanav, Allen L. Brown, Jr., and Dale E. Gaucas

ABSTRACT

Goodwin and de Kleer have each investigated certain fundamental aspects of reason (or truth) maintenance systems (RMS's), non-monotonic justifications in the case of the former and assumption-based justifications in the case of the latter. To a certain extent, each of their mechanisms can simulate the other, though not altogether satisfactorily. By recasting the reason maintenance problem in a lattice-theoretic framework we are able to develop a body of mathematical theory that elucidates reason maintenance in a general way so as to include both assumption-based and non-monotonic justifications in a direct and transparent fashion. More generally, if a method of labelling propositions so as to justify them according to some reasoning agent's constraints of belief also happens to conform to the postulates of Boolean lattices, the labelling system can be accommodated under the same umbrella of abstraction. The mathematics immediately suggests a collection of algorithms that support efficient revision of beliefs as a reasoning agent changes its assumptions and/or its constraints on beliefs.

INTRODUCTION

We propose here a single theoretical framework which subsumes various notions of reason maintenance, including the assumption-based justifications reported by de Kleer [9,8,10,11] and the non-monotonic justifications reported by oodwin [16,-7,18,19]. In this note we will give an abreviated account of a body of work that is fully reported in [4]. Our aim here is to motivate the work, present some of the mathematical theory, and interpret the theory in relation to other reason maintenance systems and in terms of algorithmic realizations.

We have a conservative view of the scope of reason maintenance systems. A similar view is implicitly evidenced in de Kleer's work and explicitly articulated by Goodwin: A reason maintenance system is a utility that supports deductive problem solving. It maintains a database of facts, some of which a client reasoning system holds as currently believed, others not.[1] It also supports relations over the facts that serve to record the arguments that sanction a reasoning agent's belief therein. Because a reason maintenance system must be founded on low-level facilities for retaining and matching data structures representing facts, it may also be convenient for the reason maintenance system to export interfaces for detecting database inconsistency and other "interrupts" triggered by the occurrence of various patterns in the database. Indeed, we see facts or propositions as exhibiting a number of salient characteristics for a problem solver: *true, provable,* and *proven*. The problem solving mechanisms for attributing those characteristics are observation, deduction, and reason maintenance.

Because of our views on how a problem solving system should be structured, there are some functions that we believe the reason maintenance system should *not* fulfill. It should not be a mechanism for managing the restoration of consistency when a reasoning agent discovers itself to be in an inconsistent state. It should neither determine what constitutes a valid deduction nor manage the sequencing of inferences. The reason maintenance system may provide support for all of the foregoing, but is not the most appropriate place to marshall such efforts.

The initial motivation for this work was the desire to unify in a single mechanism the reason maintenance paradigms of de Kleer and Goodwin. The systems of both investigators can be viewed as constraint propagation mechanisms. Given disjunctive sets of sets of premises and a set of (monotonic) deductive constraints, de Kleer's ATMS tells a client problem solving system what things it is currently obliged to believe assuming one or another of the sets of premises. Goodwin's LPT, on the other hand, tells the client problem solving system what things it is currently obliged to believe given a single set of premises under deductive constraints, some of which may be non-monotonic in nature.[2] Our original intuition was that it should be possible to account *simultaneously* for multiple sets of premises *and* non-monotonic deductive constraints.

This intuition arose from the striking similarity that we observed in the computations of reason maintenance systems and the computations of global flow analysis that underly modern optimizing compilers [2,20,21,23]. Global flow analysis can be couched in the following terms: Given the constraints imposed by individual program statements and their interconnecting topology, what facts is a reasoning agent (in this case concerned with programs) obliged to

[1] Note that failure to believe a fact is not identical to believing its negation.

[2] A monotonic deductive constraint obliges a rational agent to believe its consequent given that it currently believes all of its antecedents. A *non-monotonic* deductive constraint obliges a rational agent to believe its consequent given that it believes all of its monotonic antecedents and none of its non-monotonic antecedents.

believe about the state of computation at various points in the program's control flow? In a sense the information propagation problem solved by global flow analysis can be viewed as the dual of the reason maintenance problem. The former assigns propositions to contexts established by various paths through a program. The latter assigns contexts of belief to propositions under various deductive constraints. There are two principal methods of solving information propagation problems. Both hinge on solving systems of equations whose unknowns range over the domain of an algebraic lattice. The work that we will describe presently retains the idea of equations over a lattice, but for various technical reasons (principally non-monotonic constraints) the solution methods used in global flow analysis turn out to be inappropriate. A rather different solution method has been developed.

REASON MAINTENANCE IN A LATTICE-THEORETIC FRAMEWORK

We begin by introducing the idea of a *Boolean lattice*. A complete account of such structures can be found in any of [3,5,22]. For our purposes here, the elements of such a lattice are meant to capture the idea of alternative *situations*. With respect to any particular situation a finite reasoning agent takes certain formulae as premises.

Definition 2.1 Let β be a Boolean lattice equipped with the usual meet, join, and complementation operators; a partial order, $<$; and maximum and minimum elements, $\top$ and $\perp$ respectively. Elements of β will be called *situations*, and will be denoted by A and B. A and B (possibly subscripted) are *lattice expressions* in β. Moreover, if A and B are expressions in β then so are $A \vee B$, $A \wedge B$, $\bar{A}$ and $\bar{B}$.

Especially important to us will be the existence of the partial order, the complement, maximum and minimum elements, and the mutual distributivity of meet and join.

A *lattice unknown* is a super- and/or subscripted s or t. Each lattice expression in β and unknown is a *lattice form* in β. Moreover, if X and Y are forms in β then so are $X \vee Y$, $X \wedge Y$, $\bar{X}$ and $\bar{Y}$. Individual (fixed) lattice forms in β will be denoted by X and Y, possibly subscripted. Lattice unknowns correspond to what some investigators have called *nodes*. Every fact or proposition has an associated unknown. Note that a proposition and its negation have distinct associated unknowns.

Definition 2.2 A *lattice equation over* β is a relation of the form $X = Y$ where X is a lattice unknown and Y is a lattice form.

Definition 2.3 A *lattice equational system over* β, Σ, is any collection of lattice equations over β such that the total number of lattice unknowns occurring on the right-hand sides of the equations is finite and any lattice unknown occurs at most once on the left-hand side of an equation. The equation on whose left-hand side s appears will be called the s equation. If the right-hand side of the s equation is a lattice expression, s will be termed *trivial*.

Σ will be sub- or superscripted on those occasions when it is useful to distinguish among various equational systems. Unless there is some ambiguity in the context, we will freely say "system" without modifiers. A lattice equational system should be interpreted as encoding the way a reasoning agent's belief (or disbelief) in a collection of propositions entail belief in others.

Definition 2.4 If Σ is a lattice equational system such that the right-hand side of each equality is of the form $\vee_i \wedge_j X_{ij}$ where each X_{ij} is an element of β or an unknown (possibly complemented), then Σ is said to be in *disjunctive normal form*.

Disjunctive normal form, a consequence of distributivity in β, gives us a useful way of presenting lattice forms in general, and lattice equational systems in particular. Since we can transform any form to disjunctive normal form, we will usually treat forms over β and lattice equational systems as if they were in disjunctive normal form.[3]

Definition 2.5 A *solution* to a lattice equational system, Σ, is a function, Γ, from the lattice unknowns appearing in the system into β such that if for each equation in the system, each unknown s in the equation is replaced by $\Gamma(s)$ the equation holds in β. A lattice equational system having a solution will be termed *solvable*.

We will, in fact, take solutions as assigning values from β to every unknown, s, whether it is mentioned explicitly on the left-hand side of an equation or not. Put another way, unknowns, s, not having an associated equation, implicitly have the equation $s = \perp$. We will interpret lattice equations as constraints. A solution, then, is a labelling of propositions with situations. In particular, the situations are those in which a reasoning agent is obliged to believe the correspondingly labelled proposition given acceptance of the constraints imposed by the system.

Definition 2.6 Let X be a form in β and lattice unknowns of a system, Σ. If Γ is a solution of Σ, then $\Gamma(X)$ is the expression over β that results from substituting for each occurrence of each unknown, s, the value $\Gamma(s)$.

Definition 2.7 A *justification* of a disjunctive normal form lattice equational system, Σ, is an ordered pair $d = \langle s, X \rangle$, where s appears on the left-hand side of some equation in Σ and X is a disjunct on the right-hand side of that same equation. Also, s is called the *consequent* of the justification d and each conjunct of the disjunct X is called a *non-monotonic* or *monotonic antecedent* of d depending on whether or not it is complemented. The sets of monotonic and non-monotonic antecedents of d are respectively denoted $\alpha(d)$ and $\bar{\alpha}(d)$.

Definition 2.8 A justification, d, is *valid* with respect to a situation, A, and a solution, Γ, of an equational system Σ if and only if,

$$A \leq \bigwedge_{s \in \alpha(d)} \Gamma(s) \wedge \bigwedge_{s \in \bar{\alpha}(d)} \overline{\Gamma(s)}$$

[3] The assumption of disjunctive normal form is a convenience for mathematical analysis and not a requirement for the algorithms engendered by this analysis. This is in contrast to the ATMS's *requirement* of a disjunctive normal form representation.

We will write Valid(A, d, Γ) to indicate that d is valid with respect to A and solution Γ.

Definition 2.9 **A solution,** Γ, is *well-founded with respect to a lattice equational system,* Γ, *at lattice unknown,* s, if and only if $\Gamma(s) = \vee_i A_i$, and for each A_i, there is a partially ordered set, $\langle P_{A_i}, <_{A_i} \rangle$, such that P_{A_i} is a set of justifications from Σ and

1. there is a justification, d in P_{A_i}, whose consequent is s,
2. for every justification d, in P_{A_i}, Valid(A_i, d, Γ),
3. every unknown, s', that is a monotonic antecedent of some d in P_{A_i} is also the consequent of some justification d' in P_{A_i} and $d' <_{A_i} d$.

Definition 2.10 A solution to a lattice equational system is *well-founded* if and only if it is well-founded with respect to the system at every lattice unknown mentioned in the system.

We interpret justifications, validity and well-foundedness in the following way: Validity describes the circumstances under which the consequents of a justification are to be believed given the belief status of the antecedents. A justification therefore constitutes an independent source of support justifying belief in a consequent. Chaining justifications together constitutes a supporting argument. Since we wish for our arguments to be non-circular, we impose an additional condition, well-foundedness, to guarantee that state of affairs.

Using only the concepts we have introduced thus far, it can be demonstrated that finding solutions to systems is NP-hard in the number of equations. Doyle [12] and Goodwin [19] have questioned whether or not the well-foundedness condition might simplify the solution finding process. Unhappily, finding well-founded solutions is also NP-hard in the number of equations.

Definition 2.11 A *path* from s_0 to s_n is a sequence of triples of the form X_1, Y_1, s_1, X_2, Y_2, s_2, $\cdots$, X_n, Y_n, s_n, where X_i is an antecedent of the Y_i disjunct of the s_i equation in Σ. X_i is a complemented (uncomplemented) unknown if it is a complemented (uncomplemented) conjunct of Y_i with $X_i \in \{s_{i-1}, \bar{s}_{i-1}\}$ and $1 \leq i \leq n$. A path is *odd* if it has an odd number of complemented unknowns and *even* otherwise. A system is odd (and even otherwise) if it has an unknown, s, and an odd path from s to s.

Thus far we have established a framework within which we can formally describe reason maintenance problems. For this framework to be truly useful we must provide a way of finding solutions in a structured fashion. There is no obvious means of finding solutions of lattice equational systems because of the nature of the meet and join operators. Informally we may say that meet and join do not have "inverses" in the sense that subtraction and division are the respective inverses of addition and multiplication in an algebraic field. For the sake of brevity in the remainder of this section, we will focus on even lattice equational systems.[4]

Finding solutions depends on a pair of lattice equational system transforming operations that yield new systems whose well-founded solutions are well-founded solutions of the original system.

Definition 2.12 A *local substitution transformation under* s of a lattice equational system, Σ, *results in a new system* Σ' such that

1. the s equation of Σ is in Σ',
2. if Σ has no equation having an occurrence of s on its right-hand side, $\sigma_s(\Sigma) = \Sigma$; otherwise, all the equations of Σ except for one having an occurrence of s on the right-hand side, say the s' equation, are in Σ',
3. a new s' equation is included in Σ' that is identical to the s' equation in Σ except that one occurrence of s on the right-hand side of the s' equation is replaced by the right-hand side of the s equation,
4. there are no other equations in Σ'.

This transformation is denoted $\sigma_s(\Sigma) = \Sigma'$.

Definition 2.13 A *global substitution transformation under* s of a lattice equational system, Σ, denoted $\sigma^*_s(\Sigma)$, is defined by $\sigma^*_s = \sigma^n_s$ where n is the least non-negative integer such that $\sigma^{n+1}_s(\Sigma) = \sigma^n_s(\Sigma)$.

Definition 2.14 A *minimization transformation under* s of a lattice equational system, Σ, results in a new system Σ' such that

1. if the s equation of Σ is of the form[5] $s = X_1 \vee (X_2 \wedge s) \vee (X_3 \wedge \bar{s})$, then the equation, $s = X_1 \vee X_3$, is in Σ',
2. all the equations of Σ except for the s equation are in Σ',
3. there are no other equations in Σ'.

This transformation is denoted $\mu_s(\Sigma) = \Sigma'$.

When applied to an even equational system, Σ, a composition of the above transformations in the sequence

$$\mu_{s_1} \circ \sigma^*_{s_1} \circ \mu_{s_2} \circ \sigma^*_{s_2} \circ \cdots \circ \mu_{s_M} \circ \sigma^*_{s_M} \circ \sigma^*_{s_{M-1}} \circ \sigma^*_{s_{M-2}} \circ \cdots \circ \sigma^*_{s_1},$$

where $\{s_i | 1 \leq i \leq M\}$ is the set of non-trivial unknowns in Σ, yields a new system having only lattice expressions (constants) on the right-hand sides of its equations. These expressions can be demonstrated to constitute a well-founded solution for Σ. Such a composition of transformations in analogous to Gaussian elimination [6,14]. There is a phase of M pairs of minimization and global substitution operations followed by a phase of M global substitution operations. The first phase corresponds to "forward elimination;" the second phase corresponds to "backward substitution." We now know that for even lattice equational systems, at least, we can always find solutions. We have additional mathematical results that essentially guarantee a unique factorization for a solution, Γ. Those results

[4] To the best of our knowledge, the only use to be made of odd lattice equational systems is to implicitly encode alternatives. We believe, as does de Kleer, that alternatives are better encoded *explicitly* in assumptions.

[5] The s equation can always be rearranged to be in this form.

together with the "Gaussian elimination" just described can be used to generate in a structured fashion *all* the solutions[6] to every system, even or odd.

The lattice-based theory of reason maintenance suggests a number of algorithmic performance improvements, some oriented toward batch processing, some toward incremental processing. Certain improvements derive from topological considerations. A particular notion of connectivity can be attributed to equations, whence derive notions of strong connectivity [1,13,15] and strongly connected subsystems. Considerations on the structure of strongly connected subsystems lead to improved computational complexity results in suitably restricted cases. Similarly, purely algebraic considerations can lead to performance improvements in incremental algorithms when certain local conditions are met.

EMBEDDING ATMS AND LPT IN A LATTICE-THEORETIC FRAMEWORK

Having introduced our own formal machinery, we turn now to applying it to the description of the reason maintenance formalisms of de Kleer and Goodwin.

Assumption-based Truth Maintenance

De Kleer's basic[7] ATMS labelling algorithm can be cast in a lattice-theoretic framework as follows. Given de Kleer assumptions $\{A_i | 1 \leq i \leq n\}$, let the domain of the lattice, β, be the closure under meet, join and complement of $\{A_i | 1 \leq i \leq n\}$. An *atom* of β is an expression of the form $\wedge_{i=1}^{n} \tilde{A}_i$ where $\tilde{A}_i$ is either A_i or $\bar{A}_i$. Let A denote an atom of β and B_1 and B_2 be arbitrary elements. If A and A' are distinct atoms, $\perp = A \wedge A'$ and $\mathrm{T} = A \vee \bar{A}$. The partial order for β is defined as follows:

$$
\begin{aligned}
A \leq A_j &\equiv A_j \text{ appears uncomplemented in } \wedge_{i=1}^{n} \tilde{A}_i,\\
A \leq \bar{A}_j &\equiv A_j \text{ appears complemented in } \wedge_{i=1}^{n} \tilde{A}_i,\\
A \leq B_1 \wedge B_2 &\equiv A \leq B_1 \text{ and } A \leq B_2,\\
A \leq B_1 \vee B_2 &\equiv A \leq B_1 \text{ or } A \leq B_2,\\
B_1 \leq B_2 &\equiv \text{for every atom } A,\ A \leq B_1 \rightarrow A \leq B_2.
\end{aligned}
$$

For a given set, J, of ATMS justifications, a lattice equational system, Σ, over β can be constructed as follows. For each justified node, s, in J, Σ contains the s equation, $s = \vee_k \wedge_i X_{ik}$, where X_{ik} is the ith antecedent node or assumption of the kth justification of s. For each unjustified node s' in J, Σ contains the s' equation, $s' = \perp$.

Since de Kleer does not supply a formal proof of correctness of the ATMS algorithm, we have no direct way of establishing equivalence between the ATMS label propagation and solving the lattice equational system just given. On the other hand, de Kleer gives a formal specification for the ATMS solutions. We *can* show that solutions to the lattice equational encoding satisfy the specifications given for ATMS solutions. In particular, we demonstrate in [4] that the lattice equational solution is *sound, complete* and *minimal* in the sense that de Kleer uses those terms.

Logical Process Theory

Many of the formal concepts we introduced in §2 are either algebraic restatements or generalizations of Goodwin's graph-theoretic notions. Consequently, framing logical process theory within our Boolean lattice formalism is completely straightforward. The main task in LPT is to determine an admissible labelling of a given database, D, of inference steps. Briefly, an admissible labelling is a function from a language L to the set of labels $\{IN, OUT\}$, where every formula labelled IN has a well-founded argument. An inference step, d, is a triple, $\langle M, N, c \rangle$ where $M, N \subseteq L$, $c \in L$, and I is the set of all inference steps. The set M = M-antes(d) contains the non-monotonic antecedents of d, the set N = NM-antes(d) contains the non-monotonic antecedents of d, and c is a consequent of d. Given a database D, one can construct a lattice equational system Σ, such that a well-founded solution of Σ corresponds to an admissible labelling of D. To do this let $\{s_l | l \in L\}$ be a set of lattice unknowns and let β be the Boolean lattice consisting of the set $\{\mathrm{T}, \perp\}$. For each premise, l, let Σ contain the equation $s_l = \mathrm{T}$, otherwise of l is the consequent of some inference step in D let Σ contain the equation,

$$s_l = \bigcup_{d \in D'} \left[\left(\bigcap_{l_1 \in M} s_{l_1} \right) \cap \left(\bigcap_{l_2 \in NM} \bar{s}_{l_2} \right) \right]$$

where $D' = \{d | d \in D \wedge conseq(d) = l\}$ and M and NM are, respectively, the monotonic and non-monotonic antecedents of d. If l is not the consequent of any inference step in D we let Σ contain the equation $s_l = \perp$. Any well-founded solution, Γ, of Σ determines an admissible labelling if we associate T with IN, and ⊥ with OUT.

Extensions to ATMS and LPT

We have now seen how the model of reason maintenance proposed in §2 can embed both ATMS and LPT. Given that one wishes to have justifications that admit both non-monotonic and assumption-based support, the formalism that we have introduced can do this directly without appeal to these embeddings. De Kleer has used the *nogood* and *choose* mechanisms to simulate non-monotonic justification. The Goodwin formalism can accommodate assumptions by solving multiple labelling problems. It is instructive to contemplate natural extensions of each of their formalisms to treat (respectively) non-monotonicity and assumptions as first-class citizens.

The natural and immediate extension of the embedding of LPT considers solving n Goodwin systems of equations in parallel. We require that the n systems differ only in terms of the unknowns that correspond to premises, that is, unknowns, s, satisfying equations of the form $s = \mathrm{T}$. This is a semantically natural extension in that it corresponds to the reasoning agent's entertaining different sets of propositions as hypotheses. We augment the definitions of LPT as follows:

[6] Lattice equational systems with complemented unknowns have, in general, more than one solution. This is in contrast to the reason maintenance problem addressed by the ATMS. Although de Kleer counts each disjunct of an ATMS label as a "solution," from the point of view of lattice-theoretic reason maintenance the entire disjunctive expression of an ATMS label is a single solution.

[7] For the purposes of this discussion we exclude de Kleer's *nogood* mechanism from the basic ATMS. Any nogood environment can be accounted for if we algebraically identify the corresponding lattice expression with ⊥.

Definition 3.1 A *premise set* p is any subset of the language L. A *labelling* G is a function from $L \rightarrow P(P(L))$ where $P(L)$ denotes the power set of L. A *database* is a pair $\langle D,P \rangle$ where D is a set of inference steps and P is a set of premise sets. The antecedents of an inference step are non-empty.

Definition 3.2 Given a premise ser, p, and a labelling, G, we can define functions IN and OUT as follows:

$IN(G,p) = \{l \mid p \in G(l)\}$
$OUT(G,p) = \{l \mid p \notin G(l)\}$

Definition 3.3 An inference step d is *valid* with respect to a labelling G and a premise set p written Valid(p,d,G) if and only if M-Antes(d) $\subseteq$ IN(G,p) and NM-Antes(d) $\subseteq$ OUT(G,p).

Definition 3.4 G is a *relaxation* over database $\langle D,P \rangle$ if and only if

$\forall p \in P.$ IN(G,p) $= \{l \mid \exists d \in D.$ Valid (p,d,G) and (conseq(d) $= l$)$\} \bigcup p$

Definition 3.5 A labelling G is *well-founded* for a database $\langle D,P \rangle$ if and only if for all $p \in P$ there exists a partial ordering $<$ of $L \bigcup I$ such that:

$\forall l \in$ IN(G,p). $\exists d \in D.$ (conseq(d) $= l$) and Valid(p,d,G) and ($d < l$)
and

$\forall d \in D.$ Valid(p,d,G) $\rightarrow \forall l \in$ M-antes (d) ($l < d$)

Definition 3.6 An *admissible labelling* of a database $\langle D,P \rangle$ is a well-founded relaxation of $\langle D,P \rangle$.

Now we are in a position to encode a database in terms of a lattice equational system. Given a database, $\langle D,P \rangle$, let $\{s_l \mid l \sum L\}$ be a set of lattice unknowns. Let β be the Boolean lattice consisting of the power set of P. For each $l \in L$ construct the following equation[8]

$$s_l = \bigcup_{d \in D'} \left[\left(\bigcap_{l_1 \in M} s_{l_1} \right) \cap \left(\bigcap_{l_2 \in NM} \bar{s}_{l_2} \right) \right] \bigcup \{p \mid p \in P \text{ and } l \in p\}$$

where $D' = \{d \mid d \in D \wedge \text{conseq}(d) = l\}$ and M,NM as before. Let $\Sigma(D,P,L)$ be the set of all such lattice equations for each $l \in L$. Note that only if the l is the consequent of some inference step in D will the l equation contain lattice unknowns. Since D is a finite set there are finitely many equations with lattice unknowns. In [4] we formally demonstrate the equivalence of the above encoding to the extension of Goodwin's LPT that we informally described at the beginning of this subsection.

We extend de Kleer's ATMS to accommodate non-monotonic justifications by first reinterpreting the basic ATMS in terms of the embedding above of the extended LPT. The basic ATMS accepts a set of justifications and assumptions, and determines all possible contexts and their contents. We take the language, L, to be the set of nodes and assumptions. Each de Kleer justification, $\alpha_1, \alpha_2, \cdots, \alpha_n \Rightarrow \beta$ can be viewed as an inference step $\langle \{\alpha_1, \alpha_2, \cdots, \alpha_n\}, \varnothing, \beta \rangle$. Each de Kleer premise, x, is replaced by the set of justifications $\{A_i \Rightarrow x \mid 1 \leq i \leq n\}$ where $\{A_i \mid 1 \leq i \leq n\}$ is the set of assumptions. For any set of de Kleer justifications and assumptions, let $\langle D,P \rangle$ be a database, where D is the corresponding set of inference steps and P is the power set of the assumptions. It can be shown that for any admissible labelling, G, of $\langle D,P \rangle$, the set IN(G,p) corresponds to the context of the environment p. The non-monotonic extension follows immediately by allowing the non-monotonic antecedents of an inference step to be non-empty. Observe that LPT allows for the direct introduction of non-monotonic justifications whereas the basic ATMS mechanism does not. This is because de Kleer's native semantics for the ATMS is essentially propositional logic. To accommodate non-monotonicity some other semantics is required, hence our reinterpretation. A final note: lattice-theoretically framed reason maintenance, in its full generality does not appear to be naturally describable as an extension to either ATMS or LPT.

CONCLUSIONS

In the foregoing we have introduced a general model of the problem of reason maintenance couched in a lattice-theoretic framework. We believe that any of the reason maintenance systems familiar to us in the literature can be construed as solving systems of lattice equations. In particular, we have shown how to encode de Kleer's ATMS and Goodwin's LPT in this framework, as well as natural extensions of each of those systems to accommodate aspects of the other. We introduced the fundamental transformations of substitution and minimization and showed how they could be used to produce solutions. We have observed that we have other mathematical results that allow us to construct all solutions to all systems. We have also informally described some mathematical considerations that lead to very efficient algorithms in special cases.

We continue to investigate a number of issues in our ongoing research in reason maintenance. On the theoretical side, we believe that the assumption of a Boolean lattice that underlies our current results can be considerably loosened. In particular, we think that those results are preserved assuming only a lattice with complements. This is of both theoretical and practical interest as many measures of uncertainty are of a (non-distributive) lattice-theoretic nature [7]. No longer requiring distributivity, we can treat certainty as yet another kind of belief context to be propagated by constraints.

On the practical side, we are engaged in an implementation of the lattice-theoretic model of reason maintenance. As we gain experience in using this implementation, we will attempt to answer a number of questions. Do the theoretical improvements to which we have alluded have any practical effect on the kinds of problems that can be tackled? Are such improvements even necessary given that the worst case computational complexities are achieved through somewhat contrived pathological examples? Are the incremental algorithmic variants of practical value? If so, should they always be engaged, or should they be driven by some algorithmic or heuristic consideration?

[8] It is worth comparing this equational system with the one constructed for the unadorned Goodwin embedding. It differs only in the adjoined premise set.

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[3] Raymond Balbes and Philip Dwinger. *Distributive Lattices*. University of Missouri Press, Columbia, Missouri, 1974.

[4] Dan Benanav, Allen L. Brown, Jr., and Dale E. Gaucas. A lattice-theoretic framework for reason maintenance. Forthcoming.

[5] Garrett Birkhoff. *Lattice Theory*. Volume 25 of *American Mathematical Society Colloquium Publications*, American Mathematical Society, Providence, Rhode Island, third edition, 1967.

[6] Piero P. Bonissone and Keith S. Decker. Selecting uncertainty calculi and granularity: an experiment in trading-off precision and complexity. In L.N. Kanak and J.F. Lemmer, editors, *Uncertainty in Artificial Intelligence*, North Holland, Amsterdam, 1986.

[7] Johan de Kleer. An assumption-based TMS. Forthcoming.

[8] Johan de Kleer. Choices without backtracking. In *Proc. 4th Nat. Conf. on Artificial Intelligence*, pages 79-85, Austin, 1984.

[9] Johan de Kleer. Extending the ATMS. Forthcoming.

[10] Johan de Kleer. Problem solving with the ATMS. Forthcoming.

[11] Jon Doyle. *Some Theories of Reasoned Assumptions*. Computer Science Department Technical Report CMU CS-83-125, Carnegie-Mellon University, Pittsburgh, 1983.

[12] Shimon Even. *Graph Algorithms*. Computer Science Press, Potomac, Maryland, 1979.

[13] F.R. Gantmacher. *Matrix Theory*. Volume 1, Chelsea, New York, 1959.

[14] Michel Gondran and Michel Minoux. *Graphs and Algorithms*. Wiley, New York, 1984.

[15] James W. Goodwin. *An Improved Algorithm for Non-Monotonic Dependency Update*. Technical Report LITH-MAT-R-82-23, Linköping University, Linköping, Sweden, August 1982.

[16] James W. Goodwin. A process theory of non-monotonic inference. In *Proc. 9th Int. Joint Conf. on Artificial Intelligence*, pages 185-187, Los Angeles, August 1985.

[17] James W. Goodwin. Watson: a dependency directed inference system. In *Proc. of the Workshop on Non-Monotonic Reasoning*, pages 103-104, New Paltz, October 1984.

[18] James W. Goodwin. *WATSON: A Dependency Directed Inference System*. PhD thesis, Linköping University, Linköping , Sweden, Forthcoming.

[19] Matthew S. Hecht. *Data Flow Analysis of Computer Programs*. American Elsevier, New York, 1977.

[20] Marvin Schaeffer. *A Mathematical Theory of Global Program Optimization*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[21] L.A. Skornjakov. *Elements of Lattice Theory*. Hindustan Publishing Corporation, Delhi, India, 1977. Translated from the Russian by V. Kumar.

[22] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, New York, 1984.

Engineering Intelligent Systems: Progress Report on ABE[1]

Lee D. Erman
Jay S. Lark
Frederick Hayes-Roth

Teknowledge Inc, Palo Alto, CA 94303

Abstract

Intelligent systems combine the capabilities of expert/knowledge systems with conventional computer technologies, significantly extending the capabilities of either technology. Current expert/knowledge system tools do not address the key problems of intelligent systems engineering: large-scale applications and the reuse and integration of existing software components. ABE is a software architecture that directly addresses these problems.

ABE is a multi-level architecture for developing intelligent systems. ABE defines a virtual machine for module-oriented programming and a cooperative operating system that provides access to the capabilities of that virtual machine. On top of the virtual machine, ABE provides a number of problem-solving frameworks, such as blackboards and dataflow. Problem-solving frameworks support the construction of knowledge engineering tools, which span a range from knowledge processing modules to skeletal systems. Finally, applications can be built on skeletal systems. In addition, ABE supports the importation of existing software, including both conventional and knowledge engineering tools.

1. Background and Objectives

Expert systems have emerged from about fifteen years of research and development activities in applied Artificial Intelligence (AI). Numerous prototype applications have been demonstrated in government and industry, several commercial systems have been fielded, and the potential value of expert systems has become widely recognized. This value derives from their ability to provide a means for capturing, preserving, applying, and distributing human knowledge.

As experience has accumulated, it has become clear that most applications of this technology will not be as isolated, "expert" systems. Rather, the application of expertise (or more generally, *knowledge*) will occur in the larger context of *integrated* systems. We refer to such comprehensive systems, which combine the capabilities of expert/knowledge systems with those of more conventional systems, as *intelligent systems*. Intelligent systems differ from conventional systems by a number of attributes, not all of which are always present:

- *They pursue goals and objectives.*
  Goals form a larger context for the operation of the system. That context often makes static algorithms insufficient, requiring the system to exhibit more flexible behavior than conventional systems.

- *They incorporate, use, and maintain knowledge.*

- *They exploit diverse, ad hoc subsystems embodying a variety of selected methods.*
  The subsystems may be "intelligent" or conventional.

- *They interact intelligibly with users and other systems.*
  Intelligibilty is one of the most striking attributes of knowledge systems.

- *They allocate their own resources and attention.*
  Intelligent systems often need to be introspective and aware of their progress in applying their knowledge and subsystems in pursuit of their goals.

Most people now perceive a gap between what the intelligent systems technology should be able to do and what can be done today. While the technology holds great promise, it cannot yet supply solutions readily for many of the problems for which it should be applicable. Today, that technology transfers from research environments to applications chiefly through *knowledge engineering tools*. Prominent examples of these are the commercial products ART (from Inference Corp.), KEE (from Intellicorp), KnowledgeCraft (from Carnegie Group), and S.1 (from Teknowledge). These tools incorporate the best methods of applied artificial intelligence, and they reflect some of the best techniques for building expert systems. However,

[1] This is an early description of in-progress research. The ideas described here require experimental testing and will likely change. This does not constitute a commitment by Teknowledge to any product or service. ABE is a trademark of Teknowledge Inc.

these tools currently have several weaknesses. Generally, these reflect the small-scale and isolated nature of the applications that motivated the tools. Specifically, the major problems include the following:

- The best current tools are monolithic, single-purpose software packages. Hence they are hard to extend or apply beyond their current range of applications. They are also difficult to integrate with conventional data processing and computer technologies.
- The tools provide capabilities that are low-level. Most applications require the user to build a solution structure on top of those primitive capabilities. This design and implementation work is expensive and time-consuming, and requires a skilled and experienced knowledge engineer.
- The tools support a limited variety of data types and inference schemes.
- The inference schemes in current tools are built-in and practically hard-wired.
- Current tools do not support large-scale applications.
- The tools have been designed exclusively for uniprocessor implementations.
- The tools have not been designed in a way that makes them easy to port to alternative new machines.

To cover a larger set of potential applications, and to handle the larger context of intelligent systems, new tools are needed. In general, this new generation of tools must provide application developers with facilities to support *reuse* of previously-constructed components, incorporating the best methods of AI and knowledge engineering, *integration* of diverse component technologies, and *large-scale* application system development.

In particular, these tools need to support alternative implementations of the various knowledge engineering functionalities, and need to provide ways to configure intelligent systems and intelligent system tools out of modular functions. To be practical, such a new tool must consist of many preprogrammed functional modules and provide an effective technique for configuring these modules into larger systems. In addition to low-level capabilities, the tool must provide high-level, generic solutions to classes of problems (similar to the way that fourth-generation languages provide generic solutions for classes of database processing tasks); we call such partial solution structures *skeletal systems*. Finally, the tool must also allow the accumulation and incorporation of new and existing functional modules, both "intelligent" and conventional.

ABE is a new generation tool that satisfies the requirements for building intelligent systems. ABE is

1. an architecture and methodology for building intelligent systems by integrating heterogeneous components, including conventional (i.e., non-AI) components;
2. a modular and ever-expanding collection of knowledge-engineering capabilities, including skeletal systems; and
3. a useful initial set of proven, valuable knowledge-engineering capabilities.

open/extensible	The various levels in ABE will be accessible for modification and augmentation.
intuitive to learn	To support modification and augmentation, the various facilities, and their implementations, must be understandable.
*high performing**	ABE must be capable of being used to build systems that execute efficiently. It will allow for arbitrary tuning of application systems, in response to particular requirements.
*portable**	ABE will be portable to a variety of machines with relative ease. This includes both the ABE development environment and, especially, applications built on ABE.
*distributable/parallelizable**	ABE will support applications on a wide variety of machine architectures, especially those that are distributed and parallel.

* -- Features unimplemented at present; scheduled for phase 2.

Table 1-1: Key design characteristics of ABE

Certain characteristics of ABE are essential for its effective use. Table 1-1 lists these. Although some of these characteristics will not be implemented substantially until Phase 2, the design is committed to facilitate all of them.

1.1. Status and Plans

Direct work on ABE began in spring of 1985, under contract to the Defense Advanced Research Projects Agency (DARPA) and Rome Air Development Center (RADC). A preliminary implementation, in Common LISP on Symbolics workstations, is operational now (spring 1986). Section 4 describes portions of a recent demonstration of that version. A few selected projects will begin using an early delivery version of ABE in summer 1986. These early versions provide the basic ABE functionalities as described here, including a few frameworks and access to several existing general-purpose knowledge engineering tools. The phase 1 prototype version will be released to DARPA in 1987.

Proposed phase 2 work will emphasize higher performance, distributed configurations, versions for other computing equipment, and refinement and extension of the knowledge engineering capabilities.

2. Overview of ABE

Central to ABE is a multi-level architecture for developing intelligent systems. This architecture supports aggregations of cooperating, autonomous, problem-solving components. At the lowest level is a general model of computation. Organized around the central notion of communicating modules, the computational model is called *Module-Oriented Programming* (*MOP*). This model of computation provides the foundation and building blocks for the higher levels -- for expressing designs of intelligent systems as networks of cooperative problem-solving agents. The computational model also defines a virtual machine; this can be mapped onto underlying hardware and operating system environments.

The ABE architecture is a general-purpose software architecture for building intelligent systems. In particular, the ABE architecture supports the construction of problem-solving *frameworks* (see below). A framework is an architecture for building particular intelligent systems, and MOP is a meta-architecture for building intelligent system architectures.

The process of building an intelligent system is best accomplished by building up layers of capabilities. Each layer draws on the capabilities made available by the layer beneath it and presents a new set to the layer above it. New capabilities are often developed by modifying, restricting, or reconfiguring the capabilities from the next lower level.

The ABE architecture defines several functional levels in intelligent systems. These are listed in Table 2-1, in descending order. The ABE research effort is concentrating on providing levels 2, 3, and 4. Associated with each level is a class of user who uses the facilities at that level to provide the functionality of the next higher level: system designer, tool builder, knowledge engineer, and domain expert. Here a brief description is presented, with some examples of the facilities to be included in ABE's early delivery system. Section 3 provides more details.

The current *underlying computing environment* is the Symbolics LISP machine and Common LISP, augmented with Coral, an object-oriented language developed for ABE. The *virtual machine* is ABE's MOP (Module Oriented Programming system) and the *operating system* that supports it is called *KIOSK*. On this base the system designer layers problem-solving *frameworks* of various kinds. ABE's early versions include a dataflow framework and a blackboard framework.

Given one or more frameworks, the tool builder supplies *knowledge processing modules.* These might include capabilities such as a rule interpreter, and facilities for tasks such as maintaining knowledge bases, running cases, creating English-like translations of rules and other constructs, and producing explanations of system behavior. For example, one set of modules in ABE's library is built around structures for plans, and includes facilities for representing, creating, analyzing, and modifying them.

Level	Users of capabilities at this level
5. Intelligent system applications	End users
4. Knowledge engineering tools 4b. Skeletal systems 4a. Knowledge processing modules	Knowledge engineers and domain experts
3. Problem-solving frameworks	Tool builders
2. Virtual machine and cooperative operating system	System designers
1. Underlying computing environments	ABE implementors and system programmers

Table 2-1: Intelligent system levels, and associated users

A knowledge engineer can create a *skeletal system* by adding structure to and control over the knowledge processing modules and their interactions with other facilities (such as databases). One skeletal system in ABE's library, called *PMR* (which stands for "Plan Monitoring and Replanning"), analyzes an existing plan, monitors a database for critical assumptions of the plan that might become invalid, replans around violated assumptions, and interacts with various external agents about these activities.

The knowledge engineer customizes a skeletal system for a particular application domain by replacing some of the generic constructs with more appropriate terms. One example application domain of the PMR is planning for offensive air strike missions. For this, terms such as "flight", "target", and "ordnance" are appropriate. This customized skeletal system is called AS-PMR.

Finally, a domain expert adds to this skeletal system knowledge of specific objects and relationships to create a domain-specific *application system.* For AS-PMR, this includes such information as characteristics of particular aircraft models, targets, and ordnances. It also includes the particular rules which govern their interactions, e.g., that a particular ordnance is available on a particular aircraft and is able to destroy a particular target.

2.1. What ABE Addresses

ABE's design addresses the weaknesses described in Section 1.

An important aspect of ABE's design is the multi-level architecture and the particular choice of levels. The multiple levels provides flexibility, and have been chosen to support the goals of building intelligent systems by selecting, customizing, and combining modules from growing libraries.

As shown in Table 2-1, certain classes of users are associated with each of the levels. Although in practice a single individual might encompass more than one of these functions, the multi-level organization also supports specialization of these user roles.

The MOP computational model provides flexibility for expressing a wide range of cooperative problem-solving architectures, each with its own control and communication scheme. Diverse schemes, from highly centralized to fully distributed, are needed to implement the large variety of intelligent system applications. The virtual machine supplied with the computational model can be mapped onto a wide range of underlying hardware/OS environments. Primary targets are parallel and distributed environments. The model's flexibility can also be exploited within a single ABE application system; various subsystems can be implemented in heterogeneous problem-solving frameworks, and they can be implemented on heterogeneous computing facilities.

The developer of a framework needs both a general-purpose, open organization and a strong computational model. The open organization of communicating modules provides the needed flexibility, and the MOP computational model gives a strong semantic basis for understanding the computational properties of the systems built.

Most current efforts at building and improving tools are concentrating on improving particular AI techniques used in knowledge engineering tools. A major emphasis in the ABE project is on providing an organizing framework and facilities that allow such tools to be accumulated and re-used. ABE complements these other efforts, since it is able to import and integrate their efforts.

Another thrust of ABE is in skeletal systems. With a few notable exceptions (especially see [Clancey 83] and [Chandrasekaran 83]), the field of intelligent system engineering has largely ignored and skipped over this level, in favor of programming shells (at a level below) and applications (above). A typical knowledge-system project starts from a shell (e.g., backward-chaining rules over frames) and creates a new application system, bypassing the skeletal system level. However, explicit identification and design of generic, skeletal systems has several important advantages, including

- increased modularity of systems,
- increased reusability of solutions or parts of solutions,
- easier knowledge acquisition, and
- easier maintenance of the application.

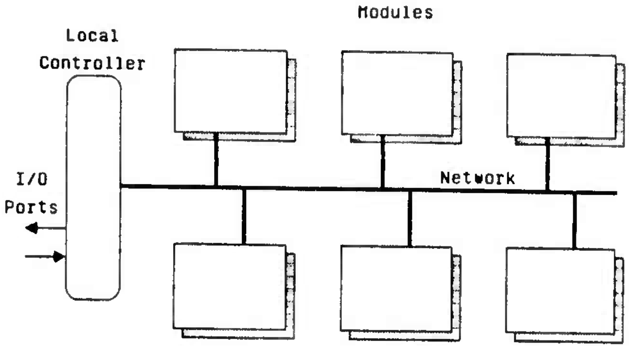


Figure 3-1: A standard KIOSK module organization

3. System Description

We now discuss in greater details levels 2 through 4 of the ABE architecture.

3.1. Virtual Machine/OS

The base level of ABE is the *virtual machine/cooperative operating system* level. The virtual machine designed for ABE embodies a computational model called *Module-Oriented Programming* (MOP). The cooperative operating system that supports this model is called *KIOSK*.

At the virtual machine level, an ABE system is composed of a set of *modules* -- see Figure 3-1. Modules communicate with one another by sending *messages* over *networks*. Modules can *connect* to many networks simultaneously and can communicate with each network independently. Modules are either *primitive* or *recursively composed* from another set of modules communicating on a network. Each composite module has a *local controller*, which manages both the communication activities on its network and the communication between the composite module as a whole and the networks external to it. The local controller also controls the allocation of processing resources among the modules on the network.

KIOSK is called a *cooperative operating system* for the MOP virtual machine because it provides services analogous to the services provided by a standard operating system. These services include module and network creation, communication primitives, computational resource modeling, and primitive resource allocation schemes. KIOSK provides an abstraction barrier which allows the ABE system to be mapped to arbitrary physical computing environments. The term "KIOSK" is used to refer to both the MOP virtual machine and the KIOSK cooperative operating system.

3.2. Problem-solving Frameworks

The next level of ABE consists of *problem-solving frameworks*, also referred to as *problem-solving architectures* or *programming languages*. We have not found a satisfactory distinction between programming languages and problem-solving architectures, so we group them together in one level.

A framework is a collection of design choices: control and resource allocation regimes, communication protocols, shared languages (syntax), and computational organizing principles. These design choices manifest themselves in ABE within the local controller of a module, because the local controller has responsibility for all of these things. An ABE system may contain instances of many different frameworks, although not all frameworks can coexist on the same network.

A framework may present a view of the world quite different from the underlying KIOSK "modules-on-a-network" view. For example, a framework may designate certain modules as having some special significance to the global operation of the system (e.g., the shared blackboard module in a blackboard-oriented framework). Also, the framework may provide its own visual representation which totally masks the underlying virtual machine.

ABE's library currently includes two primary problem-solving frameworks: a dataflow framework (called *DF*) and a blackboard framework (*BBD*), both of which are still evolving. Additional frameworks will be added. Also, the separation between the problem-solving frameworks and the underlying MOP/KIOSK level is still evolving, based on experience developing the frameworks.

The DF framework implements many of the concepts found in standard dataflow languages [Davis 82]. It also includes extra data structuring techniques and a semi-deterministic scheduler. A program for the DF framework consists of a number of independent processing modules which perform computations and communicate with each other and the outside world. The data structuring supports the use of abstract datatypes (ADTs) as the tokens passed between processing modules. The semi-deterministic scheduler supports building programs with side effects (such as communicating with the external environment).

BBD is a framework based on the blackboard metaphor [Erman 80] A blackboard system consists of a number of individual computation agents, known as knowledge sources (KSs), which communicate with each other through a shared global database, known as the blackboard. KSs monitor the blackboard with trigger patterns. When the posting of a datum on the blackboard matches a KS's trigger pattern, the KS triggers itself. The triggering operation informs the BBD scheduler that a particular KS was triggered by a particular set of blackboard objects. This KS instantiation (KSI) is itself posted on the blackboard.

The BBD interpreter has a scheduler whose function is to select a KSI for execution. This scheduler is very simple, and is refered to as a base scheduler. A special set of KSs known as "scheduling KSs" can manipulate the set of KSIs on the blackboard, thereby producing different scheduling behavior from the base scheduler.

3.3. Knowledge Engineering Tools

The third level of ABE is the *knowledge engineering tools* level. This level spans a range, with *knowledge engineering capabilities* at the lower end and *skeletal systems* at the higher. A primary research goal at this level is to develop a methodology for modularizing, describing, cataloging, reusing, and combining knowledge engineering tools.

Skeletal systems can be characterized as a way to organize and control knowledge and other facilities to solve a class of problems. In an ABE system, a skeletal system is a particular set of modules defined within a particular framework. Many of these modules will have mechanisms for customization by the knowledge engineer with application-specific knowledge. Other modules may serve as place holders, which the knowledge engineer will replace by totally new, but functionally equivalent, application-specific modules.[2] Yet other modules may represent a class of modules or a generator of new modules which the system will create at runtime. In general, a skeletal system is a partially instantiated assembly of modules for solving a class of application problems.

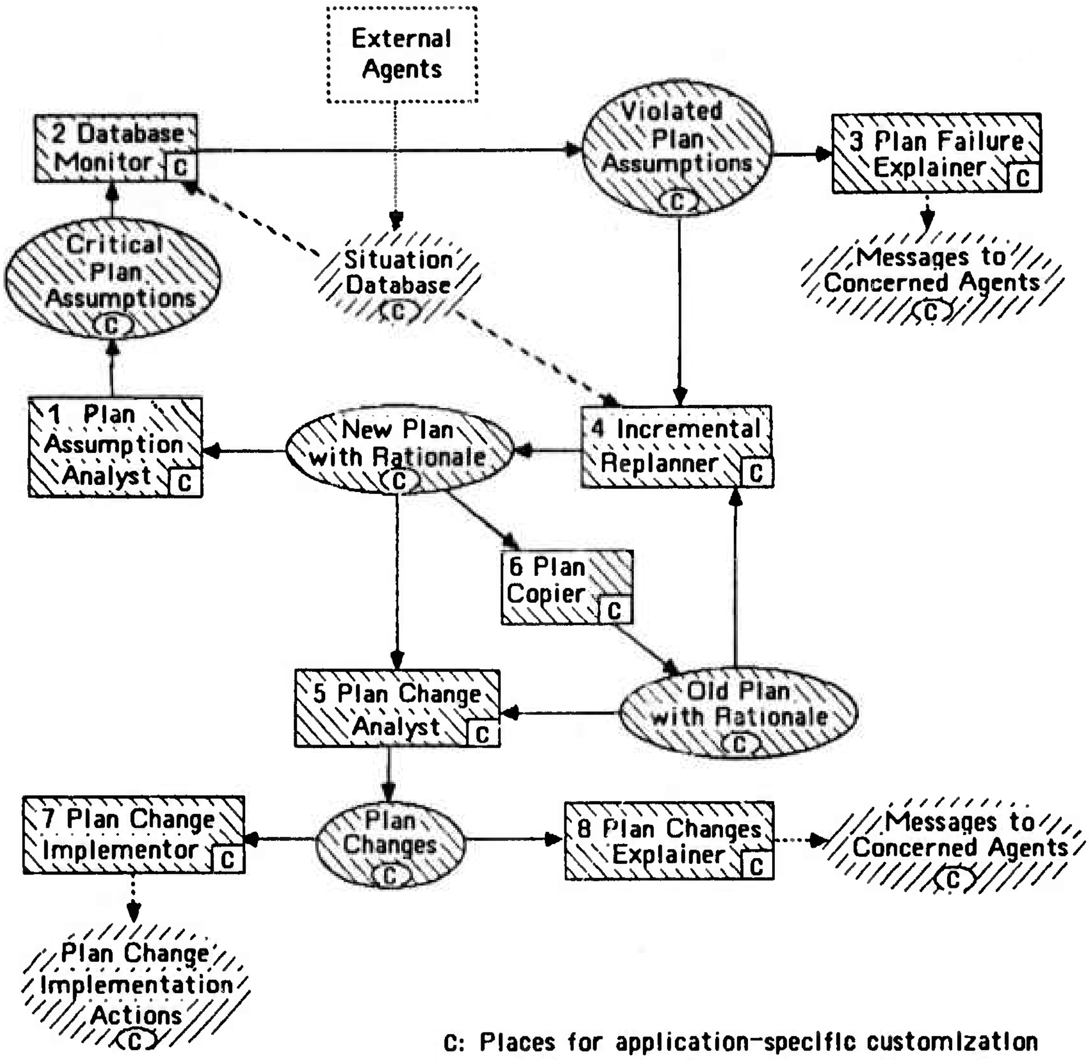
The Plan Monitoring and Replanning (PMR) Skeletal System

The PMR is the first skeletal system implemented for the ABE library. It is a generic structure for adaptive replanning -- keeping a plan consistent with a changing world. More specifically, it provides facilities to

- analyze a plan to determine its key assumptions about the world,
- monitor a database describing the unfolding world situation, looking for key assumptions that no longer hold,
- incrementally replan around these problems, and
- keep selected agents informed of important changes in the situation and the plan.

This skeletal system is independent of any particular application or application domain. For example, we have built one instance of the PMR customized for planning of air strike missions. We have built a second application, in the domain of personal travel planning.

[2] This can be viewed as an extreme form of "customization."


External Agents
2 Database Monitor C
Violated Plan Assumptions C
3 Plan Failure Explainer C
Critical Plan Assumptions C
Situation Database C
Messages to Concerned Agents C
1 Plan Assumption Analyst C
New Plan with Rationale C
4 Incremental Replanner C
6 Plan Copier C
5 Plan Change Analyst C
Old Plan with Rationale C
7 Plan Change Implementor C
Plan Changes C
8 Plan Changes Explainer C
Messages to Concerned Agents C
Plan Change Implementation Actions C

C: Places for application-specific customization information and knowledge

ACTIVE MODULES:

1) Determine the plan's key assumptions
2) Monitor the database for changes that may be critical to the plan
3) Send messages to interested agents describing the problems of interest to them
4) Invoke an incremental replanner to patch around the problems
5) Compare the old plan to the new one to identify changes
6) Save the newly created plan
7) Send messages ordering the implementation of plan changes
8) Send messages to interested agents explaining the changes of interest to them

Figure 3-2: PMR: Plan Monitoring and Replanning skeletal system (dataflow version)

There are actually two different versions of PMR. The first one is implemented in the DF (dataflow) framework. Figure 3-2 shows the modules and places in the DF version of the PMR skeletal system. The second version, is implemented in the BBD (blackboard) framework. It reuses the PMR modules, and incorporates additional ones that exploit the greater control flexibility of the BBD framework to implement more complex scheduling behaviors of its component modules. Section 4 shows some details of these skeletal systems and applications.

Basic Facilities

At the lowest level of knowledge engineering tools, ABE allows the user to program modules in Common LISP. Above that level, the Coral object-oriented programming facility, embedded in Common LISP, can be used.

Somewhat higher still, ABE supports the concept of abstract data types (ADT) as a commonly useful methodology for defining and accessing structured objects. For example, most DF programs use ADTs to implement the data tokens passed among the process modules. Similarly, the BBD framework uses ADTs to implement blackboard objects. In addition to this general ADT facility, ABE's initial library contains particular ADTs for the plan structures used in the PMR skeletal system; these (as well as the individual PMR modules) are available for reuse, perhaps with some customization or other modification.

ABE's initial library contains an abstract, symbolic database facility, known as GDB (generic database), which can be used to define, store, and retrieve symbolic structures. GDB is used by various PMR modules, both as internal databases and to represent the external world. There are two alternative implementations of the GDB -- one in MRS and one in Prolog.

Integration of Pre-existing KE Tools

One goal of ABE is to allow the use and combination of existing knowledge engineering tools of various kinds. The early delivery version will contain interfaces for several of these, including MRS, Knowledge Craft, and S.1.

MRS [Russell 85] is a research system developed at Stanford University, and available under license from Stanford. MRS provides general-purpose facilities (with an underlying first-order predicate calculus basis) for representation and, especially, control. MRS is highly articulated and modular, and therefore allows intimate integration with relative ease. A user can access MRS directly from within Common Lisp code in ABE (e.g., from within DF or BBD modules). As noted above, ABE's library also contains a version of its GDB symbolic database facility implemented in MRS.

Knowledge Craft [Knowledge Craft 85] is a commercial product of Carnegie Group, Inc. It is a general purpose knowledge-engineering "shell". The heart of Knowledge Craft is a schema (frame) system, called CRL ("Carnegie Representation Language"). Knowledge Craft also has several separate facilities, including implementations of the OPS5 forward-chaining rule system and the PROLOG logic programming language. Each of these facilities is augmented to allow access to CRL schema. ABE's initial Knowledge Craft interface supports the implementation of abstract data types as CRL schemata and, in general, translating between schemata and ADTs. The ABE library also contains a Knowledge Craft version of the same abstract symbolic database mentioned above for MRS. Finally, the interface also allows fairly direct access to any of Knowledge Craft's facilities.

The early delivery library will also supply an interface to S.1 [Erman 84]. A commercial product of Teknowledge, S.1 is a higher-level knowledge engineering "shell". S.1 provides a backward-chaining, rule-based system that also allows for expression of procedurally represented knowledge, usually for control purposes.

4. Examples

This section describes some of the features of ABE through examples of their operation and use.

Figure 4-1 shows part of the current ABE catalog. At the lowest level are the programming languages, including Common LISP, Coral (an object-oriented system built on Common LISP), MRS, and three components of Knowledge Craft: Carnegie Representation Language, Prolog, and OPS5. S.1 will be available soon.

Above the languages are the frameworks -- the various ways in which modules can be implemented. Each framework has its identifying icon. The BBD *blackboard* and DF *dataflow* frameworks are described in Section 3. The TX *transaction* framework is used for implementing a server module (such as a database) that has one or more client modules. The set of facilities for abstract data types (*ADTs*) is also considered a framework. A module implemented in the *blackbox* framework just has arbitrary code, not internally analyzable by ABE. An *importer* module is a special case of a blackbox module, one which imports some foreign code with a wrapper that makes it ABE-compatible. Finally, the *catalog* facility is itself a form of framework.

Above the frameworks is a collection of modules of various capabilities. Included are a number of ADTs (e.g., for plans and actions). Finally, there are several skeletal systems, domain-specific customizations, and applications. An application is a skeletal system that has already been customized.

The largest window in Figure 4-2 shows the central part of the dataflow version of the plan monitoring and replanning (PMR) skeletal system. Using standard dataflow notation, processing modules are shown as rectangles and token places as ovals. The dashed oval indicates the input to the PMR as a whole. The Situation Monitor module is itself implemented as a dataflow program, and that is shown in the upper right-hand window. This version of the PMR uses the MRS implementation of the symbolic database system for the world situation, shown in the lower right-hand window. While the Situation Monitor is an example of hierarchically composed module, the connections between the Situation Database and its clients via the TX (transaction) framework is an example of non-hierarchical interactions between frameworks; we call such interactions *meshing*.

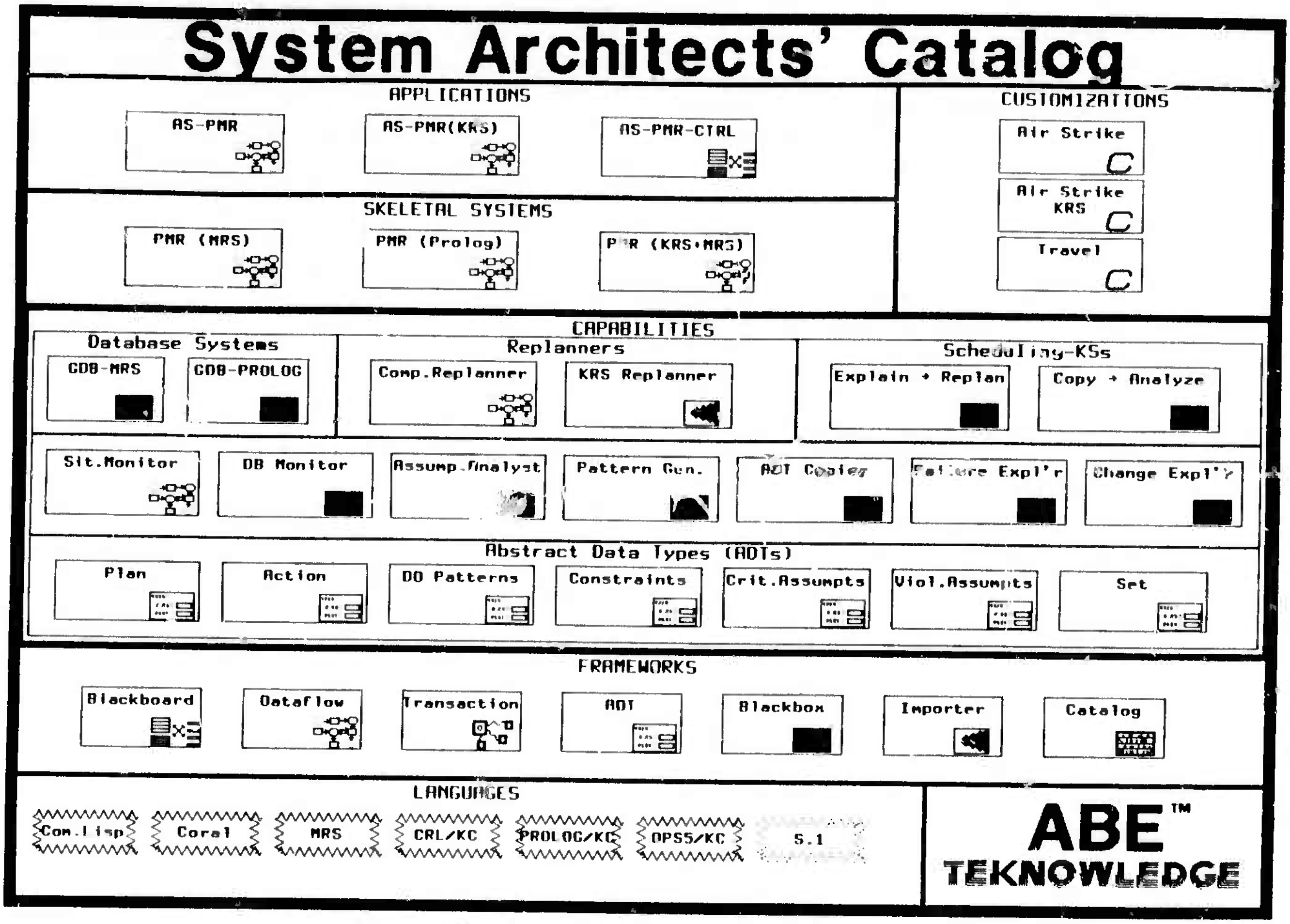


Figure 4-1: A portion of the current catalog

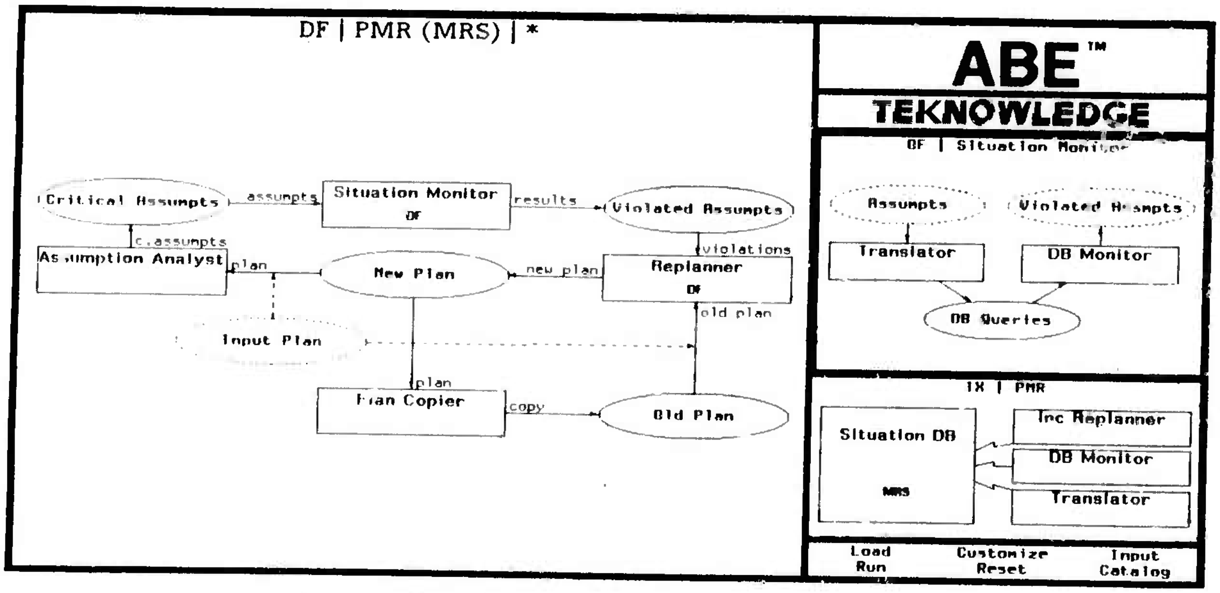


Figure 4-2: The core of the PMR system, with views of the Situation Monitor module and the Situation Database server and clients

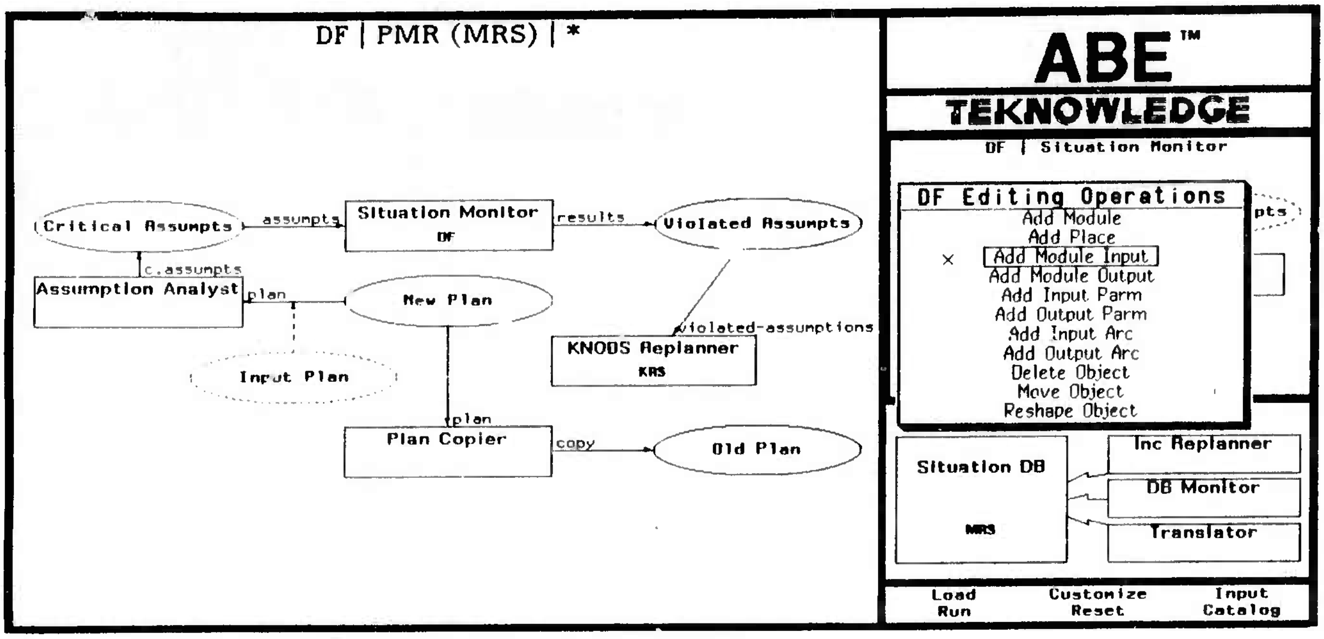
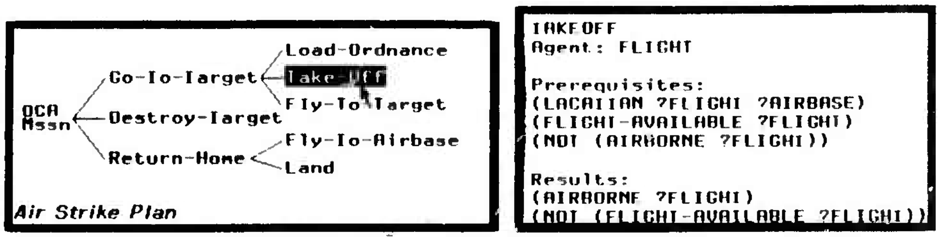
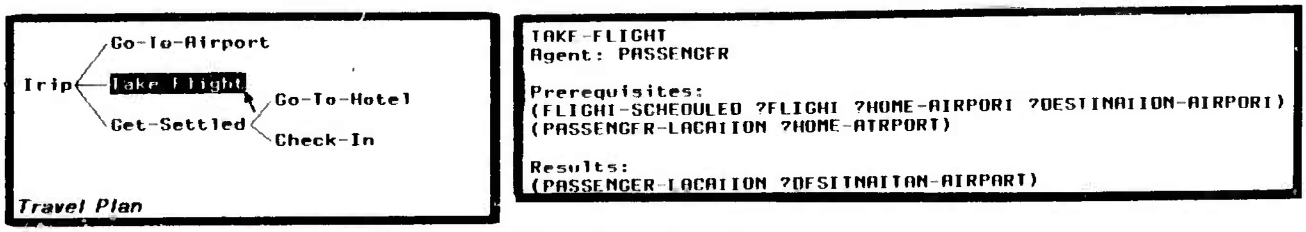


Figure 4-3: The KNOBS Replanner (KRS) being edited into the dataflow graph to replace the composite replanner



```
TAKEOFF
Agent: FLIGHT

Prerequisites:
(LACATIAN ?FLIGHT ?AIRBASE)
(FLIGHT-AVAILABLE ?FLIGHT)
(NOT (AIRBORNE ?FLIGHT))

Results:
(AIRBORNE ?FLIGHT)
(NOT (FLIGHT-AVAILABLE ?FLIGHT))
```

Figure 4-4: The Air Strike application: a plan's structure and example action

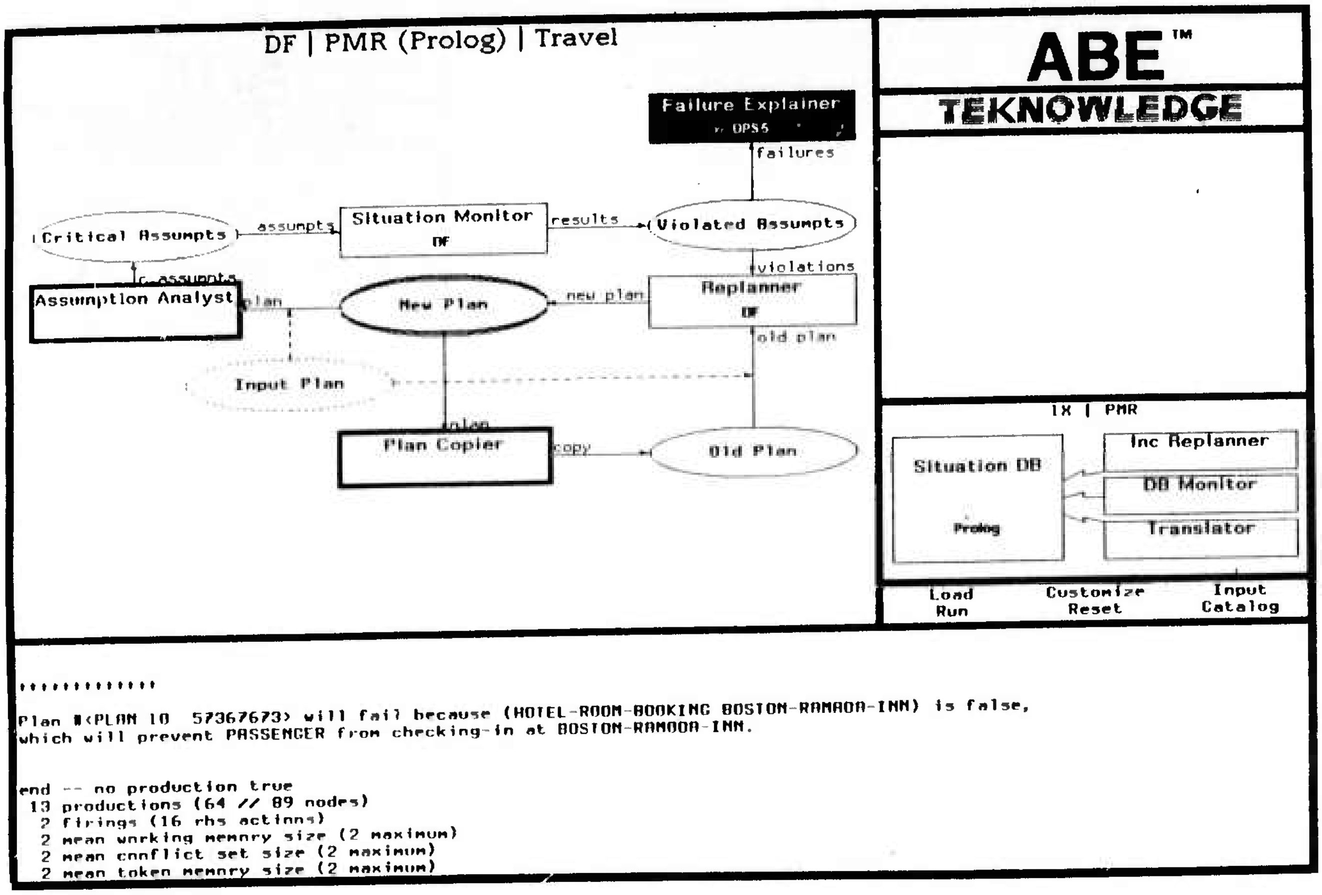


```
TAKE-FLIGHT
Agent: PASSENGER

Prerequisites:
(FLIGHT-SCHEDULED ?FLIGHT ?HOME-AIRPORT ?DESTINATION-AIRPORT)
(PASSENGER-LACATION ?HOME-ATRPORT)

Results:
(PASSENGER-LACATION ?DESITNATAN-AIRPART)
```

Figure 4-5: The Travel Planning application: a plan's structure and example action

Figure 4-3 contains an example of replacing one module with another. The relatively simple replanner implemented originally for the PMR is being replaced by the KNOBS Replanning System (KRS), imported from the Mitre Corporation. (See [Engelman 79] for a description of the earlier KNOBS work which led to KRS.) The replacement is done graphically, by deleting the box representing the original replanner and connecting in a box representing KRS.

Two applications of the generic PMR are shown in the next two figures. The Air Strike application deals with planning offensive counter air missions and is similar to that used in the KNOBS system. Figure 4-4 shows the structure of an air strike plan and an example of one action of that plan. The Travel Planning application, shown in Figure 4-5, handles trips from one's home to a hotel in a distant city. ABE's current catalog contains customizations for specializing the PMR to each of these applications. Each customization includes definitions for actions and states, plan structures, and example test-case plans and situations.



```
!!!!!!!!!!!!!

Plan #<PLAN 10  57367673> will fail because (HOTEL-ROOM-BOOKING BOSTON-RAMADA-INN) is false,
which will prevent PASSENGER from checking-in at BOSTON-RAMADA-INN.


end -- no production true
 13 productions (64 // 89 nodes)
  2 firings (16 rhs actions)
  2 mean working memory size (2 maximum)
  2 mean conflict set size (2 maximum)
  2 mean token memory size (2 maximum)
```

Figure 4-6: The PMR, with the Prolog database system and the added Failure Explainer module

Figure 4-6 shows a version of the PMR with the Situation database implemented by Prolog, in place of the MRS implementation. This figure also shows the central part of the PMR augmented with a module that generates explanations of the detected plan failures. Near the bottom of the figure, the two-line output of that module is shown. This module is implemented in OPS5 (and some internal tracing of the OPS5 operation is also shown at the bottom of the figure).

The control regime provided with the dataflow framework allows the system architect to configure a system without having to be overly concerned about control. However, if the architect wants to specify more fine-grained control, a dataflow framework is inappropriate or poor. For example, it is difficult to specify in a dataflow framework that the Failure Explainer should operate before the Replanner, which is probably desirable for the PMR. Figure 4-7 shows the same five processing modules of the DF PMR functioning as knowledge sources within the BBD blackboard framework. In addition, two scheduling knowledge sources have been added, to provide explicit scheduling knowledge.

Figure 4-7 shows the state of execution after the Failure Explainer and Replanner have both been triggered by the Situation Monitor posting on the blackboard one or more violated plan assumptions. The triggering of those two knowledge sources has also triggered the Explain-Failure-Before-Replanning scheduling knowledge source. Figure 4-8 shows the result of that scheduling knowledge source -- it has explicitly ordered on the agenda (near the top of the figure) the two other pending sources, to achieve the desired sequencing. This example shows not only the multiple frameworks and why they are desirable, but also shows that ABE's module-oriented programming style allows for the reuse of modules within a variety of frameworks.

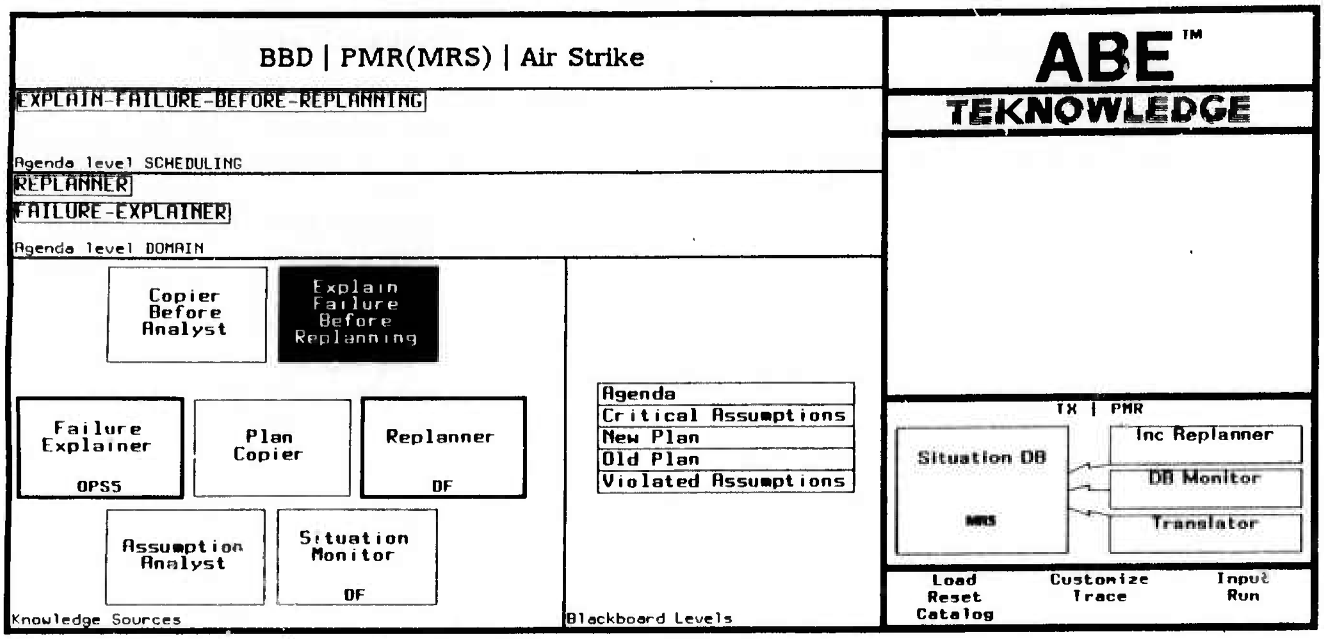


Figure 4-7: The Blackboard version of the PMR, *before* execution of the Explain-Failure-Before-Replanning scheduling knowledge source

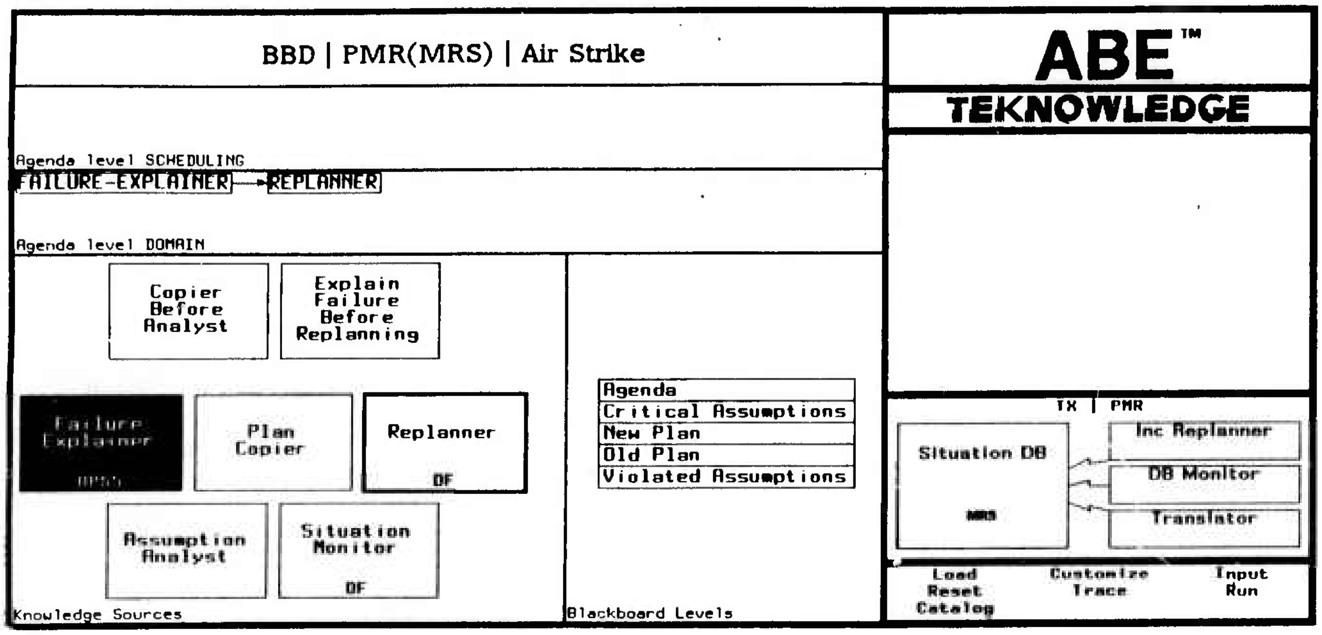


Figure 4-8: The Blackboard version of the PMR, *after* execution of the Explain-Failure-Before-Replanning scheduling knowledge source

Acknowledgments

A number of people have contributed substantially to ABE. Michael Fehling has been particularly instrumental in the conception and design of the MOP model of computation, strongly influenced by his Schemer system. William Clancey and James Bennett have also participated in the general design. Terry Barnes and Kamal Bijlani have contributed to the design work and have been central to the implementation. Bruce Bullock and Neil Jacobstein have provided managerial support and technical guidance. Stephanie Forrest and Wayne Caplinger have provided helpful comments on drafts of this paper.

RADC supplied the KRS system, implemented by the Mitre Corporation. Lt. Kevin Benner provided valuable help in importing KRS. B. Chandrasekaren and John Josephson and their colleagues at Ohio State University helped in formulating the Air Strike application. Carnegie Group Inc. provided help in importing Knowledge Craft.

This research is partially sponsored by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, NY 13441-5700 and the Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209, under contract F30602-85-C-0135.

REFERENCES

[Chandrasekaran 83]
Chandrasekaran, B.
Towards a Taxonomy of Problem-Solving Types.
AI Magazine 4(1):9-17, Winter/Spring, 1983.

[Clancey 83] Clancey, W. J.
The Advantages of Abstract Control Knowledge in Expert System Design.
In *Proc. National Conf. on Artificial Intelligence*, pages 74-78. Washington, D. C., August, 1983.

[Davis 82] Davis, A. L.
Data Flow Program Graphs.
IEEE Computer 15(2):26-41, February, 1982.

[Engelman 79] Engelman, C., C. H. Berg, M. Bischoff.
KNOBS: An Experimental Knowledge Based Tactical Air Mission Planning System and a Rule Based Aircraft Identification Simulation Facility.
In *Proc. 6th Int. Joint Conf. on Artificial Intelligence*, pages 247-249. Tokyo, 1979.

[Erman 80] Erman, L. D., F. Hayes-Roth, V. R. Lesser, and D. R. Reddy.
The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty.
Computing Surveys 12(2):213-253, June, 1980.

[Erman 84] Erman, L. D., P. E. London, and A. C. Scott.
Separating and Integrating Control in a Rule-Based Tool.
In *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, pages 37-43. Denver, CO, December, 1984.

[Knowledge Craft 85]
Knowledge Craft Manual Guide
Carnegie Group Inc., Pittsburgh, PA, 1985.

[Russell 85] Russell, S.
The Compleat Guide to MRS.
Technical Report KSL-85-12, Stanford Knowledge Systems Laboratory, Computer Science Dept., Stanford University, 1985.

EXPLANATION, PROBLEM SOLVING, AND NEW GENERATION TOOLS: A PROGRESS REPORT

B. Chandrasekaran and John Josephson
with contributions by Michael C. Tanner, Anne Keuneke
David Herman, Dean Allemang, and Todd Johnson.

Laboratory for Artificial Intelligence Research
The Ohio State University

1. Background of the Research and Overview of Accomplishments

1.1. Introduction

This is a progress report on our project on "Explanation in Planning and Problem Solving Systems." It is being written approximately at the 15-month mark. Conceptual frameworks for generation of explanation of two kinds have been built: one for explaining how decisions are made during problem solving, explaining control strategies as well as other aspects of run-time behavior, and the other to give a planner the capacity to represent an understanding of its own plan fragments, and thus to explain to the user how a plan is meant to work. A prototype mission planning system with some explanation capabilities has been built, and a number of high-level knowledge-based system construction tools have been built with features that facilitate knowledge acquisition, system implementation and explanation generation. Two of these tools (DSPL and HYPER) are discussed in this report 3, one (CSRL) predates this explanation project and has been extensively reported on [4, 5, 6, 7, 12] and several others are in various stages of design and implementation. Together they will constitute a high-level tool box for the construction of knowledge-based systems. They will be useful for building a variety of planning, diagnostic, abductive, and retrieval systems, and systems which are combinations of these types. These tools have as design features a number of "hooks" for the attachment of explanation synthesis tools.

In the first stage of the project, we have chosen "routine planning" as a task for which to build a prototype. In particular, a planning task for Offensive Counter Air (OCA) missions was chosen for analysis and implementation.

1.2. A Decomposition of the Explanation Problem

A brief recapitulation of our decomposition of the problem of explanation generation in knowledge-based systems is in order at this stage to motivate the issues discussed in this report. In our original proposal we had argued that there are three top-level components that can be distinguished:

- i) How a problem solver represents its own problem solving activity and retrieves the relevant portions appropriately in response to user queries. Here the language in which the problem solving behavior is encoded is very important for whether the response is perspicuous.

- ii) How user's goals, state of knowledge, etc, are used to *filter and shape* the output of the process in i) above so that the explanation is responsive to user's needs, is not overly and unnecessarily detailed, is couched in terms which are appropriate to the user's level of understanding, etc. Here *user modeling* is an important issue.

- iii) How an appropriate human-machine interface displays and presents the information to a user in an effective way. Here the issues include natural language understanding, natural language generation, and principles of effective graphical displays.

We argued in the original proposal that no matter how good the theories are for ii) and iii), if a poor representation is adopted for i), then at best inappropriate explanation will be presented packaged in a good interface. That is, the basic content of the explanation is generated in stage i). Thus we need to pay great attention to how *a problem solver can comprehend its own problem-solving activity*. Much of our Phase-I effort is devoted to developing a good theory of this, testing it by implementation of a prototype system, etc.

The explanation of problem solving itself in our analysis has 3 components:

1. Explaining why certain decisions were made or were not made. This has to do with how the data in a particular case related to the knowledge for making specific decisions or choices.

2. Explaining the problem solving strategy and the control behavior of the problem solver. This would typically be at a higher level of abstraction than answers to 1.

3. Explaining the elements of the knowledge base itself. For example, if the knowledge base contains plan fragments which are to be instantiated and assembled into longer plans, the problem solver may be called upon to explain the rationale behind the plan fragments. Similarly if, during a particular diagnosis, a trouble-shooter uses the knowledge that a low voltage between certain terminals is evidence for a particular malfunction, a user might want to know the reasoning behind the knowledge fragment.

It should be noted that typically 1 and 2 above involve the *run-time* behavior of a problem solver (and thus cannot in general be precompiled without running into combinatorial problems), while explanation structures for 3 above can in principle be attached to the knowledge fragments at the time the knowledge base is put together.

1.3. Overview of the Work So Far

Our work in Phase I of the project has contributed to each of the above types of explanation. Our theoretical position is that in order to generate explanation of type 1 and type 2 *at the appropriate level of abstraction*, the problem so'ving process needs to be represented at what we have called the *generic task level*. The essence of the argument is that most of the current approaches to expert system construction use knowledge representation languages and control primitives at too low a level of abstraction (the rule-frame-logical formulae level), and this makes both system design and explanation difficult, since the system designer often has to transform a higher-level problem into the lower-level implementation language. We have identified a set of higher-level building blocks in terms of which systems can be conceptualized, designed and implemented. The basic explanation constructs are then available closer to the conceptual level of the user than they would be if they had to be extracted from the implementation language level. This point of view has led us to propose a new approach to the design of knowledge-based systems, namely the generic task level. In order to facilitate expert system construction at this level, we have devoted a considerable amount of energy to the design and implementation of a set of higher level tools for the construction of expert systems of various types.

The theory itself is being put to the test at this stage for what can be called *routine planning* or *routine design* tasks. We have identified the OCA mission problem as a problem of this type, used one of our generic task languages (DSPL) for both knowledge acquisition and system implementation, and by using the constructs in DSPL effectively, have been able to show how *explanation at higher and more appropriate levels of abstraction* can be automatically generated from the problem solver. Some of the tools that we have built to lay a proper foundation for explanation-capable expert system are described in a later section (3).

Explanation of Type 3 above, viz., explanation of knowledge fragments in the knowledge base, has been approached by us in the context of the OCA mission as *explanation of plans* (i.e., the plans themselves, not the planning process). We propose that plans can be viewed as *devices*, and as such an earlier representation developed in our laboratory for representing a device's functioning can be used effectively for explaining plans.

1.4. Organization of the Progress Report

The work that has gone on in our laboratory is reported in two separate papers in this *Proceedings*. In this paper, we give a description of our work on the design and implementation of the MPA system for mission-planning, including the generation of explanation of various types. It ought to be emphasized that the MPA project is not completed, and so what is reported here should be viewed mainly as an interim report. Both the design of the planner and the explanation components are still in the process of further analysis and expansion. We also include in this paper reports on two high-level tools that we have been building for the construction of knowledge-based systems: DSPL for construction of systems that help with routine design (including planning), and HYPER, for deciding how data match hypotheses, a component of a number of distinct kinds of problem solving. These two are part of a tool-kit that includes CSRL, a language already developed and reported on, and others that are in various stages of implementation.

We include an additional paper reporting on the conceptual and theoretical foundations for much of our work on explanation. This provides the rationale for using the *generic tasks* approach, both for system construction and for explanation.

1.5. Near Term Plans

We propose to continue and add functionalities to the MPA System, and also to increase the range of explanations offered by the system. We also plan in the near term to show how our approach to explanation can be incorporated to a diagnostic or situation assessment task.

2. MPA: A Mission Planning Assistant in the KNOBS Domain

2.1. Design and Construction of the Mission Planning Assistant

David Herman, Anne Keuneke,
Michael C. Tanner, Ron Hartung, John Josephson

One major application area relevant to the Strategic Computing Program is planning and plan support systems. Our interest in planning concerns the explanation facilities that will be necessary in expert systems that assist in planning. This report summarizes our current work in this area in the domain of tactical mission planning. After investigating KNOBS [10], an existing mission planning system, we have developed our own mission planning system (MPA) using our generic task approach to building expert systems. The system is implemented in DSPL, a language initially developed in support of planning research in the domain of mechanical design [1].

The task we are investigating involves one of the functions of Tactical Air Control Centers (TACCs). Their concerns are to assign available resources to the various tasks of an "appointment" order. The output is an Air Tasking Order (ATO), which summarizes the responsibilities of each unit with respect to the day's missions. Each mission planned requires attention to such details as the selection of aircraft type appropriate to the mission, selection of a base from which to fly the mission, coordination with other missions, etc.

Our most recent objectives have been to determine the knowledge a system would require to plan a particular type of mission, the Offensive Counter-Air (OCA) mission. We are interested in the planning process, as well as the ability to explain the reasoning of the planning process. Our selection of the OCA mission in particular arose partly from the availability of the KNOBS system and its knowledge base for tactical planning support in this domain.

2.1.1. The KNOBS System

The KNOBS system was built to address planning tasks which involve the specification of values for a set of pre-established components known to be necessary for the planned activity. Planning offensive counter air missions can be viewed as such a task.

KNOBS sees planning as template instantiation - a process of filling in a number of slots with acceptable values. The order in which the slots are considered is defined in advance by the plan template, and is determined by the expert's domain planning knowledge. Acceptability of slot values is based upon satisfaction of constraints. Constraints are attached to the template (rather than the slots) to reflect the view that "all action is in the interaction of the slots". Constraints are organized as a list of "buckets", ordered to express priority in constraint satisfaction. Each bucket contains an unordered list of constraints. The testing of slot values is accomplished by traversing and checking constraints in the order specified by the priority buckets.

In order to determine acceptable choices for values of slots, KNOBS associates a generator with each slot to enumerate potential values. The generator produces a subset of all possible values of the slot. The generator is derived by "inverting" constraint knowledge pertinent to the slot.

Given the slot ordering, constraints, and generators, KNOBS "plans" as follows: The generator of the first slot is asked for its first candidate, the generator for the second slot is asked for its first candidate, and so on. At each slot filling, all applicable constraints are checked. If any are not satisfied, then the slot generator is asked for another candidate. If another candidate exists, it is tried, and so on until either all slots have accepted values or a generator runs out of candidates. If this happens, KNOBS would back up to the most recently filled slot that was involved in the constraint that failed. KNOBS is successful when all slots are filled and all constraints are satisfied. The basic planning algorithm for KNOBS can thus be described as generate and test with dependency-directed backtracking.

KNOBS was successful in showing the feasibility of AI techniques for certain classes of mission planning. The constraint technique in particular is useful where applicable, but as far as we can determine KNOBS was not intended as a generic approach to planning in general. The methodology for planning based on template instantiation and constraint satisfaction will not typically scale up well, since, as the size of problem space increases, the exhaustive depth-first nature of the search makes the technique computationally infeasible. In addition extensions or adaptations to such a template would be difficult, since most of the planning knowledge is implicit (and thus hidden) in the ordering of template slots and constraints.

These problems arise because the system does not have significant amounts of problem-solving expertise for planning. Any explanation in KNOBS is limited to answers based on a constraint - either a value is bad because it fails a constraint, or it is good because it satisfies a constraint. There is no knowledge of why the system should satisfy a constraint (its functionality) nor why this constraint (vs any other) is being considered now (plan strategy). Similarly, this knowledge is missing for slots - why is this slot necessary, and why is this the best ordering of slots - are questions which cannot be answered hy a KNOBS-like system.

Thus the KNOBS mechanism does not allow for two major types of explanation: neither the planning control strategy, nor functional knowledge of the domain can be justified. These kinds of explanation for a planner are feasible if the planning and functional knowledge is represented with appropriate structures.

It should be added that the designers of KNOBS were also aware of these limitations and are currently building a system called KRS which includes some of the additional functionalities described above.

2.1.2. Class III Design

Our approach to tactical mission planning treats the Air Tasking Order (ATO) as an abstract device to be designed. The planning of the missions or groups of missions that comprise the completed ATO involves a process similar to the process a designer undergoes when faced with a complex device to design. An overview of the design domain will illuminate this analogy. For a more comprehensive description see [1].

The general domain of design is vast. It involves creativity, many problem-solving techniques, and many kinds of knowledge. Goals are often poorly specified, and may change during the course of problem solving. However, a spectrum of design classes can be identified, varying from completely open-ended activity to the most routine, depending on what sorts of knowledge is available prior to the start of problem solving.

What we have called "Class 3 Design" characterizes a form of routine design activity. Complete knowledge of both the components and design plans for the device is assumed to be available prior to the problem solving activity. The problem solving proceeds by using recognition knowledge to select among the previously known sequences of design actions. While the choices at each point may be simple, this does not imply that the design process itself is simple, nor that the components so designed must be simple. It appears that a significant portion of everyday activity of practicing designers falls into this class. In order to explore this class of design problems, the DSPL (Design Structures and Plans Language) system was developed. [1, 2, 3] The routine design task is viewed as decomposable into a hierarchical planning task, where typically each level makes some design commitments, and the design is further refined by the lower level planners. A design problem solver in DSPL consists of a hierarchy of cooperating, conceptual specialists, with each specialist responsible for a particular portion of the design. Specialists higher up in the hierarchy deal with the more general aspects of the device being designed, while specialists lower in the hierarchy design more specific subportions of the device, or address other design subtasks. Any specialist may access a design data-base (mediated by an intelligent data-base assistant). The organization of the specialists and the specific content of each is intended to precisely capture the designer's expertise of the problem domain.

Each specialist in the design hierarchy contains locally the design knowledge necessary to accomplish that portion of the design for which it is responsible. There are several types of knowledge represented in each specialist, three of which are described here. First, explicit design plans in each specialist encode sequences of possible actions to successfully complete the specialist's task. Different design plans within a specialist may encode alternative action sequences, but plans within a particular specialist are always aimed at achieving the specific design goals of that specialist. A second type of nowledge encoded within specialists is encoded in design plan sponsors. Each design plan has an associated sponsor to determine the appropriateness of the plan in the run-time context. The third type of planning knowledge in a specialist is encoded in design plan selectors. The function of the selector knowledge is to examine the run-time judgements of the design plan sponsors and determine which of the design plans within the specialist is most appropriate to the current problem context.

Control in a DSPL system proceeds from the top-most specialist in the design hierarchy to the lowest. Beginning with the top-most specialist, each specialist selects a design plan appropriate to the requirements of the problem and the current state of the solution. The selected plan is executed by performing the design actions specified by the plan. This may include computing and assigning specific values to attrihutes of the device, running constraints to check the progress of the design, or invoking sub-specialists to complete another portion of the design. Thus design plans which refer to a sub-specialist are refined by passing control to that sub-specialist.

The discussion of the control strategies in a DSPL system has thus far only included successful plan execution. However DSPL does include facilities for the handling of various types of plan failures, and for controlling redesign suggested by such failures. The details of these features of the language can be found in [1].

2.1.3. Mission Planning as Class III Design

Our view of tactical mission planning is that it is essentially a class 3 design task. The prohlem can be decomposed into the design of subcomponents of the mission plan. In the device design domain, the design of a device is decomposed into the design of sub-assemblies and their components, etc, where each sub-assembly or component can be designed in a fairly independent fashion. In the tactical mission planning domain the ATO is decomposed into various missions or groups of missions of known types, where each mission or group of missions able to be planned relatively independently of the others, modulo resource contention considerations. In both domains, of course, each of the solutions to the subproblems must be appropriately combined into the solution for the problem which they decompose. Due to the well known limitations of human problem solving capacities, it is apparent that a human problem solver can be successful in such a situation only to the extent that he can also decompose the problem into a manageable number of somewhat independent sub-problems which can be solved separately and combined into a final solution. Using DSPL as a natural mechanism for representing the necessary knowledge, the MPA system closely mirrors these ideas.

Another type of local, declarative knowledge in a DSPL specialist is expressed in the form of constraints. Constraints are used to decide on the suitability of incoming requirements and data, and on the ultimate success of the specialist itself (i.e., the constraints capture knowledge about those things that must be true of the specialists' design before it can be considered to be successfully completed). Other constraints, embedded in the specialist's design plans, are used to check the correctness of intermediate design decisions. The use of such constraints in the MPA system easily captures the kinds of knowledge encoded as constraints in KNOBS, but incorporating the constraints into a rich overall control structure further allows the constraint knowledge to be utilized during problem solving in a sharply focused manner. Analysis of the success or failure of constraints during runtime, generated from the trace of the problem solver's execution, yields explanation capabilities similar to that found in KNOBS, but with the additional context provided by the rich DSPL control structure.

The additional context of the DSPL control structure provides the springboard for a more comprehensive explanation facility. In addition to the necessary ability to examine particular attributes of a mission plan, the control structure provides the ability to examine the problem solving strategies of the planning system. This kind of explanation is not easily extracted from a system which uses template instantiation and constraint satisfaction as its primary mechanisms for problem solving, since problem solving strategies are absent or at best implicitly represented.

2.1.4. The MPA System

The following discussion gives a general description of the planning strategies particular to the MPA system as currently implemented in DSPL.

Several caveats are in order concerning the domain of the MPA system. The MPA system currently only handles the planning of OCA missions, although we believe other missions could be handled in a similar fashion. Our prototype system does not address several minor bookkeeping aspects of mission planning, which although of no theoretical interest, would be necessary to a fully functional mission planner. Such items as assigning radio frequencies to a flight and designating mission call-signs fall into this category. Finally, although the specific military knowledge in the MPA system is adequate for demonstration purposes, it by no means meant to reflect complete or even accurate knowledge of aircraft capabilities. We believe that the knowledge represented is representative of the knowledge utilized by a human mission planner, and that the problem solving exhibited by the system fairly represents the human problem solver's activities.

The prototype MPA system contains six specialists. The topmost specialist, *OCA*, accepts the mission requirements and ultimately produces the final mission plan. The OCA specialist divides its work between two subspecialists, *base* and *aircraft*. The base specialist is responsible for selecting an appropriate base, while the aircraft specialist selects an aircraft type. The aircraft specialist has three subspecialists, one for each of the three aircraft types known to the MPA system. As needed, one of these specialists will select an appropriate configuration for its aircraft type.

Problem solving begins when the OCA specialist is requested to plan a mission. Currently the OCA specialist contains only a single design plan which first requests the base specialist to determine a base and then requests the aircraft specialist to determine (and configure) an appropriate aircraft for the mission. The current base specialist simply selects a base from a list of candidate bases geographically near the target. The aircraft specialist uses considerations of threat types and weather conditions at the target to select an appropriate aircraft for the mission. The aircraft specialist and its three configuration subspecialists represent the most elaborate aspects of domain knowledge in the MPA system.

In the current version of the MPA system, the aircraft specialist is entered with a tentative selection for the base already specified. The target and required probability of destruction are known from the input requirements of the mission. At this point each of the plan sponsors in the aircraft specialist are executed by the DSPL interpreter. The three plan sponsors determine the appropriateness of their respective plans. In this case, each of the three plans determine which of the three aircraft types should be used for the mission. Thus the three plan sponsors determine the appropriateness of using, respectively, F-111s, F-4s, or A-10s for the mission. Plan sponsors may access a global database as necessary in their execution. In the MPA system, such items as target characteristics and weather conditions are requested in determining the appropriateness of a particular aircraft type. After the suitability of all plans in the aircraft specialist has been determined, the DSPL interpreter executes the plan selector in the specialist. The plan selector, given the suitabilities of each of the aircraft types, can then determine which aircraft is most appropriate for the mission. The plan selector returns this information to the specialist, which then causes the selected plan to be executed.

Suppose the mission requirements call for a night raid. The plan sponsors for both the A-10 and F-4 would rule out the possibility of using these aircraft, since (in our domain model) neither of these aircraft have night flying capability. The F-111 plan sponsor, since it is an all-weather fighter with night capabilities, would not be excluded. The plan sponsor for the F-111, based on this and other considerations (range, ability to carry appropriate ordinance, target characteristics, etc) would find the F-111 suitable for the mission. The plan selector in the aircraft specialist, finding that two design plans have ruled out, would select the 'suitable' F-111 design plan, and return this information to the specialist. The specialist proceeds to execute the F-111 design plan, which includes marking the aircraft type in the mission template to 'F-111', and invoking the F-111 configuration specialist which in turn decides an acceptable ordinance load for the F-111 for this mission. Once the configuration of the aircraft is known, the single aircraft probability of destruction in the mission context can be computed. Finally, knowing the mission capabilities of each aircraft, the required number of aircraft can be determined in order to achieve the required probability of destruction, and the required aircraft can be reserved from the proper unit.

The MPA system could be readily extended in several directions. Additional situation knowledge at the OCA level would allow for more robust planning with less backtracking. More complete knowledge of the OCA mission for specifying various aspects of the flight plan, etc. could be added. Also, as previously mentioned, other types of missions could be encoded in hierarchies similar to the OCA hierarchy. The most theoretically interesting addition to the MPA system would be abstractions above the single mission level. Clusters of coordinated missions and even a complete ATO abstraction should be possible within the Class III Design framework. For example, extended range OCA missions requiring coordination with refueling and escort missions should be able to be planned in a straightforward fashion. The single greatest hindrance to such work is the lack of accessibility of experienced domain experts.

2.2. Explanation in the Mission Planning Assistant

Michael C. Tanner, Dean Allemang,
John Josephson, Matt DeJongh

2.2.1. Types of Questions for the Mission Planner

We have generated a broad list of questions that a user may ask of a Mission Planning Assistant (MPA). Here we will give a categorization of those questions. In this preliminary analysis we will be able to sketch techniques for answering questions in some categories. But in others we have little to say at this point.

2.2.1.1. Overall Objectives

There were questions about the objectives of the plan. Some questions of this kind can be answered directly by the mission planner:

> *Question.* What will this plan achieve?
>
> *Answer.* This plan will achieve destruction of target X with probability Y.

Other questions of this type have answers external to the program. For example,

> *Question.* Why are you doing an OCA?

The trivial answer:

> Because you told me to.

is probably not the desired answer. The reasons for planning an OCA come prior to invocation of an OCA planning assistant. On the other hand it is perfectly reasonable for an MPA program to have some built-in definition of the reasons OCAs are done.

2.2.1.2. Justifying Decisions

The most common kind of question asks for justification of some decision made during problem-solving. These seem to come in two kinds:

1. Why did you do X?
2. Why didn't you do Y?

Answering the "Why did you ...?" questions requires finding, or reconstructing, the point in problem-solving where the choice was made, then giving the reasons which support that decision. For example:

> *Question.* Why was an F-4 chosen?
>
> *Answer.* The choices were A-10. F-4, and F-111. A-10 was ruled out. In cases where F-4 and F-111 are available, I prefer to use F-4.

Any such answer may point to further decisions which might be questioned in the same way. In the above example one may ask why A-10 ruled out, and pursue the decision process further.

Answering "Why didn't you ...?" questions is a little harder. There are at least two distinct cases. In one case, the alternative might have explicitly been decided against. In the above example a "Why didn't you choose A-10?" would be answered by "A-10 was ruled out." The other case is that the explicit alternative never came up. Answering the question in this case requires an understanding of the problem-solving strategy and an explanation in those terms.

> *Question.* Why didn't you allocate a KC-135 for this mission?
>
> *Answer.* KC-135 is a tanker, tankers are only used if refueling is necessary, and refueling is not necessary for this mission.

2.2.1.3. Critique

A number of questions were related to plan criticism. A user might want to know where the weak points in the plan are. Or the user might want to know if some alternate plan is any good. We have not worked on questions of this kind but it seems as though critics could operate on the functional representation of the plan and that such criticism would not be closely related to the process of designing the plan.

2.2.1.4. Questions During Problem-Solving

Nearly all of the questions that might be asked after the MPA has produced an answer could also be asked during problem-solving. In addition there are a host of questions about the problem-solving process itself -- why something is being done now, what is left to do, etc.

2.2.1.5. Questions about Function

Many questions are about the function of various parts of the plan. If the plan is viewed as a device, it can be represented using the functional representation of Moorthy and Chandra [1]. This will be discussed in section 2.3. Using this representation it would be possible to answer questions such as "Why are airplanes used?" and "How will the mission proceed?"

2.2.1.6. Questions About the Impact of Data

Often it is useful to know how some fact affected the problem-solving and how the result would be different if that fact changed. This could be related to critique. That is, if a small change in data makes a hig difference then that data could be critical to success of the plan. It might also be useful for making small changes in the final plan, such as forcing aircraft type to be F-111. If it makes little difference, the planner should be able to say so.

2.2.1.7. Questions About Strategy

Answering some questions requires understanding the problem-solving strategy used by the program. Questions of this sort can be directly answered using the explicit encoding of the generic aspects of knowledge and control for the generic task. The framework for this is discussed in the companion paper in this *Proceedings*.

2.2.1.8. Summary Comments

The categorization given above is not meant to be exhaustive or mutually exclusive. In fact, answers to questions of one type, say justification, may include answers of another type, say strategy. For explanation of the planner itself the most important kind given above is justification of run-time decisions. For one thing, such justification would be useful for debugging a planner and is likely to be a part of many other kinds of explanation. In the remainder of this report we will describe how we are implementing justification of a certain kind for the mission planning assistant.

2.2.2. Explanation for the MPA

Our implementation is based on the organizing principle that the agent which makes a decision is responsible for justifying it. The MPA is built in DSPL so the agents which contribute to the final plan are: Specialists, Design Plans, Design Plan Selectors, Design Plan Sponsors, Tasks, Steps, and Constraints. In the present implementation there are some 200 of these agents, though not all of them contribute to any particular plan. All of these agents perform "knowledge-level" tasks (i.e., epistemically significant) so explanation of any one agent's problem-solving decisions can be given in terms of the goals of the agent which uses it, and the function of the agents it uses.

The final answer produced by the MPA can be viewed as a list of attribute-value pairs as in Knobs. That is, a list of the form:

```
Target = Berlin
Aircraft Type = F-111
Number Aircraft = 6
...
```

We have decided to concentrate on questions of the form. "How was it decided?" which can be asked of the value of any attribute. For example, selecting F-111 in the above list would initiate a dialog on the question of how MPA decided to use F-111 as the value of **Aircraft Type**. More particularly, an explanation window would appear containing the answer to "How was it decided?" produced by the agent which actually set the value of **Aircraft Type** to F-111. In this window certain other things would be selectable. Selecting any of them will produce another window with a similar explanation for the proper agent. In this way the user will be able to pose follow-up questions by using the mouse to steer through the decision dependencies.

To support "How was it decided?" explanations we determined three basic questions which all agents must be able to answer:

1. "Give me the bottom line: what did you do?" This question would be answered with a one-sentence summary of the result of the agent's action.

2. "What is your purpose?" This question would be posed by sub-agents who want to have knowledge of the context they are operating in, and should be answered by a short description.

3. "How did you do it?" This question would be answered by displaying a window with a complete explanation of the context of the agent's activation followed by a functional description of its action. The agent may have to ask its sub-agents Q1 and its super-agent Q2.

Then, in general, the explanation for "How was it decided?" is the answer to question 3 above. The answer to q3 is a combination of the answer to q2 for the calling agent and q1 for all sub-agents. So an explanation window contains:

```
In the context of <answer to Q2 for callin
   we did the following:
      <answer to Q1 from subagent1>
      <answer to Q1 from subagent2>
      <answer to Q1 from subagent3>
                 . . .
```

Below we show detailed examples of all the agent types and the explanations they can produce.

Our work to this point is about generating explanation fragments and does not address other issues of explanation such as summarization, user modeling, or human factors.

In figure 1 is a sample of the output for MPA on a particular problem. It is simply a list of attributes of OCA missions and the values that the MPA determines for them during problem solving. The user begins to get explanations by selecting one of the values and asking[1] how MPA decided on that value.

Target	**BrandenburgSAM**
PD	**.8704**
AircraftType	**F-4**
NumberA/C	**4**
Unit	**113TFW**
Airbase	**Wiesbaden**
Configuration	**B2**

Figure 1: Example of a particular OCA mission

2.2.2.1. Step Explanation

The values are actually set by a DSPL Step, so the first explanation a user gets comes from a Step. Suppose the value of NumberA/C, **4**, was chosen. A slightly simplified version of the code for the step which actually set this value is given in figure 2. This step sets some local variables by looking things up in the data base. E.g., the local variable *configuration* is set to the result of asking the data base what configuration is being used, *(KB-FETCH CONFIGURATION)*. The *KB-STORE* tells the data base to set the attribute *AIRCRAFT-NUMBER* to the value returned by the function *num-a/c*, which depends on the local variables. *REPLY* indicates that what follows is the main function of the step and sets DSPL up to handle failures if something should go wrong.

```
(STEP setNumberA/C
  (SETQ configuration (KB-FETCH CONFIGURATION))
  (SETQ requiredPD (KB-FETCH REQ-PD))
  (SETQ targetType (KB-FETCH TARGETTYPE))
  REPLY
    (KB-STORE AIRCRAFT-NUMBER (num-a/c configuration
                                       requiredPD
                                       targetType)))
```

Figure 2: DSPL code for a step

Figure 3 shows the explanation of the step given in figure 2. The context, shown in italics, is retrieved from the task which invoked the step. The values set for the local variables are remembered by the step at run-time as is the value returned by the Lisp function *num-a/c*. DSPL then fits these pieces into the general framework for explaining steps.

[1] Figuratively, since the system can only answer this one question at present.

The context of *working out the details of configuration B2* determined that:

- configuration was **B2**
- requiredPD was **.65**
- targetType was **SA-6**

So, 4 was an appropriate choice for AIRCRAFT-NUMBER.

Figure 3: Explanation for a step

In figure 4 is a general description of steps showing the relationship between the code and the explanation which can be produced from it. The purpose of the calling Task is found by asking, "What is your purpose?" of the calling Task (see section 2.2.2). The values of local variables, and the value given to the attribute, are remembered by the step at run-time and retrieved for explanation.

2.2.2.2. Task Explanation

After looking at the explanation for a step, the only further explanation is for the task which invoked it. This is obtained by selecting the context given in the step explanation.

Figure 5 gives the DSPL code for a Task. A task is simply a sequence of steps, with constraint checks possible. If the user had been looking at the explanation for the **base-assign** step and pursued its context, the explanation, which would come from the task shown in figure 5, would be that shown in figure 6. As with steps, the context is obtained from the calling agent, in this case a Plan. The rest of the explanation is obtained from the steps which make up the task.

Form:

```
(STEP <stepName>
      (SETQ <localVar1> <val1>)
            ...
      (SETQ <localVarN> <valN>)
      REPLY
        (KB-STORE <attribute> <attributeVal>))
```

Explanation:

The context of <purpose of containing task> determined that:

- <localVar1> was <val1>
- ...
- <localVarN> was <valN>

So, <attributeVal> was an appropriate choice for <attribute>.

Figure 4: Template for Steps and their Explanation

```
(TASK squadron
  (STEP squadron)
  (STEP base-assign)
  (STEP get-range))
```

Figure 5: DSPL code for a Task

Figure 7 gives a general description of Tasks showing the relationship between the code and the explanation produced from it. The purpose of the containing Plan is found by asking, "What is your purpose?" of the plan. What a particular step did, or its purpose, is found by asking, "What did you do?" or "What is your purpose?" as appropriate, of the step. DSPL fits these answers into a general framework for explaining tasks.

In the context of *considering the feasibility of an F-4 for the mission*, I did the following step:

- selection of 113TFW as squadron for the mission

I was in the process of:

- selecting a base for the mission

I had yet to do the following step:

- determine the range for the mission

Figure 6: Explanation for a Task, entered from **base-assign** step

2.2.2.3. Plan Explanation

From a task the user might select either explanation of the various steps in the tasks or of the task's containing Plan. The syntax of plans and their explanation is very similar to that of tasks. The exception is that Plans can invoke design specialists, as shown by the *DESIGN* statement in figure 8. Figure 9 shows the explanation given by this plan and figure 10 gives a general description of plans and their explanation.

2.2.2.4. Specialist Explanation

As with Tasks, a user can choose to pursue explanation of a Plan's context or of its sub-agents. The sub-agents are Tasks, which have been described. The context is given by a design specialist. The logic of design specialists is implicit, that is, defined by what a design specialist is. The Specialists' job is to choose a design plan and execute it. It chooses the plan by invoking its design plan selector. The explanation for a specialist is given in figure 11 and the general form of specialist explanation is in figure 12. The context of a specialist is given by the plan which invoked it and it knows its own purpose. The purpose of the plan it selected is obtained by asking that plan.

Form:

```
(TASK <taskName>
  (STEP 1)
   ...
  (STEP i)
   ...
  (STEP n))
```

Explanation, entered from STEP i:

In the context of <purpose of containing plan> we did:

- <what STEP 1 did>
- ...
- <what STEP i-1 did>

we were doing:

- <purpose of STEP i>

and were about to do:

- <purpose of STEP i+1>
- ...
- <purpose of STEP n>

Figure 7: Template for Tasks and their Explanatio

```
(PLAN F-4
  (TASK assignF-4)
  (TASK squadron)
  (DESIGN F-4Configuration))
```

Figure 8: DSPL code for a Design Plan

In the context of *selecting an appropriate aircraft for the mission* I was in the process of:

- assigning an F-4 for the mission

I had yet to do the following steps:

- find an appropriate squadron for the mission
- choose a configuration for the F-4 in this mission

Figure 9: Explanation for a Design Plan, entered from the **assignF-4** task

2.2.2.5. Selector Explanation

From the specialist a user could pursue the context of the specialist, the plan that called it, or the specialist's selector. Figure 13 shows the DSPL code for a Selector. The typical selector simply chooses the best perfect plan, if there are any, or the best suitable plan if there are no perfect ones. The selector shown in figure 13, however, encodes the additional knowledge that if certain plans are available they ought to be chosen.

The explanation of the selector in figure 13 is given in figure 14. Here the context comes from the specialist. The rest of the explanation comes from remembering the values of the predicates. The value returned by the selector, in this case, depends on both the fact that A-10 is not a perfect plan and that F-4 is.

The general form of selectors and their explanation is shown in figure 15. A selector is essentially an IF-THEN-ELSE statement so it must be able to remember, or reconstruct, the values of the IF part to explain which branch was taken.

Form:

```
(PLAN <planName>
  (TASK 1)
   ...
  (TASK i)
   ...
  (TASK n))
```

Explanation, entered from TASK i:

In the context of <purpose of containing specialist> we did:

- <what TASK 1 did>
- ...
- <what TASK i-1 did>

we were doing:

- <purpose of TASK i>

and were about to do:

- <purpose of TASK i+1>
- ...
- <purpose of TASK n>

Figure 10: Template for Design Plans and their Explanation

In the context of *using the old-reliable plan to plan the mission* I was performing my task of *selecting an appropriate aircraft for the mission*. I had:

- decided to consider F-4 as aircraft for the mission

I was in the process of:

- considering the feasibility of an F-4 for the mission

Figure 11: Explanation for a Design Specialist, entered from plan **F-4**

Explanation, entered from PLAN 1:

In the context of <purpose of containing plan> I was performing my task of <purpose of self>. I had:

- selected PLAN 1

I was in the process of:

- <purpose of PLAN 1>

Figure 12: Template for Explanation of Design Specialists

```
(SELECTOR aircraftSelector
  (IF (MEMBER A-10 PERFECT-PLANS) THEN
      ELSEIF (MEMBER F-4 PERFECT-PLANS
      ELSEIF (MEMBER F-111 PERFECT-PLA
```

Figure 13: DSPL code for a Design Plan Selector

The context of *selecting an appropriate aircraft for the mission* determined that:

- Since A-10 is not one of PERFECT-PLANS,
- I chose plan F-4 because F-4 is one of PERFECT-PLANS.

Figure 14: Explanation for a Design Plan Selector

Form:

```
(SELECTOR <selectorName>
  (IF <there are perfect plans>
      THEN <choose the best perfect plan>
   ELSEIF <PLAN 1 is suitable>
      THEN <choose PLAN 1>
   ELSEIF <there are suitable plans>
      THEN <choose the best suitable plan>))
```

Explanation:

The context of <purpose of containing specialist> determined that:

- Since there were no perfect plans.
- I chose PLAN 1 because PLAN 1 was suitable.

Figure 15: Template for Design Plan Selectors and their Explanation

2.2.2.6. Sponsor Explanation

From a selector the user could get explanation from any of the plan sponsors which it uses. A sponsor matches characteristics of the plan to information about the problem at hand and produces a measure of how useful the plan will be on a scale of: Ruled-Out, Unsuitable, Suitable, and Perfect. The code for a sponsor is given in figure 16. It first sets some local variables by looking them up in the data base (using *KB-FETCH* as discussed with steps). It then uses the *TABLE* construct, which is essentially a group of rules which all depend on predicates of the same values. For example, the table setting the variable *conditions* contains three rules which depend on the values returned by the functions *night* and *weather*. The first rule requires *night* to return *F* and *weather* to return *FULL*. If the predicates are true, then *conditions* will be *UNSUITABLE*. The symbol '?' in the tables represents a predicate which is always true. The table is finished when one rule matches. *REPLY* tells DSPL that what follows is the main function of the sponsor.

The explanation for this sponsor is given in figure 17. Values for the local variables are given, those fetched from the database are not justified while those determined by tables are given justification. The final *REPLY* is used to determine the actual decision made by the sponsor.

```
(SPONSOR A-10
  (SETQ target (KB-FETCH TARGET))
  (SETQ timeOverTarget (KB-FETCH TIMEOVERTARGET))
  (SETQ threat
        (TABLE (airborne)(AAA)(SAM)
           (IF     T        ?    ?   THEN UNSUITABLE)
           (IF     ?        T    ?   THEN UNSUITABLE)
           (IF     ?        ?    T   THEN UNSUITABLE)
           (IF     ?        ?    ?   THEN PERFECT)))
  (SETQ conditions
        (TABLE (night)(weather)
           (IF    F      FULL   THEN UNSUITABLE)
           (IF    F    PARTIAL  THEN SUITABLE)
           (IF    ?       ?     THEN PERFECT)))
  REPLY
        (TABLE conditions   threat
           (IF UNSUITABLE     ?       THEN RULE-OUT)
           (IF     ?      UNSUITABLE THEN RULE-OUT)
           (IF  SUITABLE      ?       THEN SUITABLE)
           (IF     ?          ?       THEN PERFECT))
```

Figure 16: DSPL code for a Design Plan Sponsor

A general picture of sponsors and their explanation is given in figure 18. Generally, the values set for local variables, the values of columns and predicates from tables, are stored away at run-time to be used in explanation. Explanation then involves parsing the sponsor's code and fitting these values into the explanation template as needed.

2.3. Understanding an OCA Mission Plan

Anne Keuneke, John Josephson

Knowledge-based systems *use* knowledge to arrive at solutions. If a system will be used to provide consultation or advise, it will need to *explain* its knowledge and problem solving in order to be acceptable and useful. In the task of planning, for instance, the planner must have access to its knowledge of problem solving strategies if it wishes to provide explanation of its design decisions. If the system hopes to provide an understanding of how the designed plan will work, it must have this knowledge, too, represented in a meaningful and accessible fashion.

To illustrate the different types of explanation capabilities arising from knowledge structures within a system, consider the task of OCA mission planning. The planning task accomplished by the MPA (Mission Planning Assistant) system at OSU, involves specification for a set of *pre-established* components. That is, the planner *knows* the mission needs a certain type of component - its job is to make a concrete commitment as to which specific component of that type would be best. The planner requires only a limited knowledge of these components in order to make such decisions. Its understanding of the resultant mission plan is thus restricted.

The context of *selecting an aircraft to consider for the mission* determined that:

- target is **BrandenburgSAM**
- timeOverTarget is **1300**
- threat is **UNSUITABLE** because:
  - SAM is TRUE
- conditions are **PERFECT** because:
  - weather is not FULL
  - weather is not PARTIAL

I determined the value of plan A-10 to be **RULE-OUT** because:

- threat is UNSUITABLE.

Figure 17: Explanation for a Design Plan Sponsor

Form:

```
(SPONSOR <sponsorName>
  (PLAN 1)
  (SETQ <var1> <val1>)
     ...
  (SETQ <vari>
        (TABLE <col 1>  <col 2>
           (IF <pred 1> <pred 2> THEN <val x>)
           (IF <pred 3>    ?     THEN <val y>)))
     ...
  REPLY
        (TABLE <col 3>  <col 4>
           (IF <pred 4>    ?     THEN <rating 1>)
           (IF <pred 5> <pred 6> THEN <rating 2>)))
```

Explanation:

The context of <purpose of containing selector> determined that:

- <var 1> is <val 1>
- ...
- <var i> is <val y> because:
  - <pred 1> is not true of <col 1>
  - <pred 3> is true of <col 1>

I determined that the value of PLAN 1 to be <rating 1> because:

- <pred 4> is true of <col 3>.

Figure 18: Template for Design Plan Sponsors and their Explanation

For example, suppose a user of the mission planner asks the question, "Why was an F-15 used?" Depending on the intentions of the inquirer, the question could be answered in different ways. For a *particular* mission, the question might be addressed directly by the mission planner. Here, the inquiry is interpreted as, "Why did you use an F-15 *instead of* any other aircraft for this mission?" Explanation would indicate what makes the F-15 appropriate (speed, weather compatible, etc.). Since this is the specific information the system used in making its decision, the planner should be able to explain it.

In the above, interpretation of the question was, "Why choose an <u>F-15</u>?". An alternate interpretation could be, "Why is the F-15 <u>used</u> in the mission plan?". A good response here might be, "The F-15 is an aircraft. Aircraft are used in OCA's because they have the ability to fly and to deliver the ordinance. These functions are used to get to the target location and to destroy the target - the primary goal of an OCA mission." This explanation requires a deeper understanding of the domain than the planner has readily available within its compiled planning knowledge. Here we need a structure to represent distinctly *how the plan works.*

[2] Moorthy, V.S. and Chandrasekaran, B., "A Representation for the Functioning of Devices that Supports Compilation of Expert Problem Solving Structures", Proceedings of MEDCOMP'83, IEEE Computer Society, September, 1983

To represent this understanding, we propose use of a knowledge structure based upon the Functional Representation of Devices as designed by Moorthy and Chandrasekaran.[2] A device is any structure (concrete or abstract) which serves a purpose. Thus, a plan can be viewed as an abstract device in that it has components which fit together in such a way to achieve a desired goal. We hope, in this paper, to illustrate how a functional representation for a plan can serve as a knowledge structure from which a more complete understanding of the specific planning domain can be derived.

The Functional Representation: An Overview

The first concept is that an agent's understanding of how a device works is organized as a representation that shows how an intended function is accomplished as a series of behavioral states of the device. The device, itself, is represented in various levels. The topmost level describes the functioning of the device in terms of the roles of its components. The next level describes the functioning of these components using the roles of their subcomponents, and so on. At each level of a device's representation there may be five significant aspects to an agent's knowledge of the functioning of the device:

-STRUCTURE: specifies the components of a device and the relations between them.

-FUNCTION: specifies WHAT is the result or goal of an activity of a device or component.

-BEHAVIOR: specifies HOW, given a stimulus, the result is accomplished.

-GENERIC KNOWLEDGE: pointers to general knowledge that shows how key states occur.

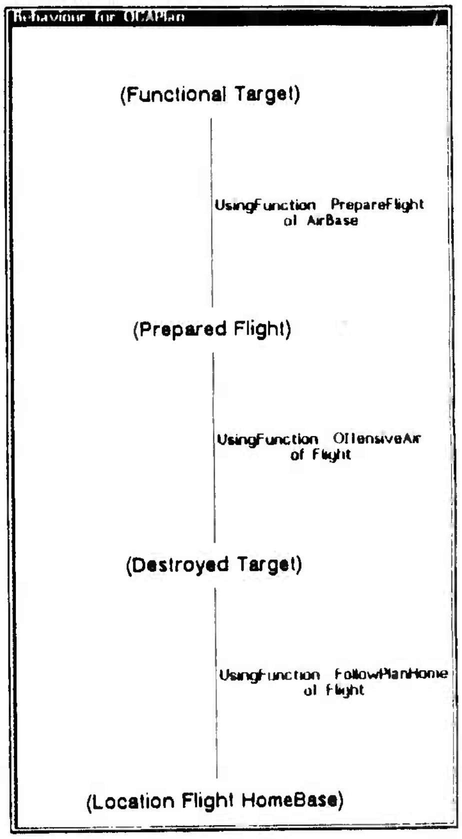
-ASSUMPTIONS: under which a behavior is accomplished.

The *functional specification* of the "abstract device" OCAMission is illustrated below by describing the main function of an OCA - to destroy a target.

FUNCTION: DestroyTarget:
TOMAKE: (Destroyed Target)
IF: (Functional Target)
PROVIDED: (Functional Flight)
BY: OCAplan

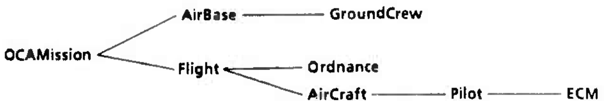
The description indicates that the plan, OCAMission, has a function called DestroyTarget. This function is used if a target is operational(functional). When this function is used, the target will be destroyed by a behavior called OCAplan. This behavior should succeed in accomplishing the goal of target destruction provided the flight is operational throughout the behavior.

The *behavioral specification* of a device describes the manner in which a function is accomplished by using the functions of components, generic knowledge, and sub-behaviors. The behavior for an OCA plan is described by a chain of events caused by the specified actions:


Behaviour for OCAPlan
(Functional Target)
UsingFunction PrepareFlight
of AirBase
(Prepared Flight)
UsingFunction OffensiveAir
of Flight
(Destroyed Target)
UsingFunction FollowPlanHome
of Flight
(Location Flight HomeBase)

The structure is meant to represent the temporal sequence (from top to bottom) of states which occur as a result of actions taken. The diagram thus indicates that the OCAPlan's behavior begins when a Target is in a Functional state. Here an OCA plan will use the function PrepareFlight of the component AirBase to make the Flight Prepared. Upon achieving this state, the plan uses the component Flight since it has the functionality (OffensiveAir) to Destroy the Target (and so on).

The structure of the OCAMission is defined by its components and relations:



Links in the chain indicate subcomponents in the sense that the first component uses the next in order to achieve its goals. The OCAMission *uses* the component AirBase to prepare the aircraft and the component Flight to get to the target and destroy it.

Some important characteristics which make this representation useful for the design and repair of devices/plans include:
1) A component is specified independent of the representation of the device which contains it. More specifically, the specification of a component does not refer to the role of the component in the composite. If replacements are necessary, this property allows for the determination of allowable substitutions by simply comparing functional capabilities of current components with alternatives.

2) Not the behavior specifications of components, but only the names of the functions are carried over to a higher level. This property is important if an agent needs to replace a malfunctioning component by a functionally equivalent but behaviorally different one. (Ie. It is not *how* the function is achieved that is imperative, but *what* is achieved.)

Since much of planning involves adaptations of already established plans, these traits which allow for such adaptations for components and behaviors are valuable.

Current Research: Enhancements to the Representation

Enhancements to the Functional Representation are being made both to further the above capabilities for adaptation and for richer understanding and explanation capabilities. New primitives established for representations include:

1. A means of distinguishing between the *definition* of a function and events which *trigger* the use of the function in the specified device.
   Example: An aircraft has the function "Fly" which is used to change location. ie. IF: (Location Aircraft x) the function Fly is used TOMAKE: (Location Aircraft y) For the OCA mission *use* of this function is triggered when the current leg of the Flightplan indicates a change of location from x to y.
   The distinction between triggers and the "definition if" is useful for replacement considerations. Functions must match definitions to be equivalent. Triggers are relative to the device in which the component is being used. If a component is replaced, adaptations to a plan may be needed to change a triggering mechanism.
   To determine if a helicopter *could* be used instead of an aircraft within an OCA plan, a top-level response involves checking the functionalities of the two devices. Here the trigger is transferable to the other device (the FlightPlan could just as easily specify helicopters as airplanes). If the functions of the devices are equivalent, the question might be sent to the plan designer of this level to determine why helicopters were not chosen as the device. Notice that functions of devices should not change but events that trigger their use may.

2. Explicit distinction of device's "secondary functions".
   These are functions which are present *in support of* another main function. Specification of such functions is needed for proper explanation and for information when considering replacement of components. Three types have been determined:

   a. Subfunctions:
      - functions a device possesses simply as a means to establish preconditions for a primary function. (e.g. takeoff for fly in aircraft)
      - functions a device possesses to support a provided clause (assumptions) on behaviors for a primary function. (e.g. windshield-wiper/car , ECM/OCA)

   b. Secondary functions:
      With respect to the desired use of the given device, these are extraneous functions. Consider a kerosene lamp one hundred years ago. It's functionality then was to give light. Today its use is often decorative (the rustic look). When purchased for this purpose, the functionality of producing light is rarely used. Notice that secondary functions could be primary functions depending on the device designer's and/or the user's purposes. (OCA missions have none of these.)

   c. Other design considerations:
      - goals because of situation context of the device (e.g. The component Flight of device OCAMission has the function FollowPlanHome. The main goal of an OCA is to destroy a target. With respect to the device OCAMission, explanation of why Followplanhome is needed involves "external" considerations. It is present because in the process of destroying the target we also hope to protect our people and resources)

3. Specification of rationales for links within behaviors.
The designer of a device specifies a function or behavior for a purpose. Explicit representation of this rationale assists explanation. This feature is necessary in planning where states may be achieved to establish conditions for future use. Linear sequencing of events does not always provide a full understanding of the behavior. (e.g. an aircraft is loaded with the ordinance long before the device OCA is ready to use the ordinance)

4. Availability of conditions on links in behaviors.
Functions can be achieved through more than one behavior. Use of a specific behavior may be contingent on specific conditions. (e.g. an aircraft *may* refuel *while flying* if it is at a refueling service and it requires fuel)

To summarize, the changes to the original specification of the functional representation language as defined by Moorthy and Chandrasekaran that have been made involve the specifications of functions and behavior links. The primitives of these objects are now as follows:

FUNCTION: <name>
IF:

TOMAKE:
BY:
PROVIDED:
TRIGGERED WHEN:
SubFunctionOf: ?
ExternalConsideration: ?

Behavior Links:
CONDITION:
RATIONALE:
LINKTYPE: (one of: as per, using function, by behavior)
SPECIFICATIONS contingent on linktype choice: identification of (knowledge/ function/ behavior)

Explanation of Plans

Understanding of an OCA plan can now be illustrated through the explanation capabilities inherent in its functional representation. The representation is capable of answering questions about its devices, functions, and behaviors. Example answers to questions will be given in the context of a top-level device of OCAMission. Explanation responses are built using access to the proper functional primitives.

I. Devices

1. QUESTION: "Why is this device needed?"
ANSWER: Device ______ is used because it has the functional capabilities to ______, ______, and ______.
EXAMPLE: "The device Flight is used because it has the following functional capabilities:
To achieve offensive air missions
To reach the target
To return to the homebase after a mission"

2. QUESTION: "What subcomponents does this device require?"
ANSWER: The structure of the device in the form of a hierarchy of components is given (as was illustrated earlier by the structure of an OCAMission).

3. QUESTION: "What are the secondary functions of this device and their roles?"
ANSWER: The device ______ has the functionality ______ because it supports the primary function ______.
ANSWER: The device ______ has the functionality ______ present because it has a design consideration for __________
EXAMPLE: "The device AirCraft has the functionality TakeOff because it supports the primary function Fly."
EXAMPLE: "The device OCAMission has the functionality MaintainResources present because it has a design consideration for preservation of the crew and aircraft."

II. Functions

1. QUESTION: "Why is this function needed?"
ANSWER: This function is needed to ensure that ______. Here, secondary functions specify that they are needed for functionalities________.
EXAMPLE: "The function Protection of ECM is needed for its capabilities to protect the aircraft and crew, to ensure that the Aircraft is not threatened, and to support conditions for the function DestroyTarget."[3]

[3] Further inquiry shows that DestroyTarget has a PROVIDED of the flight being functional.

2. QUESTION: "What does this function do?"
ANSWER: The function _______ is accomplished by behavior ______ to ensure that ______. The behavior can be used if ________. It is triggered when ______.
EXAMPLE: "The function Offensive Air is accomplished by behavior OffensiveAirTactics to ensure that the target is destroyed. The behavior can be used if the target is functional, the flight is loaded, and the constraint FuelSufficientForPlan is satisfied by FlightPlan. It is triggered when the time of departure of the OCAMission is CurrentTime."

3. QUESTION: "How is this function achieved?"
ANSWER: The behavior for the function is shown with the use of its "behavior browser" as previously shown for the OCAPlan.

4. QUESTION: "Where is this function used?"
ANSWER: Functions/Behaviors of the mission are inspected to see where the function is used.
EXAMPLE: "The function fly is used in the behavior GetThere of function FollowPlanToTarget and in the behavior GetBack of function FollowPlanHome."

III. Behaviors

QUESTION: "Why is this action performed?"
ANSWER: Either a specific rationale for the action is obtained from the link or a default answer of the following state in the behavior is specified.
EXAMPLE: "The function LoadOrdnance is used in OffensiveAirTactics because it ensures that the flight is loaded which is needed for the primary goal to destroy the target."

Potentials and Future Research

Capabilities of the functional representation as a structure of understanding are not limited to explanation of devices, functions, and behaviors. A diagnostic compiler which takes as input a functional representation of a device, and outputs an expert system for diagnosis of problems of the device is already implemented. If one views debugging of a plan as troubleshooting in an abstract device, such a diagnostic system is useful for reasoning about why a plan will or will not work.

Similarly, the representation may be useful for simulation of plans. The planner establishing the use of specific devices may wish to use this potential to check the feasibility of his planning decisions. Necessary provisions to the functional representation for such use would include:

1. the addition of a clause for behaviors which indicates side effects
In simulating a mission plan, the system would need to know that using behavior Cruise (from function Fly of AirCraft) to change the location of the Aircraft will also cause a depletion of the fuel in the aircraft.

2. a concept of time usage must be available
The functional representation is illustrated with discrete state changes. Behaviors which cause these state changes may vary in the length of time required. Some actions appear instantaneous (Ordinance delivered to target -- target destroyed), while others may have intermediate, unspecified states. This would also require a more specific definition of what constitutes a "state". Another time consideration involves the representation of behaviors which occur in parallel or in synchronous motion.

With the above capabilities of debugging and simulation in mind, obviously a planner has opportunity to use the functional representation in making its decisions. Further research is needed to determine how much assistance the representation can donate towards the building of a planner. Much of the planner's domain knowledge can be derived from this representation which specifies how the domain works. How does the planner choose what information to use (and when) for making his decisions regarding the best choice? What influences does the deeper model have on the knowledge used by the planner?

Other areas of concern include the creation of plans and/or adaptations to existing plans. Using a functional representation, components of plans are specified independent of the representation of the plan which contains them. This makes it feasible to *create* and *modify* plans given the goals desired and the functional specifications of available components.

There are many unexplored aspects to the task of planning. It is apparent that the functional representation is a useful structure for approaching the problem - for an understanding of the problem solving, as a representation of knowledge in the domain, and as an abstract knowledge structure to use in construction of plans.

3. DSPL and HYPER: Two High-Level Tools

In the companion paper, we describe a number of generic tasks around which we propose that problem solving, knowledge organization, and explanation be organized. Two of those are: class 3 design, and hypothesis matching. In our description of the Mission Planning Assistant, we indicated that a language called DSPL was used to encode the system as well as to generate the explanations. DSPL was described in some detail as part of explaining the construction of the MPA system.

In this section we present a manual for DSPL, and also a description for another tool that we call HYPER. These tools along with other tools that are under construction in our laboratory will provide a powerful set of high level tools for the construction of a variety of knowledge-based systems.

3.1. The DSPL Manual

David Herman and David C. Brown

DSPL (Design Structures and Plans Language) is a language developed for implementing expert systems which perform a kind of design problem solving. This document covers various details of loading and interacting with DSPL on a Xerox 1108 Lisp machine (a.k.a. Dandelion), running at least the Buttress release of LOOPS with at least the Koto release of INTERLISP-D. It is assumed that the reader is familiar with both LOOPS and INTERLISP-D on a Dandelion, as well as an exposure to the theoretical motivations underlying the DSPL language.

3.2. Loading DSPL

DSPL may be loaded either by installing the DSPL sysout, or by loading the DSPL system onto an existing sysout. In order to load DSPL on an existing sysout, both Interlisp and LOOPS must be already loaded. A fresh version of LOOPS is recommended, although not necessary.

To load DSPL, insert the floppy with the INTERLISP-D DSPL files on it and type in:

```
LOAD({FLOPPY}LOADDSPL)
```

Before any files are loaded, you will be asked 2 questions. The first question asks if the DSPL source should be loaded, and the second question asks if the AIR-CYL expert system should be loaded. The AIR-CYL expert system is written in DSPL and is used to illustrate the use of DSPL throughout this paper. If you are exploring DSPL for the first time, you should answer "n" to the first question and "y" to the second. When DSPL has completed loading, the DSPL icon will appear on the screen.

3.2.1. The DSPL Icon

The DSPL icon facilitates access to the top level DSPL functions through the use of the mouse. The icon allows new DSPL problem solvers to be created and existing problem solvers to be loaded from a file and browsed. It also allows certain modes of operation of the DSPL interpreter to be modified as desired. The specific commands available are briefly described below.

Left Button Commands

<Move> This command is identical to the Move command for the LOOPS icon class. It allows the icon to be placed at an arbitrary location on the screen under mouse control.

Middle Button Commands

<Create> Creates a new instance of a DSPL problem solver. The name of the new problem solver and its top-most specialist is prompted for in the PROMPTWINDOW. A Specialist Browser (described in section 3) is also created for the new problem solver. This browser organizes all access to and modification of the problem solver as it is being developed.

<Browse> Brings up a Specialist Browser for an existing problem solver. (See section 3.) The name of the problem solver is prompted for in the PROMPTWINDOW.

<Load> Causes an existing DSPL problem solver to be loaded from disk or floppy. Several variants are available in a submenu, depending on the type of file to be loaded.

<Load> This is the standard mechanism for loading an existing problem solver which was previously saved to disk or floppy. The name of the file must be typed into the PROMPTWINDOW when requested. All DSPL source code and function definition are loaded directly from the file specified. This submenu command is identical to the main menu command.

<Load Source> This version of Load reads the input file as a list of DSPL source code statements. The name of the file, the problem solver, and the top-most specialist must be entered into the PROMPTWINDOW, as requested. Each statement is parsed by the DSPL system and added to the specified problem solver. The input file may have been created either by the DSPL system (see the Save source only command, section 3), or by a text editor on another host computer.

<Set modes> This command controls certain aspects of the behavior of the DSPL system. A submenu of options is available.

<Set parser modes> Controls the amount of detail provided in the messages from the DSPL system when parsing pieces of DSPL source code. The default set-

ting prints a brief message each time a DSPL agent is successfully parsed, and an error message when a parse fails.

<Set demo fonts> Changes the display fonts in the INTERLISP-D environment to fonts which are sized appropriate for demonstrations. A submenu command allows the fonts to be returned to the standard sizes.

<Help> Provides a brief introduction to the use of the DSPL system as implemented in INTERLISP-D.

Right Button Commands

<Move> Same as the left button command.

<Close> This command is identical to the Close command for the LOOPS icon class. Close removes the DSPL icon from the screen.

All of the middle button DSPL icon functions can also be invoked under program control by sending the appropriate message to the DSPL icon instance. The pointer to the instance is maintained in the global variable DSPL.Icon, which is set when the DSPL interpreter is initially loaded. If no arguments are supplied with the message then they will be prompted for, just as if the DSPL icon was buttoned with the mouse. Alternately, the necessary arguments may be supplied with the message. The order of the arguments matches the order they are prompted for when the DSPL icon is used interactively.

3.2.2. The DSPL Browsers

Several types of browsers are used to organize and access problem solvers built using DSPL.

There are four different agent browsers in DSPL, each of which display a particular grouping of DSPL agents, as described later in this section. All of the agent browsers, however, share the common ability to create and manipulate the various DSPL constructs of the language. The following describes the operations common to all of the agent browsers.

Left Button Commands

The left button commands are displayed in a pop-up menu when left mouse button is pressed and held while the cursor is pointing to an agent label in any of the agent browsers. Again, the command selected will act on the agent which the cursor was pointing at when the mouse button was pressed. The following commands are available:

<PP> Pretty prints the DSPL agent definition for the selected agent in the PPdefault window.

<Inspect> (this agent) Brings up an INTERLISP-D inspector window on the instance of the selected agent. Two options are available.

<Inspect this agent> Identical to the above command.

<Inspect component> Similar to the Inspect command, but the selection is made from a submenu of agents which are components of the selected agent.

<Browse specialist> If the selected agent is a DSPL specialist, this command will bring up a Specialist Component Browser showing the internal structure of the specialist. If the selected agent is not a specialist, this command has no effect.

<Browse plan> If the selected agent is a DSPL plan, this command will bring up a Plan Component Browser showing the internal structure of the plan. If the selected agent is a specialist, this command will bring up a submenu of all the plans contained in the specialist. Selecting one of the plans in the submenu will cause that plan to be browsed. If the selected agent is not a plan or specialist this command has no effect.

Middle Button Commands

The middle button commands are displayed in a pop-up menu when the middle mouse button is pressed while the cursor is pointing to an agent label in any of the agent browsers. Again, the command selected will act on the agent which the cursor was pointing at when the mouse button was pressed. The following commands are available.

<Edit> Invokes Dedit on the DSPL source for the selected agent. The source may then be modified as desired. When Dedit is exited, the DSPL system parses the edited source and compiles a new agent instance, which is consequently installed into the problem solver. If any errors are encountered by the system during processing, the source may be re-edited, or optionally saved for later consideration. (See the Edit Bad Source option below.) If no changes are made to the source code, the parser is not invoked and no change is made to the problem solver. Several submenu options are available.

<Edit this agent> Identical to the above command.

<Edit object code> Similar to the edit command, but invokes Dedit on the INTERLISP-D code generated by the DSPL parser for the selected agent.

<Edit component> Similar to the edit command, but the selection is made from a submenu of agents which are components of the selected agent.

<Add undefined agent> Allows for the definition of new DSPL agents from a list of agents currently referenced but undefined in the problem solver. The

command causes a submenu of all DSPL agent types to be presented. Selection of an agent type causes a menu of the undefined agents of that type to be presented. Selection of an agent causes Dedit to be invoked on a source code template of that type. From here, the Add command works similar to Edit.

<Delete> Deletes the selected agent from the problem solver.

<DeleteFromBrowser> Removes the selected agent and its subagents from the browser. This does not affect the structure of the hierarchy, only what is displayed. This command effects are undone by the RemoveFromBadList command in the Title Menu Commands.

Title Menu Commands

The title menu commands are displayed in a pop-up menu when either the left or middle mouse button is pressed and the cursor is pointing to the title bar within an agent browser. The following commands are available.

<Recompute> This command is nearly identical to the Recompute command for LOOPS class browsers. The only difference is that the submenu item ChangeFontSize is replaced with SelectFont. This new item allows a greater selection of fonts for the browser. Recompute is called automatically when agents are added, deleted or edited via other browser commands.

<SaveValue> Same as SaveValue in the LOOPS class browser.

<RemoveFromBadList> Same as RemoveFromBadList in the LOOPS class browser.

<Where Is Agent?> This command allows selection of a DSPL agent type from a submenu, followed by the presentation of all agents currently defined for the problem solver of the selected type. The specialist containing that agent is then flashed in the Specialist Browser. Additionally, any agent browser containing the selected agent is also flashed.

<Edit> (agent) Similar to the Edit command of the Middle Button Commands, except that the desired agent is selected via a mechanism identical to the Where Is Agent? command. Several submenu options are available.

<Edit agent> Identical to the above command.

<Edit last> Invokes Dedit on the source of the last agent edited from the browser.

<Edit object code> Similar to the edit command, but invokes Dedit on the INTERLISP-D code generated by the DSPL parser for the selected agent.

<Edit last object code> Invokes Dedit on the INTERLISP-D code of the last agent edited from the browser.

<Edit unreferenced agent> Similar to the edit command, but the selection is made from a menu of agents which are referenced by no other agent in the problem solver.

<Edit bad source> Similar to the edit command, but the selection if made from a menu of agents known to have syntax errors in their DSPL source code.

<Delete> Deletes the selected agent from the problem solver. The agent is selected via a mechanism identical to the Where Is Agent? command.

<Add> (agent) Allows for the definition of new DSPL agents. This command causes a submenu of all DSPL agent types to be presented. Selection of an agent type causes Dedit to be invoked on a source code template of that type. From here, the Add command works similar to Edit. The following suboptions are available:

<Add agent> Same as above.

<Add undefined agent> Identical to the Middle Button Command.

<Inspect> (agent) Similar to the Left Button Command, except the selection mechanism is again similar to the Where Is Agent? command. Several submenu options are available:

<Inspect agent> Identical to the above command.

<Inspect last agent> Brings up an INTERLISP-D inspector window on the LOOPS instance of the last agent edited from this browser.

<Inspect problem solver> Brings up an INTERLISP-D inspector window on the instance of the problem solver.

<Browse> (specialist) Creates a browser on the selected agent from a list of all agents of a certain type. The default type is Specialist.

<Browse Specialist> This command will bring up a submenu of all the specialists currently defined in the problem solver. Selecting one of the specialists in the submenu will cause that specialist to be browsed.

<Browse Plan> Similar to Browse Specialist, but for plans.

<Browse FailureHandler> Creates a browser containing all failure handlers in the system.

3.2.3. The Specialist Browser

The Specialist Browser displays a lattice which shows the hierarchy of design specialists of the expert system. Bumper, for example, is a subspecialist of the Rest specialist, while the AirCylinder specialist is a superspecialist of the Spring, Head and Rest specialists. Each specialist of the AIR-CYL problem solver is responsible for a particular portion of the air cylinder design. As you might expect, the Spring specialist contains knowledge about designing the spring component, while the Bumper specialist contains knowledge about designing the bumper. In general, specialists lower in the hierarchy are responsible for progressively smaller sub-portions of the design problem, while the specialists higher in the hierarchy are responsible for larger assemblies in the design problem. In the AIR-CYL example, the top specialist coordinates the design of the entire air cylinder, while the tip specialists only contain knowledge about a single component in the device.

The specialist browser has the following commands in addition to the standard commands.

Left Button Commands

Set trace modes Determines which components of the selected specialist will be traced during execution. Note that tracing agents does not alter the computations made during execution. The agents to be traced are selected from a submenu of agent types in the system.

Title Menu Commands

<Save> Saves the entire problem solver to a loadable file. Since the both DSPL source and any generated INTERLISP-D code is saved, no reparsing by the DSPL system is required when the problem solver is reloaded.

<Run> Initiates execution of the problem solver. Several submenu options related to running the problem solver are available.

<Run> Identical to the above command.

<Set default trace modes> Similar to the Left Button Command in operation, except that the modes set by this command affect the tracing of all agents in the problem solver. This setting is overridden by trace modes set in an individual specialist.

<Graphic trace> This command enables a browser oriented form of tracing of the execution of the problem solver. In this mode a box is drawn around each agent as it is entered, and removed upon exit. Only agents currently being browsed are affected. A submenu of this command allows this mode to be turned either on or off, as desired.

<Single step> Causes the DSPL interpreter to halt before each agent is entered or exited. A menu pops up at the cursor to which must be buttoned to allow execution to continue. A submenu of this command allows single stepping to be turned either on or off, as desired.

3.2.4. The Specialist Component Browser

The Specialist Component Browser displays a lattice which shows the internal structure of a DSPL specialist down to the plan level. The use of browser is very similar to the Specialist Browser. Each node in the lattice represents a DSPL agent, which may be directly edited, displayed, or deleted via mouse actions.

The structure of a design specialist in DSPL is very constrained, and hence the lattice displayed in the Specialist Component Browser is very regular. The only types of DSPL agents that will be displayed in a Specialist Component Browser are specialists, selectors, plan sponsors, plans, and constraints. The root node of the lattice will always be the specialist whose components are being displayed. The rest of the agents in the lattice are organized to suggest relationships among the various components; selectors are displayed above the plan sponsors which the selector uses, plan sponsors are displayed above the plans being sponsored, etc.

The Specialist Component Browser has no additional commands over the standard commands described at the beginning of this section.

3.2.5. The Plan Component Browser

The Plan Component Browser parallels the Specialist Component Browser in both function and use. The Plan Component Browser displays a lattice which shows the internal structure of a DSPL plan and its components. Again, each node in the lattice represents a DSPL agent, which may be manipulated via mouse actions.

Plans are represented in DSPL as a sequence of actions. These actions may be thought of as commands to various types of agents to perform a specific job. DSPL plans currently contain only three such agent types; constraints, tasks, and specialists. The root node of the lattice will always be the plan whose components are being displayed. The agents referenced by the plan will appear directly beneath the plan in the browser. Additionally, the Plan Component Browser displays the structure of each task it contains. Components of each task are displayed beneath the task in the browser. Tasks are composed of agents of two types, constraints and steps. Finally, any redesign or failure handling knowledge referred to by an agent in the plan browser will be displayed beneath that agent.

The Plan Component Browser commands are identical to the Specialist Component Browser commands.

3.2.6. The Failure Handler Browser

The Failure Handler Browser displays a lattice showing relationships among every failure handler agent in the problem solver. Both system and user failure handlers are displayed.

Note that consistency is maintained among the DSPL browsers through any editing or other modifications performed via the browser commands. Deletion of an agent, for example, will result in the removal of that agent from every browser that the agent appears in.

3.2.7. The Message Trace Browser

The message browser does not show a lattice of DSPL agents. Instead, as its name implies, this browser displays a trace of the messages generated during the execution of a DSPL problem solver. The prohlem solver is initiated when a design message is sent to it. The problem solver forwards this message to the topmost specialist in its design hierarchy which in turn uses the message to activate its own plan selector in order to find an appropriate plan, etc. Each of the DSPL agents are activated by and respond with messages which can viewed via the Message Trace Browser.

Since the objects in the Message Trace Browser are not DSPL agents, the commands available somewhat different from the other hrowsers discussed.

Left Button Commands

Most of the left button commands are identical to the left button commands in the agent browsers, except that the agent that is operated on is typically the originator of the message displayed in the lattice.

Explain Displays an explanation window for this portion of the problem trace. This is the access mechanism to the explanation facilities of the DSPL problem trace.

<PP> Same as the PP command in the agent browsers.

<Inspect> (this message) Brings up an INTERLISP-D inspector window on the message instance buttoned. A submenu allows the originating agent to be inspected.

<Inspect this message> Same as above.

<Inspect agent> Inspects the agent which originated this message.

<Browse> Bring up a browser on the originator of this message.

<WhereIs> Same as the agent browser command.

Middle Button Commands

<Edit> Same as the agent browser command.

3.2.8. Running DSPL

The execution of a DSPL system proceeds in a top-down fashion, beginning from the top-most node in the design hierarchy. At each node in the specialist hierarchy, the knowledge encoded in the plan selectors and plan sponsors is used to select a plan appropriate to the current state of the planner. On finding such a plan, if one exists, the specialist proceeds to execute the plan. This overall control strategy of the DSPL interpreter is known as plan selection and refinement.

3.2.9. Building a DSPL Expert System

This section gives a brief, incomplete description of how to build an expert system in DSPL.

Having loaded the DSPL system from floppy, the creation of a new DSPL problem solver is begun by buttoning the create command of the DSPL icon. The name of the problem solver as well as the name of the top-most specialist is prompted for in the PROMPTWINDOW. Enter these items as requested. An empty specialist browser is displayed, from which the structure of the design problem solver can be entered. Buttoning the Add undefined agent command will display a menu with a single item on it; the name of the top-most specialist which was entered when the create command was buttoned. Buttoning this item will cause Dedit to display a DSPL specialist template with the name of the specialist already entered. Simply exiting from Dedit will cause this agent, the top-most specialist in the design hierarchy, to be added to the specialist browser. The first agent of the new design system has been created. Additional agents are added by using the Add agent command to edit DSPL templates as needed.

The recommended procedure for building a design system with DSPL is to first define the specialist hierarchy, then "flesh out" the hierarchy with design and rough-design plans and associated sponsor and selector knowledge in each specialist as appropriate. This gives a fairly complete overview of the system's organization. The addition of task and step knowledge is typically the most time consuming job in building a system due to the proportionately larger amount of knowledge to be entered. The task of entering this potentially large volume of data is made easier by the organized nature of the specialist hierarchy.

At any point in the development of the problem solver, the system may be executed to test its operation. Any missing agents necessary for execution will be noted by the DSPL interpreter. Missing DSPL constraint and step knowledge may be "dummied out" taking advantage of facilities such as ASKUSER in either step or constraint bodies.

3.3. HYPER: The Hypothesis Matcher Tool

Todd Johnson and John Josephson

INTRODUCTION

This paper describes HYPER -- a software tool that is used to build knowledge-based agents which perform the generic task of hypothesis matching for relevance. We first describe the classification tool called CSRL which gave rise to, and greatly influences, HYPER. Next, we describe hypothesis matching as a generic task and proceed to discuss the particulars of the tool. We then describe the types of explanation we expect from a hypothesis matcher. Finally we give an example of a system which uses hypothesis matching as a subtask.

3.3.1. CSRL -- Motivation for HYPER

Over the last two years much work has been done at OSU-LAIR using the classification system-building language called CSRL [4, 5, 6, 7, 12]. Using CSRL one can easily build systems which classify a description of a situation into a set of nodes in a class hierarchy. Portions of medical diagnosis can be thought of as classification where a patient's symptoms are classified into disease classes. Systems built using CSRL are organized as a classificatory hierarchy of conceptual specialists as in Figure 1. This figure represents part of the hierarchy used by an automobile diagnosis expert system, called Auto-Mech. Auto-Mech asks questions about a particular car and attempts to diagnose the problem by classifying the current state of the car as a specific malfunction class. Each specialist in the hierarchy represents a malfunction, with subnodes representing a more specific malfunction than their parent nodes. For example, LowOctane, WaterInFuel, and DirtInFuel are more detailed descriptions of the BadFuel "malfunction." Each specialist in the hierarchy contains knowledge that helps it to estabi'sh, that is, to determine whether the current situation is relevant to its concept. Thus BadFuel must "look" at the car's symptoms and decide if they "look like" a fuel problem.

In a CSRL system problem solving proceeds top-down using the Establish-Refine strategy developed in MDX [8]. First, the top node in the hierarchy attempts to establish itself. If it succeeds, then it attempts to refine itself by establishing its subnodes. In Figure 1, Auto-Mech establishes if it determines that something could be wrong with the car. Once Auto-Mech is established, FuelSystem will attempt to establish itself by determining whether the problem is with the car's fuel system. At run-time each specialist can be taken to represent a hypothesis concerning the relevance of its concept. For instance, in order to establish or reject itself BadFuel must determine the relevance of the hypothesis: "Something is wrong with the fuel." Thus hypothesis matching for relevance becomes an important subtask of classification.

So far nothing has been said about the representation of the knowledge used by each conceptual specialist. This knowledge must be used to map a partial situation description into evidence for or against the specialist's hypothesis. That is, the knowledge is used to determine the relevance of the specialist's concept to the current situation. CSRL encodes this information in a mechanism called a Knowledge Group. Knowledge Groups work by mapping situation features into a fixed range of confidence values. Each specialist contains a Knowledge Group which is invoked whenever the specialist is asked to establish itself. If the confidence value of the specialist's Knowledge Group is above a certain threshold, the specialist is considered to be established otherwise it is taken to be rejected. Thus Knowledge Groups do the work of matching for relevance.

After building several systems we began to realize the usefulness of this task in non-classification systems. In fact, we decided that hypothesis matching for relevance should be a separate generic task. Work then began to separate CSRL into two separate tools: CSRL for classification and HYPER for Hypothesis Matching.

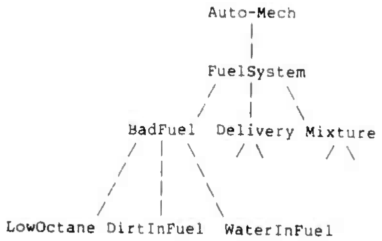


Figure 19: Auto-Mech's conceptual specialist hierarchy

3.3.2. Hypothesis Matching as a Generic Task

A Generic Task is characterized by a task specification, the specific kinds and organization of domain knowledge, and a family of control regimes appropriate to the task [9]. This information is vital to the production of a good knowledge level tool since without it we can produce little more than an adhoc and narrowly useful system. The generic task formulation for hypothesis matching is as follows:

Task Specification

Given a concept and a set of situation features, determine the degree to which the concept matches the situation.

Organization of Knowledge

A hierarchical organization of evidence abstractions. The top node computes the degree to which the concept matches the situation. Subnodes compute evidence components for their parent nodes. For example, the BadFuel hypothesis matcher in Figure 2 has two subnodes: PerformanceRelated and FillupRelated. These subnodes respectively rate the evidence for Performance problems and Fillup problems indicative of bad fuel. A similar task is performed by Samuel's signature tables.

Kinds of Knowledge

What the evidence components are, how to determine their strengths, and how to combine evidence.

Control

Control is initiated in a top-down fashion. The top node can call on any of its subnodes to gather evidence. Evidence abstraction data flows bottom-up.

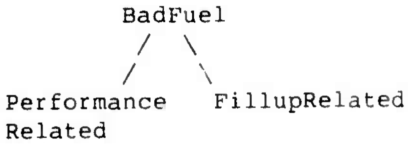


Figure 20: Hierarchical structure of the BadFuel hypothesis matcher

Details of HYPER

As a tool HYPER provides the following facilities to the system builder:

1. A browser for creating, displaying, and editing the evidence abstraction hierarchy for a hypothesis matcher.

2. A language for representing the knowledge which maps features to confidence values. This language represents the internal structure of each node in a hypothesis matcher's hierarchy.

3. Explanation Facilities (Discussed in the next section.)

The hypothesis matchers produced using HYPER are independent knowledge-based agents. Invocation is accomplished by sending a Match message to a particular matcher. When a hypothesis matcher receives a Match message, it evaluates the features and returns a confidence value. A confidence value is a symbolic measure of relevance. The default range provided by HYPER is: HighlyUnlikely, Unlikely, Unknown, Likely, and HighlyLikely. The range of confidence values can be supplied by the system designer to suit whatever purpose is needed.

A tabular representation of the BadFuel hypothesis matcher is shown in Figure 3. The column headings represent the features to be matched against each row of the table. The entries in each row are tests to be performed upon the corresponding features. Question marks represent "don't care" conditions. For BadFuel the features are actually the result of the evidence components FillupRelated and PerformanceRelated. Each row in the table represents a set of tests to apply to the features followed by the confidence value to be returned if the row matches. The confidence value can either be one of the symbolic values or a hypothesis matcher which can be used to compute a value. For example, if both FillupRelated and PerformanceRelated returned HighlyLikely, then the first row in the table would match. In this case, BadFuel would return HighlyLikely. The rows are evaluated from top to bottom, left to right, until a row matches. The confidence value of the matching row is then returned. A certain amount of optimization is done during the evaluation of the table to avoid evaluating unnecessary components.

```
BadFuel:
  PerformanceRelated        FillupRelated
  ------------------------------------------
  (EQ HighlyLikely)       (GE Unknown)     => HighlyLikely
  (EQ Likely)             (GE Unknown)     => Likely
         ?                (LT Unknown)     => HighlyUnlikel
         ?                       ?         => PerformanceRelated
```

Figure 21: Top node of the BadFuel matcher

3.3.3. Explanation in HYPER

Since hypothesis matchers are viewed as independent agents, it makes sense to directly ask a matcher about its behavior rather than an additional "module" whose purpose is to construct an explanation. Also, because hypothesis matching is a generic task, knowledge and control are represented at a level which facilitates explanation. For these reasons, hypothesis matchers designed using HYPER come complete with the ability to handle the following explanatory questions.

Why Value? Asks for an explanation of why a certain value was returned. This requires knowledge of run-time behavior.

Justify Knowledge
Questions of the form "Why do you say knocking and pinging indicate a high likelihood for bad fuel?" Such questions require justification of the knowledge being used by the agent.

Why not value? This asks for an explanation of why a certain value was *not* returned. Such questions require knowledge of the control strategy, as well as, the run-time behavior. Other possible questions of this form include: Why not higher/lower, and What do I need to do to make the value X?

"Why value" questions can easily be answered by simply stating why rows failed or succeeded. An example of this is given in the next section. Because Hypothesis Matchers represent compiled knowledge, justification requires the use of pre-canned strings. HYPER provides a facility for attaching appropriate explanatory strings to each row of the table. Explanations given by HYPER can appear in either a machine readable form or a human readable form, thus explanations can be used by both other agents and the human user of the system.

3.3.4. Using HYPER from CSRL -- An Example

The following example shows how a hypothesis matcher can be used from a classification system. We will use the portion of Auto-Mech shown in Figure I. To begin let us assume that BadFuel has received an establish-refine message. The top node of BadFuel's hypothesis matcher is shown in Figure 3. The two subnodes, PerformanceRelated and FillupRelated, are shown in Figure 4. The function AskYNU? asks the user a question expecting a reply of yes, no, or unknown, and returns T, F, or U.

When the BadFuel specialist receives an Establish message it must attempt to establish or reject itself. To do this it sends a Match message to its hypothesis matcher. Referring to Figure 3, the matcher first attempts to evaluate (EQ HighlyLikely) with respect to PerformanceRelated. In order to perform this comparison, PerformanceRelated must be evaluated. The matcher then calls PerformanceRelated causing the sequence of events shown in Figure 5. Since in this case HighlyLikely is returned, the comparison succeeds and the matcher tries to determine whether FillupRelated is greater-than or equal to Unknown (GE Unknown); continuing with evaluating the first row of BadFuel as shown in Figure 3. Thus FillupRelated must be evaluated. Figure 6 shows the sequence of events resulting in a confidence value of HighlyUnlikely. Since HighlyUnlikely is less than Unknown, (GE Unknown) fails thus causing the first row of BadFuel to fail. The matcher then moves to the second row and immediately fails on (EQ Likely) since PerformanceRelated returned HighlyLikely. Next the third row is tried. The first test is a "don't care" condition so evaluation proceeds to the second test in the row (LT Unknown). Since FillupRelated returned HighlyUnlikely the test clearly succeeds, meaning the entire row has matched. The matcher then returns the associated confidence value, HighlyUnlikely, to the BadFuel Specialist.

Now that the BadFuel specialist has a confide value it must decide whether to reject or establ The establish threshold is set at Likely so with a c fidence value of HighlyUnlikely BadFuel firmly reje itself and does not attempt to establish any of its s nodes.

```
PerformanceRelated:
  Q1:   AskYNU? ``Is the car slow to respond''
  Q2:   AskYNU? ``Does the car start hard''
  Q3:   (And AskYNU? ``Do you hear knocking or pinging sounds''
         AskYNU? ``Does the problem occur while accelerating''

  Q1             Q2              Q3
-------------------------------------------------
(EQ T)           ?               ?          => HighlyUnlikely
  ?            (EQ T)            ?          => HighlyUnlikely
  ?              ?             (EQ T)       => HighlyLikely
  ?              ?               ?          => Unknown

FillupRelated:
  Q1:   AskYNU? ``Have you tried a higher grade of gas''
  Q2:   AskYNU? ``Did the problem start after the last fillup''
  Q3:   AskYNU? ``Has the problem gotten worse since the last
                  fillup''

  Q1             Q2              Q3
-------------------------------------------------
(EQ T)           ?               ?          => HighlyUnlikely
  ?            (EQ T)            ?          => HighlyLikely
  ?            (EQ F)          (EQ T)       => Likely
  ?              ?               ?          => HighlyUnlikely
```

Figure 22: Tabular representation of BadFuel's subnodes

```
(BadFuel sends a Match message to PerformanceRelated)
Is the car slow to respond? no
Does the car start hard? no
Do you hear knocking or pinging sounds? yes
Does the problem occur while accelerating? yes
(PerformanceRelated returns HighlyLikely)
```

Figure 23: Run-time snapshot of PerformanceRelated

```
(BadFuel sends a Match message to FillupRelated)
Have you tried a higher grade of gas? yes
(FillupRelated returns HighlyUnlikely)
```

Figure 24: Run-time snapshot of FillupRelated

Suppose now that the person running Auto-Mech wishes to know why the BadFuel specialist rejected itself. Without appeal to its hypothesis matcher, the specialist can only answer the question by saying that HighlyUnlikely was less than the establish threshold. However, since HYPER provides explanation facilities, BadFuel can send the message "Why HighlyUnlikely?" to its matcher and give the user a better explanation, such as that shown in Figure 7. A general explanation browser may then be used to ask further questions about the initial explanation.

```
BadFuel hypothesis matcher resulted in Hig

         PerformanceRelated returned Highly
         FillupRelated returned HighlyUnlik

         (GE FillupRelated Unknown) is fals
         (EQ PerformanceRelated Likely) is
         (LT FillupRelated Unknown) is true

HighlyUnlikely is below the establish thre
BadFuel rejected.
```

Figure 25: Explanation about why BadFuel rejecte

CONCLUSION

Because hypothesis matching appears to be a very useful generic task we feel that a robust version of HYPER is needed for our set of high-level tools. Such a version will greatly speed the development of other useful tools and systems. The first implementation of HYPER has just been completed and is undergoing testing. Part of this testing involves the rewriting of CSRL to allow the use of hypothesis matchers as independent agents separate from the CSRL language. This is beginning to bring up issues about agent integration, and about the designer interface needed to switch between several cooperating high level tools. Thus, HYPER is forcing us to look at issues vital to the production of a useful set of knowledge level tools.

Acknowledgments

We gratefully acknowledge the cooperation of the Mitre Corporation in providing us with the code and knowledge base for the KNOBS system, and for taking the time to host a visit by some of us to view the system and ask probing questions.

REFERENCES

[1] Brown, D.C. / Chandrasekaran, B.
Expert Systems for a Class of Mechanical Design Activity.
1984
Paper for IFIP WG5.2 Working Conference. Sept. 84.

[2] Brown, D.C.
Expert Systems for Design Problem-Solving Using Design Refinement with Plan Selection and Redesign.
1984
Dissertation.

[3] D. C. Brown and B. Chandrasekaran.
Knowledge and Control for Design Problem Solving.
January 10, 1985.
Technical Report, Laboratory of Artificial Intelligence Research, Department of Computer and Information Science, The Ohio State University.

[4] Bylander, T. / Mittal, S. / Chandrasekaran, B.
CSRL: A Language for Expert Systems for Diagnosis.
In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 218-221. , August, 1983.
An extended article appears in the Special Issue of *Intn'l Jrnl. of Computers and Mathematics* on "practical artificial intelligence systems", and another article on CSRL, with emphasis on uncertainty handling will soon appear in *AI Magazine*.

[5] T. Bylander and J. W. Smith, M.D.
Using CSRL for Medical Diagnosis.
In *Proceedings of MEDCOMP'83*. IEEE Computer Society, 1983.

[6] T. Bylander.
Syntax and Semantics of CSRL in INTERLISP-D.
April 9, 1985
Technical report, Laboratory of Artificial Intelligence Research, Department of Computer and Information Science, The Ohio State University.

[7] T. Bylander.
Using CSRL in INTERLISP-D.
April 9, 1985
Technical report, Laboratory of Artificial Intelligence Research, Department of Computer and Information Science, The Ohio State University.

[8] Chandrasekaran, B.
Decomposition of Domain Knowledge into Knowledge Sources: The MDX Approach.
Proc. 4th Nat. Conf. Canadian Society for Computational Studies of Intelligence :1-8, May, 1982.
An expanded version appears as "Towards a Taxonomy of Problem Solving Types," in the Winter 1983 issue of *AI Magazine*.

[9] Chandrasekaran, B.
Generic Tasks in Expert System Design and Their Role in Explanation of Problem Solving.
May 1985
Invited paper presented at the National Academy of Sciences/ Office of Naval Research Workshop on Distributed Problem Solving, May 16-17, 1985, Washington, D.C., appears in the *Proc. of the Workshop* to be published by the National Academy of Sciences.

[10] Engelman, C. / Millen, J.K. / Scarl, E.A.
KNOBS: An Integrated AI Interactive Planning Architecture.
1984

[11] Sembugamoorthy, V. / Chandrasekaran, B
Functional Representation of Devices and Compilation of Diagnostic Problem Solving Systems.
August, 1984
To appear in *Cognitive Science*.

[12] M. C. Tanner and T. Bylander.
Application of the CSRL Language to the Design of Expert Diagnosis Systems: The Auto-Mech Experience.
In *Proceedings of the Joint Services Workshop on Artificial Intelligence in Maintenance*, pages 131-152. Department of Defense, 1984.

GENERIC TASKS IN EXPERT SYSTEM DESIGN AND THEIR ROLE IN EXPLANATION OF PROBLEM SOLVING[1]

B. Chandrasekaran
Laboratory for Artificial Intelligence Research
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

ABSTRACT

We outline the elements of a framework for expert system design that we have been developing in our research group over the last several years. This framework is based on the claim that complex knowledge-based reasoning tasks can often be decomposed into a number of *generic tasks each with associated types of knowledge and family of control regimes.* At different stages in reasoning, the system will typically engage in one of the tasks, depending upon the knowledge available and the state of problem solving. The advantages of this point of view are manifold: (i) Since typically the generic tasks are at a much higher level of abstraction than those associated with first generation expert system languages, knowledge can be represented directly at the level appropri ie to the information processing task. (ii) Since each of the generic tasks has an appropriate control regime, problem solving behavior may be more perspicuously encoded. (iii) Because of a richer generic vocabulary in terms of which knowledge and control are represented, explanation of problem solving behavior is also more perspicuous. We briefly describe six generic tasks that we have found very useful in our work on knowledge-based reasoning: classification, state abstraction, knowledge-directed retrieval, object synthesis by plan selection and refinement, hypothesis matching, and assembly of compound hypotheses for abduction.

1. Information Processing Tasks in Knowledge-Based Reasoning

Intuitively one thinks that there are types of knowledge and control regimes that are common to diagnostic reasoning in different domains, and similarly there would be common structures and regimes for say design as an activity, but that the structures and control regimes for diagnostic reasoning and design problem solving will be generally speaking different. However, when one looks at the formalisms (or equivalently the languages) that are commonly used in expert system design, the knowledge representation and control regimes do not typically capture these distinctions. For example, in diagnostic reasoning, one might generically wish to speak in terms of malfunction hierarchies, rule-out strategies, setting up a differential, etc., while for design, the generic terms might be device/component hierarchies, design plans, ordering of subtasks, etc. Ideally one would like to represent diagnostic knowledge in a domain by using the vocabulary[2] that is appropriate for the task. But typically the languages in which the expert systems have been implemented have sought uniformity across tasks, and thus have had to lose perspicuity of representation at the task level. The computational universality of representation languages such as Emycin or OPS5 -- i.e., the fact that any computer program can be written in these languages, more or less naturally -- often confuses the issue, since after the system is finally built it is often unclear which portions of the system represent domain expertise and which are programming devices. In addition, the control regimes that these languages come with (in rule-based systems they are typically variants of *hypothesize and match*, such as forward or backward chaining) do not explicitly indicate the real control structure of the system at the task level. E.g., the fact that R1 [12] performs a linear sequence of subtasks -- a very special and atypically simple version of design problem solving -- is not explicitly encoded: the system designer so to speak "encrypted" this control in the pattern-matching control of OPS5.

These comments need not be restricted to the rule-based framework. One could represent knowledge as sentences in a logical calculus and use logical inference mechanisms to solve problems. Or one could

[1] Research supported by Defense Advanced Research Projects Agency, RADC Contract F30602-85-C-0010, and Air Force Office of Scientific Research grant 82-0255. This paper was originally presented at the Office of Naval Research/National Academy of Sciences Symposium on Distributed Artificial Intelligence, May 1985.

[2] We also use the term *primitives of the language* in the rest of the paper to refer to the vocabulary.

represent it as a frame hierarchy with procedural attachments in the slots. (It is a relatively straightforward thing, e.g, to rewrite MYCIN [14] in this manner, see [16].) In the former, the control issues would deal with choice of predicates and clauses, and in the latter, they will be at the level of which links to pursue for inheritance, e.g. None of these have any natural connection with the control issues natural to the task.

Actually the situation is even worse: because of the relatively low level of abstraction relative to the information processing task, there are control issues that are artifacts of the representation, but often in our opinion misinterpreted as issues at the "knowledge-level." E.g., rule-based approaches often concern themselves with conflict resolution strategies. If the knowledge were viewed at the level of abstraction appropriate to the task, often there will be organizational elements which would only bring up a small, highly relevant pieces of knowledge or rules to be considered without any conflict resolution strategies needed. Of course, these organizational constructs could be "programmed" in the rule language, but because of the status assigned to the rules and and their control as knowledge-level phenomena (as opposed to the implementation level phenomena, which they often are), knowledge acquisition is often directed towards strategies for conflict resolution, whereas the really operational expert knowledge is at the organizational level.

This level problem with control structures is mirrored in the relative poverty of knowledge-level primitives for representation E.g., the epistemology of rule systems is exhausted by data patterns (antecedents or subgoals) and partial decisions (consequents or goals), that of logic is similarly by predicates, functions, and related primitives. If one wishes to talk about types of goals or predicates in such a way that control behavior can be indexed over this typology, such a behavior can often be programmed in these systems, but there is no explicit encoding ol them that is possible. E.g., Clancey [8] found in his work using Mycin to teach students that for explanation he needed to attach to each rule in the Mycin knowledge base encodings of types of goals so that explanation of its behavior can be couched in terms of this encoding, rather than only in terms of "Because <...> was a subgoal of <...>."

The above is not to argue that rule representations and backward or forward chaining controls are not "natural" for some situations. If all that a problem solver has in the form of knowledge in a domain is a large collection of unorganized associative patterns, then data-directed or goal-directed associations may be the best that the agent can do. But that is precisely the occasion for weak methods such as hypothesize and match (of which the above associations are variants), and, typically, successful solutions cannot be expected in complex problems without combinatorial searches. Typically, however, expertise consists of much more organized collections of knowledge, with control behavior indexed by the kinds of organizations and forms of knowledge in them.

To summarize the argument so far: There is a need for understanding the generic information processing tasks that underlie knowledge-based reasoning. Knowledge ought to be directly encoded at the appropriate level by using primitives that naturally describe the domain knowledge for a given generic task. Problem solving behavior for the task ought to be controlled by regimes that are appropriate for the task. If done correctly, this would simultaneously facilitate knowledge representation, problem solving, and explanation.

At this point it will be useful to make further distinctions. Typically many tasks that we intuitively think of as generic tasks are really *complex generic tasks*. I. e., they are further decomposable into components which are more elementary in the sense that each of them has a homogeneous control regime and knowledge structure. For example, what one thinks of the diagnostic task, while it may be generic in the sense that the task may be quite similar across domains, it is not a unitary task structure. Diagnosis may involve classificatory reasoning at a certain point, reasoning from one datum to another datum at another point, and abductive assembly of multiple diagnostic hypotheses at another point. Classification has a different form of knowledge and control behavior from those for data-to-data reasoning, which in turn is dissimilar in these dimensions from assembling hypotheses.

Thesis: Given a complex real world knowledge-based reasoning task, and a set of generic tasks for each of which we have a representation language and a control regime to perform the task, if we can perform an epistemic analysis of the domain such that (i) the complex task can be decomposed in terms of the generic tasks, (ii) paths and conditions for information transfer from the agents that perform these generic tasks to the others which need the information can also be established, and (iii) knowledge of the domain is available to encode into the knowledge structures for the generic tasks; then that complex task can be "knowledge-engineered" successfully and perspicuously. Notice that an ability to *decompose* complex tasks in this way brings with it the ability to *characterize* them in a useful way. We can see, e.g., that the reason that we are not yet able to handle difficult design problem solving is that we are often unable to find an architecture of generic tasks in terms of which the complex task can be constructed.

In the rest of this paper, we will briefly describe some of the elementary generic tasks that we have had occasion to identify and use in the construction of expert systems. While we have been adding to our repertoire of elementary generic tasks over the years, the basic elements of the framework have been in place for a number of years. Our work on MDX [4, 5], e.g., identified *classification*, *knowledge-directed information passing*, and *hypothesis matching* as three generic tasks, and showed how certain classes of diagnostic problems can be implemented as an integration of these generic tasks. (We have earlier referred to them as *problem solving types*, but in [6], we began to call them generic tasks.) Over the years, we have identified several others: *object synthesis by plan selection and refinement* [1], *state abstraction* [7], and *abductive assembly of hypotheses* [11]. There is no claim that these are exhaustive; in fact, our ongoing research objective is to identify other useful generic tasks and understand their knowledge representation and control of problem solving.

2. Some Generic Tasks

2.1. Characterization of Generic Tasks

Each generic task is characterized by the following:

1. A task specification in the form of generic types of input and output information.

2. Specific forms in which the basic pieces of domain knowledge is needed for the task, and specific organizations of this knowledge particular to the task.

3. A family of control regimes that are appropriate for the task.

From the nature of the control regime, we can determine the *types of strategic goals* the problem solving for the task has. These goal types will play a role in providing explanations of its problem solving behavior.

When a complex task is decomposed into a set of generic tasks, it will in general be necessary to provide for communication between the different structures specializing in these different types of problem solving. Note that a decomposition does not imply that there is a predetermined temporal ordering on when the generic tasks are performed: typically the agent for a generic task is invoked when another agent needs information that the former can provide. Further there is no implication that there is a unique decomposition. Depending upon the availability of particular pieces of knowledge, different architectures of generic tasks will typically be possible for a given complex task.

We will now proceed to a brief characterization of these generic tasks.

- *I. Classification*

Task specification: Classify a (possibly complex) description of a situation as an element, as specific as possible, in a *classification hierarchy*. E.g, classify a medical case description as an element of a disease hierarchy.

Forms of knowledge: <partial situation description> ---> <evidence belief about confirmation or disconfirmation of classificatory hypotheses. E.g., in medicine, a piece of classificatory knowledge may be: certain pattern in X-ray & bilirubin in blood ---> high evidence for cholestasis.

Organization of knowledge: The above classificatory knowledge *distributed* among concepts in a classificatory concept hierarchy. Each conceptual "specialist" ideally contains knowledge that helps it determine whether it (the concept it stands for) can be *established* or *rejected*. The form of the knowledge as stated above is the form needed for this decision.

Control Regime: (Simplified form) Problem solving is top down. Each concept when called tries to establish itself. If it succeeds, it lists the reasons for its success, and calls its successors, which repeat the process. If a specialist fails in its attempt to establish itself, it rejects itself, and all its successors are also automatically rejected. This control strategy can be called *Establish-Refine*, and results in a specific classification of the case. (The account is a simplified one. The reader is referred to [5] for details and elaborations.)

Goal types: E.g., Establish <concept>, Refine (subclassify) <concept>

Example Use: Medical diagnosis can often be viewed as a classification problem. In planning, it is often useful to classify a situation as of a certain type, which then might suggest an appropriate plan.

- *II. State abstraction*

Task Specification: Given a change in some state of a system, provide an account of the changes that can be expected in the functions of the system. (Useful for reasoning about consequences of actions on complex systems.)

Form of knowledge: <change in state of subsystem> ---> <change in functionality of subsystem> <change in state of the immediately larger system>

Organization of Knowledge: Knowledge of the above form distributed in conceptual specialists corresponding to system/subsystems. These conceptual specialists are connected in a way that mirrors the way the system/subsystem is put together.

Control regime: Basically bottom up, but follows the architecture of the system/subsystem relationship. The changes in states are followed through, interpreted as changes in functionalities of subsystems, until the changes in the functionalities at the level of abstraction desired are obtained.

Goal Types: E.g., Abstract consequent state. Deduce change in functionality.

Example Use: Answering questions of the form: "What will happen if this valve is closed, while the turbine is running?" Generic usefulness is in consequence finding.

- *III. Knowledge-Directed Information Passing*

Task specification: Given attributes of some datum, it is desired to obtain attributes of some other datum, conceptually related to the original datum.

Forms of Knowledge: i. Default value of <attribute> of <datum> is <value> ii. <attribute>ψof <datum>ψis inherited from <attribute> of parent of <datum> iii. <attribute> of <datum> is related as <relation> to <attribute> of children of <datum>. iv. <attribute> of <datum> is related as <relation> to <attribute> of <concept>.

Organization of Knowledge: The concepts are organized as a *frame hierarchy.* Default for slots corresponds to form i. above, the IS-A or PART-OF links between parents and children determine the types of inheritance in form ii. and iii. Procedural attachments or "demons" are used to encode form iv. Each frame is a specialist in knowledge-directed data inference for the concept.

Control regime: A concept, when asked for the value of one of its attributes first checks the data base to see if the actual value is known, then uses inheritance relationships to determine if the value can be obtained by inference from the values of appropriate attributes of its parent or children, then uses any demons that may be attached to the slot to query other concepts in other parts of the hierarchy for values of their attributes. If none of it succeeds and if it is appropriate the default value is produced as the value.

This is basically a hierarchical information-passing control regime, with demons providing an override of the hierarchical regime.

Goal Types: E.g., Inherit value of <attribute>. Ask for <concept, attribute value) to infer <attribute> by <relation>.

Example Use: Knowledge-based data retrieval tasks in wide variety of situations. Inferring a medical datum from another, when the latter is available but the former is needed for diagnostic reasoning. E.g., diagnostic reasoning needs information about whether the patient has been exposed to "anesthetics," because it has diagnostic knowledge that relates a diagnostic conclusion to this datum, but the patient data do not include any reference to "anesthetics," but mentions "major surgery a few weeks before." Assuming that the knowledge base for the data retrieval system encodes the piece of knowledge that relates "surgery" and "possible exposure to anesthetics," performing the reasoning that connects the two data items is an example of knowledge-based data retrieval.

- *IV. Object Synthesis by Plan Selection and Refinement*

Task Specification: Design an object satisfying specifications (object in an abstract sense: they can be plans, programs, etc.).

Forms of knowledge: Object structure is known at some level of abstraction, and pre-compiled plans are available which can make choices of components, and have lists of concepts to call upon for *refining* the design at that level of abstraction.

Organization of Knowledge: Concepts corresponding to "components" organized in a hierarchy mirroring the object structure. Each concept has plans which can be used to make commitments for some "dimensions" of the component.

Control Regime: Top down in general. The following is done recursively until a complete design is worked out: A specialist corresponding to a component of the object is called, the specialist chooses a plan based on some specification, instantiates and executes some part of the plan which suggests further specialists to call to set other details of the design. Plan failures are passed up until appropriate changes are made by higher level specialists, so that specialists who failed may succeed on a retry.

Goal Types: E.g., Choose plan, execute <plan element>, refine <plan>, redesign (modify) <partial design> to respond to failure of <subplan>S, select alternative plan, etc.

Example: Expert design tasks, synthesis of everyday plans of action.

- *V. Hypothesis Matching*

Task Specification: Given a hypothesis and a set of data that describe the problem state, decide if the hypothesis matches the situation.

Form and Organization of Knowledge: (One form) A hierarchical representation of evidence abstractions, top node is the degree of matching of the hypothesis to the data, and nodes at a given level are components of evidence for the evidence abstraction at the higher level. E. g., say the hypothesis of *goodness* of a position in a game is the one to be matched against the data describing the board configuration. Goodness may be defined at the top level in terms of two abstractions: *defensibility* and *offensive opportunities*. Form of knowledge then for this must be such as to enable mapping degrees of belief in each of these evidence abstractions to degree of belief in the *goodness* abstraction. The *defensibility* abstraction, e.g., may in turn be defined either by direct data or intermediate abstractions. Samuel's *signature tables* can be thought of as performing this task.

Goal types: Evaluate evidence for hypothesis, evaluate evidence for contributing abstraction .

- *VI. Abductive Assembly of Explanatory Hypotheses*

Task Specification: Given a situation (described by a set of data items) to be explained by the best explanatory account, and given a number of hypotheses, each associated with a degree of belief and each of which offers to explain a portion of the data (possibly overlapping with data to be accounted for by other hypotheses), construct the best composite hypothesis out of the given hypotheses.

Forms of Knowledge: causal or other relations (such as incompatibility, suggestiveness, special case of) between the hypotheses, relative significance of data items.

Organization of Knowledge: For relatively small number of hypotheses, this is a global process. For large numbers, some form of recursive assembly will be called for, implying knowledge organized at different levels of abstraction of the assembled hypotheses.

Control Regime: (Simplified version: see [11] for a fuller discussion.) Assembly and criticism alternate. In assembly, a means-ends regime, driven by the goal of explaining all the significant findings, is in control. At each stage, the most significant datum to be explained results in the best hypothesis that offers to explain it being added to the composite hypothesis so far assembled. After each assembly, the critic removes explanatorily superfluous parts. This loops until all the data are explained, or no hypotheses are left.

Goal Types: e.g. account-for <datum>, check-superfluousness-of <hypothesis>.

Example Use: In medical diagnosis, the classification generic task may produce a set of classifications, each of which accounts for some of the data. The best account needs to be put together. The Internist system [13] and the Dendral system [2] perform this type of task as part of their problem solving.

3. Encoding Knowledge at the Level of the Task

For each generic task, the form and organization of the knowledge directly suggest the appropriate representation in terms of which domain knowledge for that task can be encoded. Since there is a control regime associated with each task, the problem solver can be implicit in the representation language. I.e., as soon as knowledge is represented in the shell corresponding to a given generic task, a problem solver which uses the control regime on the knowledge representation created for domain can be created by the interpreter. This is similar to what representation systems such as EMYCIN do, but note that we are deliberately trading generality at a lower level to specificity, clarity, richness of ontology and control at a higher level.

We have designed and implemented representation languages for a simpler versions of two of these generic tasks: classification [3], and object synthesis by selection and refinement [1]. We plan to implement a family of such representation languages.

4. Generic Tasks and Explanation of Problem Solving

We have developed a framework for providing explanations for the decisions recommended by expert systems, and this is the basis of a four-year research effort sponsored by the Defense Advanced Research Projects Agency. For the purpose of this discussion, we can say that understanding the problem solving behavior of an expert problem solving system requires inspecting three structures, each corresponding to a type of explanation:

Type 1: trace of run-time, data-dependent problem solving behavior, viz., which pieces of knowledge were used and how. E.g., Why do you say that the patient has cholestasis? Ans: Because the patient has high bilirubin in blood, and jaundice and Xrays suggest an obstruction in the biliary duct. This is typically done by checking which data items in the current case matched the piece of knowledge that enabled the system to make a particular decision.

Type 2: understanding how a piece of knowledge relates to the domain, how it can be justified. E.g., Why do you say that high bilirubin in blood suggests cholestasis? Here whatever answer is given, no data about the current case are being used. The system is really being called upon to justify its knowledge.

Type 3: understanding the control strategy used by the program in a particular situation. E.g. Why didn't you consider portal hypertension in this case? Ans: Because I had ruled out circulatory diseases, portal hypertension is a special case of circulatory diseases, and *my strategy is not to consider special cases when I have ruled out the general case.*

The explicit encoding of the generic aspects of knowledge and control behavior for each generic task can be directly used to produce explanations of Type 3. We will give some examples

Classification

Q: Why do you wish to know if the patient had been exposed to anesthetics?

A: Because I was trying to *establish* hepatitis.

Q : Why?

A: Because I had already established that the patient had liver disease. Now I am trying to subclassify the case, and hepatitis is one of the subcategories. If I can *reject* hepatitis, I can reject all the following diseases which are its subclasses.

State Abstraction

Let us assume that the user had asked the system, "What will happen if Valve A is closed?"

Q: Why do wish to know the engine speed?

A: Because I am trying to decide the *change in functionality of* the cooling system when Valve A is closed, and engine speed is a datum I need for this.

Q: Why are you considering the cooling system?

A: Because, Valve A is a *component of* the Cooling system.

Data Retrieval

Q: Did the patient have exposure to anesthetics?

A: Did he have any surgery?

Q: Why do you ask?

A: I do not have direct record of anesthetic exposure. I failed to *inherit* a value for this attribute from its parent -- no direct record of any drug not being given. I also failed to infer No Exposure, by *ruling out its children.* Then the Anesthetic concept *suggested* that If surgery had been performed, anesthetic can be reasonably inferred.

Hypothesis Assembly

Q: Why was hypothesis part H' included in the best explanation?

A: In order to *account-for* < datum >

Q: Why wasn't H'' chosen to explain D?

A: Because assuming < partially assembled conclusion >, H' is the best way to explain < cluster of data >.

Q: Why was hypothesis H accepted?

A: Because it is the only plausible way to *account-for* < cluster of data >.

Plan Refinement

Q: Why did you choose Plan A'?

A: Because, I am trying to complete the specification for Plan A, for *refining* which I need <subgoal> accomplished. The specialist for <subgoal> selected Plan A' due to <reasons>.

Q: What will you do if you fail in Plan A'?

A: <Subgoal> specialist will *select* Plan A".

Q: What if it fails?

A: Parent specialist will ***redesign*** Plan A, by weakening <constraint>.

In the foregoing examples, the italicized terms represent the type of goal that is being pursued. Points to be noted here are: this explanatory richness (compared to the terminology of goal-subgoals) is made by possible by encoding the control regimes specific to each generic task: and, the explanation is directly related to the problem solving of the system.

4.1. Comparison with Related Work

With respect to providing explanation there are two key ideas that we are offering in this paper: one, explanation of problem solving strategies, which are manifested as appropriate control behavior by the problem solver, can be based on the generic task that a problem solver is engaging at a given stage in problem solving; and two, which is implicit in what we have said so far, is that control for each task be represented abstractly so that explanations can be couched in terms of these abstractions.

Swartout and Clancey have done significant investigations of issues in explanation generation by problem solving systems. The work of both authors uses the notion of ***abstract representation of control*** as a basic idea for explanation. It will be useful to relate our ideas to those of these investigators.

4.1.1. The Work of Clancey's Group:

Clancey has contributed several ideas that are relevant in this context: one, in [9], he discussed the advantages of abstract representation of control in reasoning systems, and specifically pointed out their potential role in explanation; two, in [8], he proposed that, in order to give explanatory capabilities to MYCIN for purposes of teaching (he created a system called GUIDON based on MYCIN) an explanatory skeleton be attached to each rule encoding the role of the rule in problem solving; and three, in his work on NEOMYCIN [10], he and his group represent the diagnostic strategy explicitly (in terms of abstract subtasks and their relations to diagnosis on the one hand and to the domain data on the other).

The most advanced work by Clancey's group on explanation is that on NEOMYCIN, and thus we will concentrate on that in this section. Here diagnostic strategy is represented explicitly as a collection of subtasks, with conditions for moving from subtask to subtask also explicitly stated. This representation enables an explanation of strategy to be produced at the task and sub-task level of generalization.

This work is in many ways quite close in spirit to our approach, with the following comments throwing light on the differences.

1. NEOMYCIN's representation of abstract strategies is implemented as a body of metarules in the rule-based paradigm. We would note here that the rule paradigm plays no intrinsic role in this and can be viewed as *merely* an implementation language. In our approach we would advocate a representation language with generic primitive terms for directly encoding control along the lines discussed earlier in the paper.

2. The above comment raises the question of the appropriate language in which conch the tasks abstractly. In this paper we have proposed a set of generic tasks and suggested that they (and others to be added as needed on empirical grounds, but at about the same level of grain size) comprise the elementary tasks in terms of which complex (generic) tasks such as diagnosis be decomposed. While we have been able to demonstrate this claim to a certain extent for the diagnostic strategy employed by the MDX system, it is a matter of further empirical research to see whether and how NEOMYCIN's diagnostic strategy be so decomposed.

With respect to point 2 above, are there advantages from an explanation point of view for such a decomposition even if it were possible? At this point we can only give the following tentative answers. To the extent that the subtasks in NEOMYCIN were developed by a direct study of the diagnostic task, it is likely that some of these tasks (and consequently the terms which they contribute to the explanation) are more informative at the diagnostic task level. But if our theory is right, the additional abstractions specific to diagnosis can be obtained naturally from the abstraction at the generic task level. The generic tasks in our sense will have the further advantage of providing the primitives for other "molecular" tasks in addition to diagnosis.

4.1.2. Swartout and the XPLAIN System:

Swartout's XPLAIN system [15] can be summarized for our purposes as follows. It has a component called Domain Principles, which is best thought of as a base of *control abstractions* of the goal-subgoal type. They are of the form, "If goal is G, and if <pattern1>, ... <patternN> occur in the domain knowledge base, set up subgoals SG1, ... SGN respectively." As a concrete example, G might be "Administer <drug>," pattern1 might be, "<finding> and <drug> cause <bad side effect>," and SG1 might be, "Control toxicity of <drug>." One can imagine an instructor teaching a group of students about administration of drugs in general, and telling them that if, for a particular drug, there is a possibility of a bad side effect, then make sure to do whatever will be needed to control the drug toxicity. Note that this has some degree of generality in that it can be used to set up systems for a number of different drugs: if a certain drug does not cause bad side effects, then this particular subgoal will not be set up by the system. In general one can best think of this approach as specification of an *expert system generator*, in that the same Domain Principles base can be used to generate, e.g., systems to recommend the administration of different drugs. The Domain Principles then can be thought of as a collection of control abstractions. However, these control abstractions are domain-specific. Terms such as *administer* and *control toxicity* in the example above are used to index and name goals, but do not have general purpose problem solving relevance across domains. The only elements in the above example that *are* generic in our sense are, *If goal*, and *set up subgoal*...

As one would expect, the basis for the explanation capability of XPLAIN arises from the goal-subgoal control abstractions in Domain Principles. The generation of explanation in XPLAIN is very similar to that in rule-based systems in that the goal-subgoal structure in Domain Principles is used for the explanation in a way very similar to the rule-tracing in backward-chaining systems such as Mycin. While explanation in Mycin is done using the trace of the rules that fired in a particular problem, XPLAIN uses the goal-subgoal relationships that went into the construction of the expert system, with very similar effects. XPLAIN can use the names of the goals and subgoals and the terms in the patterns to provide a richer quality to the explanation: "Because goal is to *administer* digitalis, and *digitalis causes dangerous side effects*, there is a need to *control toxicity of digitalis*."

Where our work differs from this effort is in the power that is available in the control abstractions that are indexed by generic tasks. This enlarges the kinds of explanations that can be provided in a domain-independent way, and that can arise directly from the control behavior in the problem solving process.

Acknowledgment: The paper has benefited from the comments of Tom Bylander, Jon Sticklen and John Josephson.

REFERENCES

[1] Brown, D.C. / Chandrasekaran, B.
Expert Systems for a Class of Mechanical Design Activity.
1984
Paper for IFIP WG5.2 Working Conference, Sept. 84.

[2] Buchanan, B. / Sutherland, G. / Feigenbaum, E.A.
Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organcic Chemistry.
1969
in Machine Intelligence 4, American Elsevier, New York.

[3] Bylander, T. / Mittal, S. / Chandrasekaran, B.
CSRL: A Language for Expert Systems for Diagnosis.
In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 218-221. , August, 1983.
To appear in the Special Issue of *Intn'l Jrnl. of Computers and Mathematics* on "practical artificial intelligence systems".

[4] Chandrasekaran, B. / Mittal, S. / Gomez, F. / Smith M.D., J.
An Approach to Medical Diagnosis Based on Conceptual Structures.
Proceedings of the 6th International Joint Conference on Artificial Intelligence :134-142, August, 1979.
IJCAI79.

[5] Chandrasekaran, B. / Mittal, S.
Conceptual Representation of Medical Knowledge for Diagnosis by Computer: MDX and Related Systems.
In M. Yovits (editor), *Advances in Computers*, pages 217-293. Academic Press, 1983.

[6] Chandrasekaran, B.
Expert Systems: Matching Techniques to Tasks.
1983
Paper presented at NYU symposium on Applications of AI in Business. Appears in *Artificial Intelligence Applications for Business*, edited by W. Reitman, Ablex Corp., publishers.

[7] Chandrasekaran, B.
Towards a Taxonomy of Problem-Solving Types.
AI Magazine 4(1):9-17, Winter/Spring, 1983.

[8] Clancey, William J.
The Epistemology of a Rule-Based Expert System--a Framework for Explanation.
Artificial Intelligence 20(3):215-251, May, 1983.

[9] Clancey, William J.
The Advantages of Abstract Control Knowledge in Expert System Design.
In *Proceedings of AAAI-83*, pages 74-78. Amerian Association for Artificial Intelligence, 1983.

[10] Hasling, Diane Warner / Clancey, William J. / Rennels,Glenn.
Strategic Explanations for a Diagnostic Consultation System.
In Coombs, M. J. (editor), *Developments in Expert Systems*, pages 117-133. London and New York: Academic Press, 1984.

[11] Josephson, John R. / Chandrasekaran, B. / Smith, J.W.
Assembling the Best Explanation.
In *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems.* IEEE Computer Society, Denver, Colorado, December 3-4, 1984.
A revised version by the same title is now available.

[12] McDermott, J.
R1: A Rule-Based Configurer of Computer Systems.
Artificial Intelligence 19, 1:39-88, 1982.

[13] Pople, H. W.
Heuristic Methods for Imposing Structure on Ill-Structured Problems.
In P. Szolovits (editor), *Artificial Intelligence in Medicine*, pages 119-190. Westview Press, 1982 .

[14] Shortliffe, E.H.
Computer-based Medical Consultations: MYCIN.
Elsevier/North-Holland Inc., 1976.

[15] Swartout, W. R.
XPLAIN: A System for Creating and Explaining Expert Consulting Programs.
Artificial Intelligence 21(3):285-325, September, 1983.

[16] Szolovits, P. / Pauker, S. G.
Categorical and Probabilistic Reasoning in Medical Diagnosis.
Artificial Intelligence :115-144, 1978.

Representing Actions with an Assumption-Based Truth Maintenance System

Paul H. Morris
Robert A. Nado

IntelliCorp
1975 El Camino Real West
Mountain View, California 94040

ABSTRACT

The Assumption-based Truth Maintenance System, introduced by de Kleer, is a powerful new tool for organizing a search through a space of alternatives. However, the ATMS is oriented towards inferential problem solving, and provides no special mechanisms for modeling actions or state changes. We describe an approach to applying the ATMS to the task of representing contexts that model actions. The approach extends traditional tree-structured context mechanisms to allow context merges. It also takes advantage of the underlying ATMS to detect inconsistent contexts and to maintain derived results. Some results are presented concerning possible approaches to the treatment of merges in questionable circumstances. Finally, the analysis of actions in terms of a truth maintenance system suggests the need for a more elaborate treatment of contradiction in such systems than exists at present.

1. Introduction

The Assumption-Based Truth Maintenance System (ATMS), introduced by de Kleer [2], is a powerful new tool for organizing an efficient search through a space of alternatives. By explicitly recording the dependence of reasoning steps on individual choices, a truth maintenance system is able to share partial results across different branches of the search space. In effect, knowledge gleaned in one context is automatically transfered to other contexts where it is relevant. The ATMS permits simultaneous reasoning about multiple, possibly conflicting contexts, avoiding the cost of context switching.

The ATMS as presently constituted views problem solving as purely inferential. This is an appropriate stance for a broad class of constraint satisfaction problems. However, problems involving temporal changes or actions require some additional mechanism. As de Kleer [5] points out, "... problem solvers [may] act, changing the world, and this cannot be modeled in a pure ATMS in which there is no way to prevent the inheritance of a fact into a daughter context." In this paper we explore one approach to using the ATMS to support the modeling of actions. The basic idea is to extend a traditional tree-structured context mechanism (as in CONNIVER and QA4 [1]) to allow context merges and to take advantage of an underlying ATMS to detect inconsistent contexts and to maintain derived results. This approach has been implemented in the KEEworlds^{TM} facility of the KEE^{TM} (Knowledge Engineering Environment^{TM}) system.[1]

In the following sections, we give a functional overview of the KEEworlds facility. We then describe the underlying representation in terms of the ATMS. Special attention is given to the situation where a world has multiple parents. This is followed by a discussion of non-monotonic reasoning about actions in a more general TMS setting, suggested by the worlds mechanism. We close with some remarks about related systems.

2. Worlds

The basic structure provided for modeling actions is a directed acyclic graph of *worlds*. Each world may be regarded as representing an individual, fully specified action or state change. A world together with its ancestors in the graph represents a partially ordered network of actions. Each successor of a world in the graph then represents a hypothetical extension of the world's associated action network to include a new subsequent action. The world graph as a whole may thus be regarded as representing multiple, possibly conflicting, action networks. Each partially ordered action network resembles a procedural net of NOAH [9], or NONLIN [10], where the actions are fully specified. We assume that the effects of a fully specified action can be represented by additions and deletions of base facts, so each world has a set of additions and deletions associated with it

[1] KEEworlds, KEE and Knowledge Engineering Environment are trademarks of IntelliCorp.

which represent the actual primitive changes determined by the action. Since an action corresponds to an *application* of an operator, not an operator itself, this assumption is somewhat less restrictive than that of STRIPS [8] in that it imposes fewer constraints on the representation of the operators. Figure 2-1 shows an example worlds graph, from the blocks world. The deletion and addition at W2, for example, represents the movement of block *a* to the table.

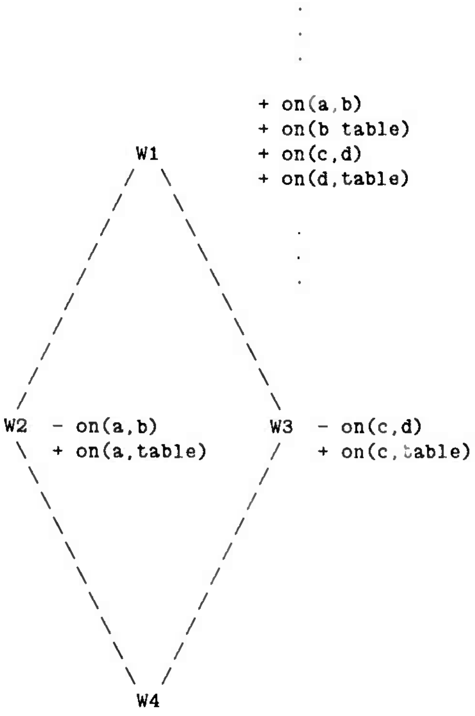


Figure 2-1: Worlds Graph

To simplify the discussion we will assume for the moment that the graph is a tree, i.e., each world has at most one parent and a branch of the tree corresponds to a linear sequence of actions. Later, we will consider the consequences of multiple parents.

Observe that we may associate each world with the state that results from applying the changes encoded by the world and all of its ancestors. Hence, a world plays a double role, representing both a state change and a state. The facts in the state will in general be augmented with deductions using general knowledge of the domain. Thus, the facts which are true at a world fall into the following three categories:

1. facts inherited from ancestor worlds

2. direct additions at this world

3. deductions from facts in 1 and 2

In keeping with the view that additions and deletions represent actual changes, they are only recorded where they are *effective*, that is, an addition only occurs where the fact did not previously hold, and a deletion where it did hold.

The inherited facts follow a principle of inertia (essentially the STRIPS assumption [11]): a fact which is added at a world continues to be true in succeeding worlds, up until (but not including) a world where it is deleted.

The deduced facts may include the distinguished fact FALSE, representing a contradiction. A world where FALSE can be deduced is marked as inconsistent. The system generally avoids further reasoning in such worlds (however, it is possible and sometimes useful to do meta-level reasoning about inconsistent worlds).

3. Worlds in ATMS

Before discussing how the worlds graph is implemented in terms of the underlying ATMS, we give a brief sketch of the ATMS mechanisms that are used, primarily to establish terminology. The reader is urged to consult de Kleer [3, 4, 5] for a full description of the ATMS.

The basic elements of the ATMS are *assumptions* and *nodes*. An assumption in the ATMS corresponds to a decision or choice, and is used as an element of context descriptor. Nodes correspond to propositional facts or data, which may be justified in terms of other nodes, or assumptions. By tracing back through the justification structure, it is possible to determine the ultimate support for a derivation of a node as a set of assumptions. Such a set is called an *environment* for the node. Since a node may have multiple derivations, it may also have multiple environments. The set of (minimal) environments for a node is called its *label*. Computing the labels of nodes is one of the major activities of the ATMS The primary transaction that the ATMS supports is adding a justification. This causes the labels of affected nodes to be recomputed. There is a special element called FALSE, denoting contradiction, which is similar to a node, and may have justifications. The environments that would be in its label are called *nogoods* and constitute minimal inconsistent environments. Environments which are discovered to be inconsistent, i.e., which are supersets of nogoods, are removed from the labels of nodes so that they are not used for further reasoning.

Each world has two ATMS entities associated with it, reflecting its double role: a *world assumption* and a *world environment*. The world assumption corresponds to the action encoded by the world, and may also be thought of as the choice or decision that led to the action. The world environment, on the other hand, corresponds to the state, and actually consists of the set of world assumptions from the given world and all of its ancestors. It is convenient to use the ATMS itself to compute the world environment. This is accomplished by having a special *world node* associated with each world. This node may be thought of as representing the statement that the world's action occurs. The world node, N_W, is given a single justification

$$N_{WP} \wedge A_W \rightarrow N_W$$

where N_{WP} is the world node of the parent, and A_W is the world assumption of the given world. It is not difficult to see that this results in all world nodes having a single environment, of the form described.

Adding a fact F at a world can now be accomplished by supplying a justification in terms of the world node. However, to allow for the possibility of later deletion, a ***nondeletion*** assumption is included. Thus, the justification has the form

$$N_W \wedge A_{W,F} \rightarrow F$$

where $A_{W,F}$ is the nondeletion assumption. A distinct nondeletion assumption is required for each separate addition of a fact at a world (to allow independent deletion). If F is deleted at a subsequent world W1, the justification

$$A_{W1} \wedge A_{W,F} \rightarrow FALSE$$

is supplied to the ATMS, where A_{W1} is the world assumption for W1. We will call nogoods resulting from justifications of this form *deletion* nogoods.

Apart from the justifications supplied by the system to represent additions and deletions, and justifications for world nodes, there will be justifications installed by the user to represent deductions from the primitive facts. These deductions need be performed only once as the presence of the justifications in the ATMS allows the efficient determination, via label propagation, of which derived facts hold in which worlds.

Derivations of FALSE are used to determine inconsistent worlds, representing dead ends in the search. The nogoods determined by the ATMS may, however, contain nondeletion assumptions in addition to the world assumptions. However, only the latter represent choices in the search, and we wish these to take all the "blame" for dead ends (we discuss this further in section 5). Thus, the multiple worlds system incorporates a feedback loop which installs in the ATMS reduced nogoods with the nondeletion assumptions removed. These nogoods are subsets of the original ones, and so, in accordance with the minimality requirement, the latter are removed. This process ensures that the deletion nogoods are the only ones containing nondeletion assumptions.

To test whether a fact holds in a world, we can compare each environment in the node label with the world environment. The comparison is done as follows (in principle; the actual algorithm is equivalent, but more efficient). The world environment is extended with as many nondeletion assumptions as are consistent with it (the extension is necessarily unique since each nogood contains at most one nondeletion assumption). The extended world environment is then checked to see if it is a superset of the fact environment. If so, the fact is regarded as true in the world.

4. Merges

We now consider the more complex situation where a world has multiple parents: we call such a world a *merge*. The ability to perform merges allows a problem to be decomposed into nearly independent components, which can be worked on separately and later recombined. As before, the changes represented by the ancestor worlds are combined. In the example of figure 2-1, the world W4 is merge. Thus, the state corresponding to W4 will have both blocks moved to the table. We wish to stress that a merge is not the same as a simple union of the facts in the parent worlds, but rather combines the *changes* from all the ancestor worlds.

In the example of figure 2-1, the changes along the two branches are independent. More generally, a difficulty arises in that the effect of changes may depend on the order in which they are applied, resulting in an ambiguous merge. In figure 4-1, we show two examples of such merges. In both cases, the state at W5 depends on the order of the preceding changes.

There are a number of ways of dealing with this difficulty. We have already introduced the requirement that additions and deletions at worlds be effective with respect to the state resulting from actions in ancestor worlds. However, from a strict standpoint of fully specified actions, the additions and deletions could be required to be effective even with respect to actions in sibling or cousin worlds. Thus, one might forbid a merge if the ancestor subgraph of the proposed merge possesses any linearization in which an addition or deletion is ineffective. One can then prove the following result.

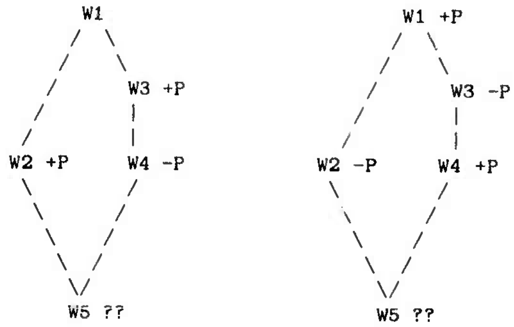


Figure 4-1: Ambiguous Merges

> **Theorem 1:** A merge that is not forbidden by the above criterion is unambiguous.

It is also possible to prove the following result, which assists in the identification of such forbidden merges.

> **Theorem 2:** A graph of worlds admits a linearization in which an addition is ineffective if and only if there are at least two worlds where the addition occurs, such that neither is an ancestor of the other.

A similar result holds for deletions. With this approach, the merges in figure 4-1 would be disallowed.

It is of interest that the above restriction resembles that required for conflict-free procedural nets [10] where actions that violate each others' preconditions must be ordered so that one is an ancestor of the other. Indeed, additions and deletions which are mandatory are, in effect, preconditions. From this perspective, the separate branches of the networks of figure 4-1 are in conflict because each branch deletes a precondition of the other.

If one does not require that additions and deletions be effective with respect to non-ancestor actions, a weaker condition which guarantees unambiguous merges is as follows:

> **Theorem 3:** A sufficient condition for a merge to be unambiguous is that the ancestor subgraph may not contain two worlds, one of which deletes a fact and the other of which adds it, such that neither is an ancestor of the other.

This criterion also prohibits the examples of figure 4-1.

Another approach to removing the ambiguity is to adopt additional criteria for defining the merge. In the *pessimistic merge*, an individual fact belongs to the merge if it survives in every linearization of the actions. The rationale is that we may then be assured the fact holds, irrespective of the order in which the actions were performed. Otherwise, we are ignorant of the fact, and the absence of the fact from the merge simply denotes such ignorance, not falsity. Notice that when the effect of the actions *is* order independent, this definition reduces to the previous one. With the pessimistic merge, the fact P is absent at W5 in both examples of figure 4-1. A dual to the pessimistic merge is the *optimistic merge* where a fact is true in the merge if it is true in *some* linearization. Again, this reduces to the original merge in the case of order independence. With the optimistic merge, P is present at W5 in both examples.

We [illegible]uss the ATMS representation for merges. When a w[illegible] multiple parents, the justification for the world node includes each of the parent world nodes among the justifiers. The justification scheme for additions and deletions works as before. The different merges are obtained by different selections of which additions the deletions affect, i.e., which justifications for FALSE are entered. For the pessimistic merge, the deletions are effective with respect to all except descendant additions. For the optimistic case, the deletions are effective with respect to ancestor additions only (the optimistic merge tends to be easier to implement efficiently, although less defensible on semantic grounds).

One might imagine a wide variety of possible merge algorithms. There are two overriding constraints that led to the schemes described here. One is the necessity of quickly determining whether a potential merge would produce a consistent world, since that is expected to be a high frequency operation. The schemes described allow the merge to be computed as a simple union of ATMS environments. The other constraint is the existence of a large core of unambiguous cases where there is only one reasonable value for the merge.

A further merge type which has some intuitive appeal, but does not appear to admit an efficient implementation, arises as follows. It is possible to show that every linearization of the ancestor subgraph in which all additions and deletions are effective gives the same result for the merge. Thus, one might define the merge to be this common value (if there is any such linearization). In figure 4-1, this would lead to P holding at W5 in the left example, but not in the right.

5. Actions and NonMonotonicity

It is instructive to consider how actions might be represented in a more general TMS setting, as suggested by the worlds system. For definiteness, and for contrast, this will be cast in terms of a Doyle-style truth maintenance system [6]. The general approach we follow is to use a form of nonmonotonic inference to reason about the effects of actions. However, the behavior we require in response to contradiction is somewhat different from the standard approach in truth maintenance systems.

We will regard a context, or current state of the system, as describing the evolution of a situation to a particular point in time. Besides containing assertions about facts in the "present" such as "block *a* is on block *b*," the context records past actions like A3: "block *a* was placed on block *b*". Note that there may he several occurrences of individual actions with the same description; we distinguish between the occurrences by giving them unique identifiers such as A3. The numbering of the identifiers is not intended to imply temporal order. Thus - so far - the relative timing of past actions has not been represented.

The positive effects of an action can be represented by justifications linking the occurrence of past actions to present facts. For example,

$A3 \wedge P5 \rightarrow$ block *a* is on block *b*.

P5 is a preservation condition of the form "block *a* was not moved off block *b* after A3." In order to allow deletion, we justify P5 as an assumption by giving it a nonmonotonic justification of the form

$(D5) \rightarrow$ P5

Here, "(D5)" indicates that D5 is an OUT-justifier, where D5 is the statement that "some action after A3 moves block *a* off block *b*". If a subsequent action, say A4, moves the block off, we supply a justification

$A4 \rightarrow D5$

causing the OUT-justifier to come IN, thereby undercutting the derivation of "block *a* is on block *b*." Note that the information about the relative timing of actions is now implicitly represented by these justifications.

A difficulty with this representation arises when the problem solving process generates contradictions that represent dead ends in the search space. We do not wish the preservation assumptions to be implicated in these; rather, we wish the assumptions representing choices of actions to be the ones considered for revision. Choosing a preservation assumption as culprit during backtracking would amount to postulating the existence of an unknown action that deletes one of the facts leading to the contradiction. However, if we make the separation between problem solving and truth maintenance suggested by de Kleer, then from the point of view of the TMS, the only actions which exist are those which the problem solver has informed it about. Some new mechanism is required to ensure that the TMS handles this correctly. One possibility is to have something like a "sheltered" assumption, which could be refuted directly, but not indirectly in response to a contradiction.

Incidentally, the need for a more discriminating process of culprit identification is not confined to the difficulty with preservation assumptions. As another example, consider a situation where a burglar is planning to break into a house late at night. To accomplish his purpose, he must choose some method of entry. One method is to break in a window. However, this may have the consequence of waking the occupants, if they are home, which would defeat his purpose. Let us suppose the burgler makes the default assumption that the occupants are home. The difficulty is that a standard truth maintenance system, in attempting to resolve the "contradiction" of waking the occupants, might elect to revise the assumption that the occupants are home, even though that is not subject to the burglar's control, instead of the real culprit, breaking the window. The system would in effect regard the undesired consequence of waking the occupants as evidence for their absence. However, it is only when there is independent evidence for the occupants being absent that this possibility is worth considering. This example of "wishful thinking" suggests that truth maintenance systems in general need a more refined treatment of contradiction handling.

Although the approach outlined here could be adapted to using the ATMS more directly for modeling actions, it would be cumbersome for a user to have to input the justifications representing additions and deletions by hand. The worlds facility described earlier provides a framework which represents a more convenient interface to an action modeling system.

6. Closing Remarks

The worlds considered here resemble the data pools of McDermott [7]. However, the result of a merge in the data pool approach is determined by the arbitrary order in which items are added and deleted in worlds (*beads* in McDermott's terminology). This means that two graphs with the same apparent external structure may have different results for a merge. Another difference is that data pools apparently have no notion of contradiction. One attractive aspect of McDermott's approach is that justifications may have OUT-justifiers.

The Viewpoints^{TM} facility of Inference Corporation's ART^{TM} system appears quite similar in behavior to the worlds facility described here.[2] However, it is difficult to make detailed comparisons since little information has been made available about the underlying mechanisms of ART.

We have described an approach to constructing a context mechanism that represents a partially ordered net, ork of actions or state changes. A realization of the mechanism has been described in terms of an underlying Assumption Based Truth Maintenance System. An examination of a similar representation in a classical TMS system suggests a shortcoming in the way existing truth maintenance schemes handle contradictions.

The approach described has been implemented as part of the KEEworlds facility of KEE and appears to provide a useful and efficient tool for reasoning about multiple situations. The KEEworlds facility integrates the multiple worlds system with an existing frame-based representation system, provides a graphical browser for manual exploration of worlds and allows rule-based generation of worlds during either forward or backward chaining.

References

[1] Bobrow, G. and B. Raphael.
New Programming Languages for Artificial Intelligence Research.
Computer Surveys 6(3):153-174, 1974.

[2] de Kleer, J.
Choices Without Backtracking.
In *Proceedings, AAAI-84*. Austin, Texas, 1984.

[3] de Kleer, J.
An Assumption-Based Truth Maintenance System.
Artificial Intelligence 28(1), 1986.

[4] de Kleer, J.
Extending the ATMS.
Artificial Intelligence 28(1), 1986.

[5] de Kleer, J.
Problem Solving with the ATMS.
Artificial Intelligence 28(1), 1986.

[6] Doyle, J.
A Truth Maintenance System.
Artificial Intelligence 12(3), 1979.

[7] McDermott, D.
Contexts and Data Dependencies: A Synthesis.
IEEE Transactions on Pattern Analysis and Machine Intelligence 5(3):237-246, May, 1983.

[8] Nilsson, N.J.
Principles of Artificial Intelligence.
Tioga Publishing Company, Palo Alto, Ca., 1980.

[9] Sacerdoti, E.D.
A Structure for Plans and Behavior.
Elsevier North-Holland, 1977.

[10] Tate, A.
Generating Project Networks.
In *IJCAI-77*, pages 888-893. Cambridge, Massachusetts, 1977.

[11] Waldinger, R.J.
Achieving Several Goals Simultaneously.
In Elcock, E. and Michie, D. (editor), *Machine Intelligence 8*, pages 94-136. Ellis Horwood, Chichester, 1977.

This research was supported in part by the Defense Advanced Research Projects Agency under contract No. F30602 85 C 0065.

[2] Viewpoints and ART are trademarks of Inference Corporation

CAGE and POLIGON: Two Frameworks for Blackboard-based Concurrent Problem Solving

H. Penny Nii

Knowledge Systems Laboratory
Computer Science Department
Stanford University

The two articles following this one, *User-Directed Control of Parallelism: The CAGE System* and *POLIGON: A System for Parallel Problem Solving*, describe two different skeletal systems representing two models of concurrent problem solving. Both systems are designed for parallel execution of application programs built with the systems. This paper describes the context in which these systems are being developed and summarizes the differences between the two systems.

The Context

The POLIGON and the CAGE systems are being developed within the context of two different families of experiments within the Advanced Architectures Project. Each family of experiments consists of a vertically integrated set of programs from each level of system hierarchy outlined in the project proposal (i.e. application, problem-solving framework, knowledge representation and retrieval, implementation language, and hardware/system architecture levels). POLIGON and CAGE are two systems at the problem-solving framework level. The design of both the POLIGON and the CAGE systems are based on the Blackboard problem solving model [4].

The Experiments

Each family of experiments starts with a different set of high-level constraints:

Hardware/system architecture: The POLIGON system is designed for distributed-memory, multi-processor systems. It assumes that the underlying system has a large number (100's to 1000's) of processor memory pairs with very high bandwidth inter-processor communication. The CAGE system, on the other hand, assumes a shared-memory, multi-processor system with tens to hundreds of processors. The underlying system architecture influences the additional constructs at the programming language level needed to support parallel executions. It also has significant affect on the design of blackboard frameworks.

Control of parallelism: The POLIGON system is designed with an assumption that the underlying problem solving framework on which the application is to be mounted must be intrinsically parallel. The POLIGON system is designed so that predefined constructs in the framework always run in parallel. For example, all rules are evaluated in parallel and all changes to blackboard nodes are made in parallel. The user has some ability to introduce serialization. CAGE, on the other hand, assumes that the user needs control over what is to run in parallel. Thus, everything in CAGE runs serially unless specified otherwise by the user. There are prespecified places where the user can introduce parallelism. For example, the user can specify that the condition parts of rules be evaluated in parallel and the action parts be executed in series.

The family of experiments of which CAGE is a part consists of CAGE (problem solving framework) implemented in Qlisp [2] (implementation language) running on a shared-memory architecture (system architecture) simulated on CARE [1] (system simulator). The other family of experiments consists of POLIGON (problem solving framework) implemented in CAOS [5] and Zetalisp (implementation language) running on a distributed-memory architecture (system architecture) simulated on CARE. Both CAGE and POLIGON run on the same system simulation program and share its software measurement tools. Both skeletal systems will mount the same application problems.

In keeping with the goals of our Project, the primary objective of the two families of experiments is to discover methods that would speed up the execution of knowledge-based application programs. There are, however, additional reasons for the two experiments that relate to the primary objective:

To compare the performance gains between shared versus distributed-memory, multiprocessor systems.

[1] This research was supported by DARPA/RADC (F30602-85-C-0012), by NASA (NCC 2-220), and by Boeing Computer Services (W-266875).

To provide input to the implementation language level (QLisp, CAOS and other concurrent Lisp languages);

To gain some understanding of the differences in 'programmability' between POLIGON and CAGE. More specifically, address the question of whether it is easier/better to let the user have complete control over the parallelism in a program; and as a corollary, to determine the limits of concurrency that can be designed into a framework, and the kinds of concurrencies that are problem specific and need to be expressed by the user.

To determine the extent of control, or serialization, needed in both systems in order to solve a class of problems, and to discover how to apply the needed control.

To determine if multiplicative speed-up can be effected between knowledge sources, rules, and lower level (for example, rule clause evaluation) concurrencies.

To determine what level of process granularity is most appropriate for each hardware/systems architecture.

Comparison of the CAGE and POLIGON Systems

CAGE and POLIGON are concurrent blackboard systems with two different underlying design philosophies. CAGE is an extension of the AGE [3] system with primitives to express parallel execution of knowledge sources, rules, and parts of rules. It is a conservative, incremental approach to building parallel systems. POLIGON is a demon-driven system in which all blackboard nodes are viewed as active agents (and thus each blackboard node can potentially be a processor/memory pair). A change made to a node causes appropriate rules to be evaluated and executed. POLIGON represents a shift in the way we view blackboard systems.
Both systems have programming languages associated with them, the POLIGON language and the CAGE language. The first objective in providing a language at the problem solving level is to facilitate the writing of application programs. This is accomplished by abstracting much of the system detail into language constructs. The second objective is to keep separate the parallelism in the application problem, as expressed by the language, and the parallelism built into the framework that remain invisible to the user. This separation allows us to experiment with parallelism in the application program independent of experiments with parallelism within the framework. Thus, we can for example, keep the application constant and change the parallel constructs within the framework, or keep the framework constant and rewrite the application. In order to facilitate the porting of an application program between POLIGON and CAGE, both languages are syntactically similar. However, the semantics of the languages are very different because the underlying systems are very different. The differences are summarized below.

CAGE	POLIGON
Incremental additions of parallelism to a serial system	Redesigned parallel system
User controlled parallelism	User controlled serial operations
Granularity of parallelism under user control	Granularity of parallelism fixed - rules and actions
Shared memory multi-processor machines	Distributed memory multi-processor machines

Figure 1: Summary of Differences: CAGE and POLIGON

We now describe and discuss some of the issues specific to the CAGE and the POLIGON systems. The discussions should serve as a background to the detailed description of the systems in the separate papers.

CAGE

There are several obvious places for concurrency in blackboard systems, the knowledge sources, rules within the knowledge sources, and the components of the rules.

Knowledge Source concurrency: Knowledge sources are logically independent partitions of domain knowledge. Each knowledge source is event-driven and becomes active when changes relevant to the knowledge source are made to the blackboard. Theoretically, therefore, all knowledge sources can be active at the same time as long as events relevant to each of the knowledge sources occur at the 'same time'. However, knowledge sources are often serially dependent in order to solve a problem. At run time some synchronization (i.e. serialization) must be enforced.

In the class of applications we are considering, the solution generation process characteristically occurs in a pipeline fashion up the blackboard hierarchy. That is, the knowledge source dependencies form a chain from the knowledge sources working on the most detailed level of the blackboard to those working on the most abstract level. When the program is model-driven, the pipeline works in the reverse direction. The task for CAGE in exploring concurrency at this level of granularity is to determine what percentage of the knowledge sources can be active at the same time in the pipe.

Rule concurrency: Each knowledge source is composed of many rules. The condition part of the rules are evaluated for a non-NIL condition (a match) and the action part of those rules that match are executed. The condition-part of all the rules in a knowledge source can be evaluated in parallel. In those cases where the action part of all the rules that match are to be executed, the action part can be executed as soon as the match is completed. However, if only one of the rules is to be fired (single-hit), then the system must wait until all the condition parts are evaluated, and one rule must be chosen whose action part will be executed. (Note that this is very similar to the OPS conflict-resolution phase.) In addition, one can imagine evaluating all of the condition parts in parallel and executing the appropriate action parts in series.

The situation in which all rules are evaluated and fired concurrently will result in the most speed-up, since many rules will be in the state of being evaluated and being executed at the same time. However, if the rules need access to the same blackboard item, memory contentions become a hidden point of serialization. At the same time, the integrity of information on the blackboard cannot be guaranteed. The condition which triggered the action part of the rule may not be the same by the time it is executed. CAGE needs to address these problems, determine the effect on solution quality and overall performance gain of the application program.

Condition-part concurrency: Each condition part of a rule consists of many clauses to be evaluated. These clauses can be computed in parallel. Often these clauses involve relatively large numeric computation (e.g. calculating a track), making parallel clause evaluation worthwhile. On the other hand, often the clauses refer to the same data item, making the clause evaluation appear to be parallel, but in fact forcing serialization at the data-access level with no gain (and most likely a loss) in speed of computation. The task at this level of granularity is to determine if parallelism at this level is worthwhile. It may be that what is needed at this level is a fast algorithm for matching the condition parts and an appropriate knowledge representation scheme.

Action part concurrency: Often, when a condition part matches, there are many actions to be executed. This is one place where no difficulty is anticipated in parallel execution.

Combining the concurrencies: The action parts of rules generate events, and the knowledge sources are activated by occurrences of these events. In the AGE system events were posted on an event-list and a control monitor invoked the knowledge sources based on those events. In order to eliminate the serialization inherent in this control scheme, a mechanism to activate the knowledge source upon the completion of the action parts of rules is needed. The immediate activation of a knowledge source after action part execution (for example, by broadcasting an 'event message' to all the knowledge sources) results in the loss of global control over knowledge source activation. In some cases, this is acceptable. In other cases, for example when knowledge sources need to be activated on a priority basis (exemplified by the need for the Agenda mechanism in AGE), some control mechanism is needed. The task here is to determine the best (least overhead) control mechanism appropriate to the application.

POLIGON

As mentioned earlier, the application programs are event-driven in blackboard systems. Events are normally defined by the user and expressed as changes to the blackboard nodes. Because a knowledge source is activated by the occurrences of events, and because knowledge sources are collections of rules, one can view the rules as being activated (indirectly) by changes to some blackboard nodes. We can take this line of reasoning one step further and say that a rule is activated by changes to particular slots of blackboard nodes. If we associate a set of rules directly with a slot on a node and evaluate and execute the rules whenever the slot is changed, we have a system with active blackboard nodes.

Conceptually, at least, every blackboard node can be thought of as a processor-memory pair. Each node contains a data structure to store the partial solutions, and the rules are activated whenever a particular slot is changed. Slots with a property that enable rule triggering are called "trigger slots". When the action part of a rule is executed, the changes to the blackboard are made via messages to the nodes to be changed. If the change to is to a trigger slot, then the condition part of the "triggered rules" are evaluated; changes to non-trigger slots do not cause processing.

A major difficulty with this approach is the loss of control, specifically, an ability to control the order of rule firing. By bypassing the intermediate control step where manipulation of the events and selection of knowledge sources occurs, the system has no global control. The rules will be firing almost indiscriminately all over the blackboard as solution state changes. There is no way to implement problem solving strategies, for example. In addition, rules will not be evaluated in situations when the *non-occurrence* of a change to the blackboard is significant. Such ability is important in signal interpretation programs.

In spite of many anticipated difficulties, we have developed a demon-driven system in hopes of gaining experience with such a system and discovering solutions to the problems. Although there is a substantial shift in the problem solving behavior, POLIGON is being evolved out of the functionalities that were present in AGE. At this point POLIGON is characterized by the following:

> Knowledge sources exist only as a conceptual aid in partitioning the problem space.
>
> Levels of in the blackboard data exist as a class hierarchy. A level is a class and a node is an instance of a class. There is also a super-class that knows about the classes. (For clarity, the class will be referred to a more familiar term, the level.)

All nodes are active entities.

Each rule must specify, in addition to the condition and action parts, the level and the node with which it is to be associated, i.e. it must designate a 'trigger'. A trigger consists of a slot name and a trigger-condition, which are to be interpreted as follows: whenever the value of the slot is changed, evaluate the trigger condition. If the trigger condition is non-nil then the rule becomes triggered. A triggered rule is put on a process queue for later evaluation.

The rules can use data futures, and for the time being all bindings are made through lazy evaluation. This means that all bindings are made only when needed. In addition, processing can continue while values are being fetched from other nodes.

The major control problem to be addressed in demon-systems is the serialization of demon activations. Potential for control in POLIGON exists in three places: (1) On the node, where action parts of the rules can be serialized, for example. (2) In the level manager, which knows about the all the nodes on the level. (3) In the super-manger which knows about all the level managers. The level manager that can create and garbage collect the nodes, and knows which rules to attach to a newly created node. The level manager is the only agent that knows about all the existing nodes on its level. Thus, to send a message to all the nodes on a particular level, a message is sent to the level manager which forwards it to all its nodes.

In addition to the parallel evaluation of the condition parts of rules, the actions in the action part of the rules are executed in parallel.

Because of POLIGON's uncontrolled parallelism the solution to a problem will be indeterminate. That is, every execution of an application problem can potentially result in different answers. The challenge is to organize the knowledge in such a way that "acceptable" solutions are produced each time.

Most of the same concurrencies made available to the user in CAGE are built into the system in POLIGON. The major challenge in POLIGON is the serialization of rule execution. For example, the ability to synchronize the execution of actions in CAGE has no counterpart in POLIGON. Since the system is demon-driven at the rule level, there are very few handles available to control the activation of rule evaluation.

Summary

CAGE and POLIGON thus are two very different approaches to the expression of parallelism at the problem solving framework level. As we develop and test applications using these frameworks, we expect to gain a more concrete understanding of their relative strength and weaknesses with respect to usability, application characteristics, and speedup. Each system is discussed in more detail in the following two articles.

References

[1] Bruce Delagi.
CARE Users Manual.
Technical Report KSL-86-36 (working paper), Knowledge Systems Laboratory, 1986.

[2] Gabriel, R.P. and J. McCarthy.
Queue-Based Multi-Processing Lisp.
In *Proceedings of the 1984 Symposium on Lisp and Functional Programming.* August, 1984.

[3] H. Penny Nii and Nelleke Aiello.
AGE: A Knowledge-based Program for Building Knowledge-based Programs.
Proc. of IJCAI 6 :645 - 655, 1979.

[4] H. Penny Nii.
Blackboard Systems.
Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April, 1986.
To appear in *AI Magazine*, vol. 6-6 and vol. 6-7, 1986.

[5] Eric Schoen.
The CAOS System.
Technical Report KSL-86-22, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April, 1986.
Also in this Proceedings.

User-Directed Control of Parallelism; The CAGE System

Nelleke Aiello

Knowledge Systems Laboratory, Stanford University

I INTRODUCTION

CAGE*, Concurrent AGE**, provides a framework for building and executing application programs as a concurrent blackboard system. With CAGE, the user can control which parts of the blackboard system are executed in parallel. A blackboard application can be implemented and debugged serially on CAGE. Once the serial version is debugged, concurrency can be introduced to different parts of the system, allowing the user to experiment with various configurations. We believe this incremental approach will facilitate the construction of concurrent problem solving systems and will teach us much about programming in a parallel environment. This paper describes the design of the CAGE system and gives detailed instructions for implementing an application, using the CAGE language and compiler [Rice 86]. We have included advice, warnings, and caveats based on our experience using CAGE.

The target parallel system architecture for the CAGE system is currently the same as that of QLAMBDA, a queue-based multi-processing Lisp ([Gabriel 84]and McCarthy) on which the parallel simulation is based. We are assuming a shared memory and a large number of processors. The user can specify his CAGE application in an extension of the L100 language, called the CAGE language, and use the CAGE compiler to generate CAGE code. CAGE runs on LOQS, a functional simulator for QLAMBDA. CAGE is implemented in ZETALISP for Symbolics 3600 machines and TI Explorers.

II OVERVIEW OF CAGE DESIGN

CAGE is a blackboard framework system. In addition to the basic AGE [Nii 79] functionality, CAGE allows user-directed control over the concurrent execution of many of its contructs. The basic components of a system built using CAGE are:

1. A global data base (the blackboard) in which emerging solutions are posted. The elements on the blackboard are organized into levels and represented as a set of attribute-value pairs (a frame).
2. Globally accessible lists on which control information is posted (e.g. lists of events, expectations, etc.).
3. An indefinite number of knowledge sources, each consisting of an indefinite number of production rules.
4. Various kinds of control information that determine (a) which blackboard element is to be the focus of attention and (b) which knowledge source is to be used at any given point in the problem solving process.
5. Declarations that specify what components (knowledge sources, rules, condition and action parts of rules) are to be executed in parallel, and when to force synchronization. During the execution of the user's application CAGE will run these specified components in parallel.

Using the concurrency control specifications, the user can alter the simple, serial control loop of CAGE by introducing concurrent actions. CAGE allows parallelism ranging from concurrently executing knowledge sources all the way down to concurrent actions on the right- or left-hand-sides of the rules. The serial execution and parallel executions possible in CAGE are summarized below.

```
in KS Control
    serial: pick one event and execute associated KSs

    parallel:
        1. as each event is generated execute associated
           KSs in parallel***
        2. wait until several events are generated then
           select a subset and execute relevant KSs for
           all subset events in parallel

in KS
    serial: 1. evaluate bindings
            2. evaluate LHS then execute RHS of one rule
               whose LHS matches (in written order)
            3. evaluate all LHS then execute all RHS
               whose LHSs match

    parallel:
            1. evaluate bindings*
            2. evaluate all LHSs in parallel
                a. then synchronize (i.e. wait for all
                   LHS evaluations to complete)
                   and choose one RHS(pick one in order)
                b. then synchronize and execute the
                   RHSs serially (in written order)
                c. execute RHS as LHS matches*

in Rule
    serial: evaluate each clause then execute each action

    parallel:
        evaluate clauses in parallel then execute actions
        in parallel*
        (first nil clause --> no match; first all non-NIL
        clauses --> match)

in clause
    serial:   Lisp code          parallel: Qlambda code
```

*This research is supported by DARPA/RADC under contract number F30602-85-C-0012, by NASA under contract number NCC 2-220, and by Boeing Computer Services under contract number W-266875.

**CAGE is based on the AGE System and we have assumed here that the reader is familiar with the AGE system.

***The starred options indicate the greatest use of concurrency.

III BUILDING APPLICATIONS IN CAGE

In each of the following sections we will outline the application data that must be supplied by the user and how that information should be structured for use by the CAGE System. The CAGE System provides a CAGE language with which the user can write his application. The type of user-supplied information is similar to that required for applications constructed in the original AGE system. However, the structure of the user information is somewhat different from that of an AGE application.

A. Blackboard Data Structure

There are two major components in the CAGE blackboard structure, the hypothesis *classes* (frequently called levels in hierarchical blackboard structures) and the hypothesis *nodes*. The user must specify the classes that make up his application's blackboard structure. For each class, the user must define the fields to be associated with the nodes created in that class. Nodes are created in those classes, either a priori by the user or dynamically while executing the user's rules. The following example shows the definition of several classes and their fields in the CAGE language.

```
Class Definitions for Model "example" :

Class name-of-levela :
    attribute1
    attribute2
    attribute3
    ...

Class name-of-levelb :
    attribute4
    attribute5
    ...
```

This will compile into two macro calls, DEFHYPOTHESIS-STRUCTURE and DEFLEVEL, which the CAGE System will in turn compile into the appropriate hypothesis structure.

```
(defhypothesis-structure
  user-hypothesis-structure
  (application-system-root)
  name-of-levela
  name-of-levelb
  name-of-levelc
             ...)

(deflevel name-of-levela
  ((attribute1 nil)
   (attribute2 nil)
   (attribute3 nil)
             ...))
```

Each of the levels(or classes) will be defined as an object with the attributes as instance variables and with the nodes as instances of those objects as they are created. (The user can define methods for the level objects which are generally used for printing information contained in the nodes on those levels.)

Definitions:

user-hypothesis-structure: A name the user gives the application's blackboard structure.

application-system-root: A handle on the above hypothesis structure for user access, generally a node where the input data, or a massaged version of the input data will reside, or the top level of a hierarchical hypothesis structure.

name-of-level: Each level or class must have a user supplied name.

node: An instance of a level, created either before or during the execution of the application, inheriting all the attributes of that level, but no values.

attribute: For each level the user must specify the names of the slots, which will become a template for the instance nodes, which in turn will contain the values used by the KSs. These values are initially NIL.

link: The user may also define links for connecting nodes. These links are defined in the knowledge sources which use them and consist of a link name and an optional, opposite link. The value of a link on a node is the name of another node.

value: The value of an attribute depends on what was stored there by the rules and its structure depends on how it was stored. Values can be modified only by the user's initialization function and by the application rules. The structure of the values is arbitrary. How values are added or changed is explained in the knowledge source section.

B. Control Structure

All CAGE control information is referenced through the Control-Structure object. The major components of the Control-Structure are:

User-Initialization: This is a user-defined function, handling any initialization needed for the user's program, e.g. setting-up the appropriate blackboard structure (on top of the predefined hypothesis framework) from the input data.

Termination-Condition: Another user-defined function, which determines when the application should be terminated. The Termination-Condition can access the step-lists for events or expectations, perhaps checking for a significant event; or the blackboard, checking a particular node or nodes. It should return a non-nil value when the application is to be terminated.

User-Post-Processor: When the termination condition is true, a user supplied post processing function is invoked. This function can be used to print out the application's results in a readable form, or to handle any other post processing details.

Event-Info: This is a pointer to the Event-Information object which contains both the user-specified information on how events should be scheduled, and run-time data including the event list and the current focus event.

Expect-Info: Similar to the Event-Info pointer, this object keeps track of the expectations generated by the application and information specifying how those expectation should be scheduled.

Control-Rules: A list of of control rules defined by the user to determine when to execute which control step (event or expectation). The control rules are defined using the DEFCONTROL-RULE macro. Each control rule consists of a condition, an arbitrary LISP expression and a steptype, either event or expect. The following example of a control rule says that if there are any events pending on the event list (steplist of event-info is not null), then do an event next.

```
Example:

Control Rule : Crule-1
    Condition Part:
         If    : event-info⊕steplist
    Action part : event
```

LHS-Evaluator: The default function for evaluating the conditions of a rule if the knowledge source containing that rule has no left hand side evaluator over-riding this default. For most applications the CAGE provided function QAND will suffice. It is a serial or concurrent boolean AND depending on the parallel options selected by the user.

1. Event-Information

A blackboard system can be executed in several ways, the simplest being event-driven. This means that each time a rule action is executed the system records that change to the blackboard as an event. Each event is added to a list called the *event list*. The scheduler selects an event from the event list to become the next *focus event*. The type of focus event is matched against the preconditions of the knowledge sources, and all the matching knowledge sources are activated. The rules of the activated knowledge sources are evaluated, those rules with satisfied conditions are executed and the cycle repeats until the termination is true.

To run a blackboard model with an event-driven control structure, certain control information must be supplied by the user.

selection-method: a function that determines which event to select from the event list. The user can write his own *best-first* selection method or use one of the CAGE provided functions, FIFO, LIFO, or AGENDA. If the AGENDA selection method is chosen, the user must also specify the agenda and an order.

agenda: An ordered list of event types supplied by the user. (See knowledge source specification for definition of event type.)

order: LIFO or FIFO order in which to check the agenda. There may be several different events of the same type on the event list.

collection rules: In some applications many events of the same type and the same node are generated and added to the event list. If the user specifies that type of event as a collection rule, then only one event is pursued and the others are *collected* and deleted from the event list.

2. Expect-Information

In an expectation-driven system, a rule may specify an expected result or change on the blackboard as one of the actions of that rule (called an expectation rule). When an expectation rule is executed, the expectation part of the rule is added to the *expectation list*. Later, when the control rules specify that an "expect" step should be executed, a focus is selected from the expectation list. If a change has occurred on the blackboard that satisfies the expect portion, actions associated with the expectation rule are executed.

Much of the information required to execute an expectation-driven system is similar to that of an event-driven system. The user must supply a selection-method, possibly including an agenda and order, and collection rules. Some additional information is required to execute expectation.

matcher: a function which defines how to match expectations to the blackboard. CAGE provides on default, PASSIVEMATCH, which simply evaluates the expectation portion of the expectation rule to see if its value is non-nil.

C. Knowledge Sources

CAGE knowledge sources are a partitioning of the application knowledge into sets of rules. Each knowledge source consists of some declarative information and a set of rules.

1. Knowledge Source Declarations

The definition of a knowledge source consists of more than just groups of rules. In order to properly interpret those rules, CAGE needs to know certain knowledge source control information, e.g.,

1. Under what circumstances should this knowledge source be invoked?
2. How should the rule conditions be evaluated,
3. what levels of the blackboard structure will be changed?
4. Which one or all of the rules whose conditions are true should be executed?
5. Are there any local variables or links to be defined for this KS?

The following features are available for the user to tailor a knowledge source to his own specifications:

Preconditions: A list of tokens, representing the *event types* used in rules. If the focus event has an event type that matches one of the knowledge source's preconditions, then that knowledge source is activated.

Levels: A list of pairs of blackboard levels or classes. The user must specify between which levels of his hypothesis structure a knowledge source makes inferences.

Links: If a knowledge source adds links between nodes on the blackboard, they must be defined here. The definition consists of a list of pairs of link names, a link and its inverse.

Hit Strategy: There are two main hit strategies available in CAGE, SINGLE and MULTIPLE. When a knowledge source with a single hit strategy is interpreted the rules of that KS are evaluated, in order, until one rule's condition evaluated to true. Then that rules actions are executed and no other rules are even considered. With a multiple hit strategy, the conditions of all rules of a knowledge source are evaluated and then all the actions of rules which successfully evaluated executed. In conjunction with either single or multiple hit strategies, the user can also specify ONCEONLY. This will cause a rule to be marked when its conditions are successfully evaluated. Its actions will be executed and it will never be evaluated again during that run of the application.

Definitions: A list of local definitions, available to all the rules of a knowledge source. The definitions are an efficiency feature to avoid the repeated calculation of the same value by all the rules. The structure is similar to that of LET, a list of pairs, a variable name and an expressions to be evaluated and assigned to the the variable. If the value is NIL it can be omitted.

Rule Order: A list of rule names, representing the rules of the knowledge source. This is the order in which the rules will be evaluated serially. Because the rules are actually defined as methods of the knowledge source to which they belong, each name should begin with a colon (:).

LHS Evaluator: The user can optionally specify a left hand side rule evaluation function for each knowledge source. There is also a default LHS evaluator specified for the entire application in the Control data. The evaluator specified here will override the default evaluator for this specific knowledge source. The LHS evaluator is a function which determines how the rule conditions are evaluated. CAGE provides several built-in functions which the user can select, including AND, for a simple boolean AND of the conditions and QAND for a concurrent boolean AND.

The following is an example of the definition of a knowledge source from the CRYPTO system written in the CAGE language.[****] The name of this knowledge source is "combine-weights", it has two preconditions, makes inferences from the Cryptoletter level of the hypothesis structure to the alphabet-letter level, defines a pair of bi-directional links, and uses the single-hit rule selection strategy. The combine-weights knowledge source also makes two definitions, possible-values gets the value NIL and lhs-evaluator the value QAND.

```
Knowledge Source : combine-weights
  Preconditions : Confirmation, Contradiction
  Classes : Cryptoletter : alphabet-letter
  Links : Possible-Value-of : possible-Letters
  Rule Selection : Single

Definitions :
    possible-values ≡ nil
    lhs-evaluator ≡ qand
```

This compiles to the following CAGE macros.

```
(defknowledge-source COMBINE-WEIGH1S
  :preconditions (confirmation contradiction)
  :levels ((cryptoletter alphabet-letter))
  :links((possible-value-cf possible-letters))
  :hit-strategy (single)
  :bindings ((possible-values))
  :rule-order (:letters )
  :lhs-evaluator qand)
```

2. Rules

CAGE rules consist of three major parts; definitions, conditions, and actions. Here is an example from CRYPTO in CAGE.

```
Rule :  letters {3}

Definitions :
    possible-values ≡
       possible-values(focus-node⊕
                            possible-letters)

Condition Part :
    If       : qand(focus-node·is-cryptoletter,
                    possible-values)

Action Part :
    Changes :
        Change Type    :  Update
        Updated Node   :  focus-node
        Event Type     :  possible-assignment
        Updated Slots  :
           possible-letters ↔ possible-values


;Combine the weights of identical possible
;values.
```

CAGE also provides a macro for defining rules called DEFRULE, to which the above will compile.

```
(defrule (combine-weights :letters)
  ((possible-values
     (possible-values
       ($value focus-node :possible-letters
                   :all))))
  ((is-cryptoletter focus-node)
    possible-values )
  ((propose :EVENT-TYPE 'possible-assignment
            :CHANGE-TYPE 'update
            :HYPOTHESIS-ELEMENT focus-node
            :LINK-NODE nil
            :ATTRIBUTES-AND-VALUES
              '((possible-letters
                   ,possible-values supersede))
            :SUPPORT 'combine-weights)
   ))
```

After specifying the knowledge source to which a rule should be added and the name of the rule, preceded by a colon, the user must specify the three major parts of the rule.

Definitions: The definition part of a rule is similar to a LET in structure. The local variables set here are available only to this rule, both in the condition and action parts, as well as other definitions of this rule. This is an optional component of a rule, and can be NIL.

Conditions: The second part of a rule contains the conditions. These can be one or more arbitrary LISP expressions which will be evaluated according to the left hand side evaluator as specified in the local knowledge source or at the control level. The conditions can reference both local variable definitions or variables bound at the knowledge source level. The CAGE system provides several access functions for retrieving values from the hypothesis structure, which can be used in the conditions of rules. It is important when writing the conditions of rules for a CAGE application to keep in mind the feasibility of running those clauses concurrently, i.e. keeping them independent of each other.

Actions: The action clauses make up the final part of a CAGE rule. These clauses have a very specific structure as evidenced by the preceding examples. The actions specify what changes are to be made to the hypothesis structure by a rule and how those changes should be made. The user must specify what node and attributes on the blackboard are to be changed, what the new links or values are, and how those changes are to be made (possibly deleting some old values). The user must also specify an event type, a name representing the type of change this action makes to the blackboard. If and when the event created by this action is selected as a focus event, this token will be matched against the preconditions of the knowledge sources to determine which KS to invoke next.

D. Initialization

There are two types of initialization which can occur at the beginning of a CAGE run. First CAGE must create the instances of all the application defined flavors which will constitute the executable form of the user's system. In addition, the user can do any other initialization he feels appropriate by defining his own initialization function, the name of which should be stored in the application's control structure. Since the major components of the application are defined as flavors, initialization can be done by defining :initialize or :after :init methods.

[****] The colons in the CAGE language ar[illegible]rs when separated by spaces from other words in the language. Colons indicate keywords when they directly precede a word.

E. Input Data

The user must define two functions to handle his input data.

1. INPUT-PROCEDURE(Record, Time) : Given an input record, retrieved automatically at the correct time by CAGE, do what ever should be done with that input,e.g. add it to the blackboard.
2. TIME-OF-INPUT-RECORD(Record) : Given an input record, return the time stamp.

At the beginning of each run the user will be asked to specify an input data file by typing in the file name or selecting a file from a menu of pre-specified input data file names. The data file consists of records that can be read by the above two functions. A time stamp is mandatory on each input record.

IV SPECIFYING CONCURRENCY

CAGE supports the concurrent evaluation of pieces of knowledge. Once an application has been debugged in serial mode, the user can specify one or several knowledge source components to be executed in parallel. For example, the user might specify that the rules of the knowledge source be evaluated concurrently, or perhaps just the actions of the rules or a combination of the available options. With a minimum amount of recompilation, the user can change his parallel specifications and experiment with many different configurations.

In general more speed-up should occur as more components are run in parallel. But for some applications the overhead of setting up the new processes and inter-process communication costs will be greater than the speed-up gained by executing particular components concurrently. For example, if most or all of the knowledge sources of an application contain only one rule, then it would not be efficient to evaluate rules in parallel since for any one KS invocation there would only be one item to evaluate.

A. Concurrent Components

The use of knowledge sources to partition the knowledge in blackboard systems and, in particular, the structure of the knowledge sources in CAGE provide several obvious places for concurrency. The knowledge sources group the domain knowledge into independent modules, which theoretically, could be invoked independently and concurrently. Within each knowledge source the rules provide another source of parallelism, and within each rule, the clauses of the condition and action parts provide yet another. Of course not all clauses, rules or even knowledge sources are actually implemented totally independently of each other and some serialization may be necessary to correctly solve the application problem.

The following are the options for parallelism available in CAGE, grouped according to their allowed use in combination.

Clause level: can be used in combination with each other or any other parallel option.

actions: Execute the RHS action clauses of a rule in parallel. Note: When running RHS actions concurrently a non-deterministic system may result if both destructive (Supersede in CAGE) and constructive (Modify) actions occur to the same object in parallel. (Same object and attribute) A QLOOP macro is used to initiate the parallelism for loop actions, requiring recompilation of the rules containing loop actions.

lhs: Evaluate the LHS condition clauses of a rule in parallel. Note: Use the rule bindings to set any local variables tested here, insuring that the lhs clauses will be independent. A QAND macro is provided as the LHS-evaluator to initiate the concurrency for the conditions, requiring recompilation when this option is used.

rule-bindings: Evaluate the definitions of a rule in parallel. Again, these definitions should be independent of each other if their concurrent evaluation is to result in an actual speed-up.

Rule level: bindings can be used in combination with any of the other options, but only one of the rule options, single, multiple, sync or nosync can be used at a time.

bindings: Concurrently evaluate the definitions at the beginning of a knowledge source.

rules-single: Evaluate all of the conditions of the rules of a knowledge source concurrently, but only execute the actions of one successfully evaluated rule.

rules-multiple: Evaluate all of the conditions of the rules of a knowledge source concurrently, then serially execute the actions of all the successfully evaluated rules.

rules-sync: Evaluate all of the conditions of the rules of a knowledge source concurrently, then concurrently execute the actions of all applicable rules.

rules-nosync: Begin evaluating the conditions of the rules of a knowledge source in parallel and execute the actions of each rules as soon as the conditions are known to be true. With this option there is no synchronization between the left and right hand sides of rules.

Knowledge source level: Only one of the knowledge source options can be set at any one time.

kss: Invoke all the applicable knowledge sources concurrently at step selection, synchronizing by waiting for all knowledge sources to complete execution and add events to the event list before concurrently invoking a new set of kss.

kss-nosync: Invoke all applicable knowledge sources as soon as a new event is created. This option provides the least control of all the options available and does no synchronization. Many applications will have to be changed slightly to execute reasonably under these conditions, particularly removing any possible circular knowledge source invocations. To implement the parallel execution of knowledge sources without any synchronization, the control loop of CAGE was drastically altered from that described at the beginning of this paper. (See CAGE Overview.) Without any synchronization, as soon as an event is created it immediately allows all relevant knowledge sources to be invoked. No events are added to the eventlist and no focus event is ever selected. A timed loop was added to the top level control to re-invoke the user's initial knowledge source in case the system exhausts all previous events before the termination condition is satisfied.

kss-minisync: Add an event to the event list and do minimal computation at the point of synchronization before invoking the next set of knowledge sources. The main computation done is the collection and pruning of similar events, leaving fewer events to activate subsequent KSs. The mini-sync and no-sync options are different from the parallel kss option in that they don't use the serial step-selection procedure.

B. How to specify and change parallel components

A function, SELECT-PARALLEL-OPTIONS is provided to allow the user to quickly change the selected parallel options. SELECT-PARALLEL-OPTIONS has no arguments. A menu of parallel options will pop-up on the screen and the user can select new options or delete old ones.

V DESIGN DETAILS

CAGE is currently implemented in an object-oriented style, using the Flavors feature of ZETALISP. The top level object in CAGE is called the BLACKBOARD. From the Blackboard object there are pointers to each of the principle components of the system, as follows

control-structure: all control information specified before compilation is stored here, as well as pointers to run-time control structures.

hypothesis-structure: the blackboard solution space, which must be structured by the user.

knowledge-source-list: names of the knowledge sources containing the production rules of the user's application.

user-functions: optional, user-defined functions invoked by the rules

information-structure: optional, user-defined, static data structures

A separate data structure, Parallel-Specifications, is used to store the parallel options selected by the user.

The DEFKNOWLEDGESOURCE macros will create, at compile time, an object for each knowledge source, and a set of associated methods. During the initialization process an instance of each knowledge source object is created. Other instances may be created during system execution if one of the concurrent knowledge source options is selected. One of the associated methods, SETUP-AND-START, evaluates the knowledge source definitions and initiates the rule interpretation when a knowledge source is invoked.

Each rule is created as three methods, EVALUATE-DEFINITIONS, EVALUATE-CONDITION, and EVALUATE-ACTION, associated with the rule's name using the :case method-combination feature of Flavors. The keywords of the action clause listed above are keywords in the method definitions, and therefore must be preceded by colons in the macro definition of a rule.

CAGE utilizes a global variable, PARALLEL-SPECIFICATIONS, whose value is a list of the current parallel options specified by the user. It is initially NIL and is updated using SELECT-PARALLEL-OPTIONS.

During execution CAGE prints out messages indicating the state of the execution and uses some simple graphics to help the user observe the simulation of concurrency. A set of small windows will appear on the right side of the screen, one for each process initiated by CAGE. Any state messages generated by the parallel process will appear in one of these associated windows, instead of the main terminal i/o window. There is only room to display 12 of these small i/o windows at the same time and still have them large enough and leave them up long enough to be readable. If more than 12 processes are active at the same time, the windows will overlap.

VI FUTURE DIRECTIONS

The next step for CAGE will be a reimplementation on CARE. The instrumentation in CARE will provide us with the needed tools for measuring the speed-up gained from each of the various concurrent options in the CAGE System. CAGE users will be able to implement and debug their applications in the current CAGE-on-LOQS system with its fast simulation time. Once an application is debugged it could then be run on the CAGE-CARE system for complete and accurate measurements.

References

[Gabriel 84] Gabriel, Richard P. and McCarthy, John.
Queue-based Multi-processing Lisp.
Proceedings of the ACM Symposium on Lisp and Functional programming :25 - 44, August, 1984.

[Nii 79] Nii, H. P. and N. Aiello.
AGE: A Knowledge-based Program for Building Knowledge-based Programs.
Proc. of IJCAI 6 :645 - 655, 1979.

[Rice 86] Rice, J. P.
The L100 Language and Compiler Manual.
Technical Report KSL-86-21, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

Poligon, A System for Parallel Problem Solving

J. P. Rice

Knowledge Systems Laboratory, Stanford University

Summary

The **Poligon**[1] system is a new, domain-independent language and attendant support environment, which has been designed specifically for the implementation of applications using a Blackboard-like problem-solving framework in a parallel computational environment.

This paper describes the Poligon system and the Poligon language, its salient and novel features. Poligon is compared with other approaches to the programming of parallel systems.

1. Introduction

The larger project of which Poligon is only a small part will not be discussed here in any detail. Design decisions made in other parts of the project will be held to be axiomatic, though some mention of these decisions will be made in order to show the motivation for the features of Poligon. The primary objective of the overall project is to achieve significant speedup of knowledge based systems, particularly those directed at real-time signal understanding.

The purpose of the Poligon language is to express the problem solving behaviour of human experts in order to map them onto a problem solving framework, which will run on simulated parallel hardware.

The fields of knowledge representation and problem solving are rich and complex. This paper will not go into any great detail in describing the problem solving processes involved. Poligon tries usefully to express knowledge both in a declarative and procedural sense, through rules [Davis 77]; and in a structural sense, through the configuration of the solution space. These will be described below.

Some crucial design criteria and early design commitments have affected the development of Poligon, the consequences of which will be described in this paper. These can be summarised as follows.

- Poligon is intended to be a language for both problem solving and the general purpose programming necessary to support it. Unlike most programs, Poligon programs must also address the problems of real-time processing, including asynchronous events and input data backup. Poligon, therefore, must assist in this respect.
- The overall project's strategy is to solve problems significantly faster than existing systems through the exploitation of parallelism. Poligon is targeted at a MIMD, distributed-memory, message-passing machine with ~thousands of processors. This hardware gives direct support for futures, remote objects and such efficient message-passing strategies as *Broadcast* and *Multicast* so as to take full advantage of its processor interconnection network.
- A consequence of the desire to achieve a significant order of parallelism in Poligon programs is that many of the control mechanisms used in serial problem solving systems, such as schedulers and event queues, have been discarded because they are highly serial. Most actions in Poligon programs are, therefore, performed asynchronously. Rules, the primary mechanism in Poligon for describing things and for getting things done, are activated as daemons. Much of the work in Poligon is aimed at providing mechanisms to cope with this chaotic behaviour.

This paper contains the following;

- A discussion of related work in parallel languages.
- A discussion of the design approach guiding the development of Poligon.
- A description of the abstraction mechanisms provided by the Poligon system with some small examples.
- Some concluding remarks.
- References for further reading on the subject.

1.1. Knowledge Representation and Problem Solving in Poligon

The primary purpose of this paper is to discuss the Poligon language. It is, however, not possible completely to divorce this from the underlying hardware and from its purpose; knowledge representation and problem solving.

Poligon can be described loosely as a "Blackboard System". What this means in practice is that the problem solving metaphor of Poligon is one of cooperating experts gathered around a blackboard, posting ideas about their deductions on the blackboard. For an exposition on the term "Blackboard System" the reader is encouraged to read [Nii 86]. Poligon tries usefully to express knowledge both in a declarative and procedural sense, through rules and functions; and in a structural sense, through the configuration of the solution space on the blackboard. In particular, the term "blackboard" will be used to describe the set of all of the nodes in the solution space of the system.

[1] The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

The suggestion that Poligon is a blackboard system is a little controversial. There are a number of respects in which this is not a satisfactory label. This term will, however, be used freely from now on for lack of a better label. The reader is encouraged to substitute for the term "Blackboard system" any term, such as "Frame System" which seems best to fit his mental model of what is being described.

1.2. Poligon's Model of Parallelism

It seems appropriate here to describe Poligon's model of parallelism. In its simplest form this can be thought of as *An Element in the Solution Space as a Processor.*

This gives some idea of the granularity that is being sought. It is, however, by no means the most efficient way to implement Poligon. Poligon programs want to be able to execute rules and parts of rules associated with a particular *Node* in the solution space in parallel. These rule activations need processors, on which to execute.

Thus a modified version of Poligon's model of parallelism could be *A Rule Activation as a Process, with sufficient processors to cope with the parallelism exhibited by the rule during its activation.* This tends towards a mapping of solution space elements onto a cluster of processors to service the rule activations. In practice, however, a number of nodes might be folded over the same set of processors, either because nodes become quiescent or because the load balancing in the system is sub-optimal.

2. Related Work

Work in this field falls into two distinct categories; work on parallel knowledge based systems and work on languages for parallel symbolic computation. The former is, at present, a very sparse field and, will not be discussed here, though some references are given in § 6. The latter is much more highly developed.

Much work is already being done on parallel languages for general computation. Amongst these languages are **Actors**, **MultiLisp** and **QLisp** on the one hand and concurrent logic programming languages and purely functional languages on the other. Often missing from this work is a thrust toward the investigation of large applications in parallel domains, for instance the development of parallel knowledge representation and problem solving systems. This is, of course, what Poligon attempts to do. This section will discuss briefly Actors, QLisp and Multilisp, since these are the parallel symbolic computation languages which are most relevant to the development of Poligon and the software which lies beneath it.

2.1. Actors

Actors [Hewitt 73] probably come the closest in their behaviour to Poligon, at least at an implementation level. Actors are independent, asynchronously communicating objects. As is the way with purely object oriented systems they communicate only through message passing and have tightly defined operations. The mutual control of Actors an parallelism is achieved by the support of procedure call and coroutine model message passing. The modularity afforded by this sort of programming metaphor may well be especially useful for the programming of distributed-memory, message-passing hardware, since having a close match between the hardware and software metaphors is likely to achieve better performance. It is not in any way surprising that the operating system level software, which underlies Poligon, is founded on many of the same principles as Actors. It has yet to be seen whether this programming methodology is able in practice to extract significant amount of parallelism from problems, though clearly this project hopes that it is.

2.2. MultiLisp and QLisp

MultiLisp [Halstead 84] and **QLisp** [Gabriel 84] are lumped together because, at least in some senses, they have strong generic resemblances. They are both, at the user level, extensions to existing Lisp dialects which provide mechanisms for the expression of parallelism, such as parallel Let constructs and parallel function argument evaluation (QLet and PCall). It is assumed by both of these systems that the hardware at which they are targeted is a form of shared-memory multiprocessor. Although there is no particular reason why such systems could not be implemented on a distributed-memory system, they are optimised for shared-memory multiprocessors. These are currently the most readily available form of multiprocessor. They would, however, need significant extensions in order to be able to exploit a distributed-memory system as is shown in **CAREL** [Davies 86], an implementation of QLisp for distributed-memory machines. The assumption of shared-memory, MIMD processors in these systems imposes constraints on the languages. They assume, at least to an extent, that processes will be expensive and that the user must have control over their creation. Poligon assumes quite the opposite.

3. The Design of Poligon

Poligon will be discussed first in terms of the way in which the language relates to the problems being solved and its underlying systems. Next the language will be discussed in terms of the requirements for languages in general and parallel languages in particular.

3.1. Background and Motivation

The philosophy behind the design of Poligon comes from intellectual and pragmatic pressures. It attempts to steer a middle course between the extreme purism of applicativists and the extreme pragmatism of the proponents of side-effects.

From the outset, the project was oriented towards real-time problem solving. Blackboard systems are well known to be of interest as tools in the knowledge engineer's toolkit. Little work has been done to investigate the appropriateness of the blackboard metaphor to parallel execution or the meaning of parallel blackboard systems, though it is frequently claimed that they are full of latent parallelism. The excellent formal properties of pure applicative and logic languages may well be of little use in a system which, for whatever reasons, needs to express side-effects and which has to cope with real-time constraints. Poligon is a system in which some of the formal rigour of truly applicative systems has been put aside in favour of a pragmatic approach to the exploitation of parallelism.

The **BB1** project [Hayes-Roth 85], also a project at the HPP, is an attempt to investigate the behaviour of highly controlled problem solving systems. It attempts to use a great deal of meta-knowledge and makes significant use of globality of reference in order to support an holistic view of its solution space, thus providing a basis for meta-level reasoning. The Poligon project is an attempt to investigate quite the reverse. Poligon has very little support for meta-knowledge and allows no global data or global view of the solution space whatsoever. The purpose of this experiment is to determine whether a system, unconstrained by a great deal of serialising control knowledge, might still be able to find useful answers faster than an highly controlled system, such as BB1, which would be extremely difficult to speed up significantly through parallelism.

The Poligon system pictures the elements in its solution space as processes resident on processors distributed across a grid, with the code necessary for them intimately associated with them. Because no global control is permitted in Poligon the activation of rules is necessarily completely daemondriven.

The project hopes to achieve significant speed-up through parallelism. This can be done only if much parallelism is extracted from the problem. Ideally, the system would try to achieve its parallelism by exploiting parallelism in the program's implementation at a very fine grain. This can, in principle, extract the maximum amount of parallelism available. On its own it has drawbacks, however. The costs of processes and the problems of synchronisation at a fine grain size make it difficult to exploit such parallelism without the use of hardware mechanisms significantly different from those available with prevailing technologies. This approach is also only part of the story. It neglects the fact that a properly parallel decomposition of the source problem is crucial to finding a lot of parallelism. One could summarise the problems, therefore, as expressing the problem in a sufficiently parallel fashion and the matching of the parallelism in the program to the grain size of the underlying hardware. Poligon addresses these issues.

Parallelism is very hard to find in conventional programs. Applicative systems have an advantage in this respect because of their relative lack of need to express parallelism explicitly. Their unchanging semantics when parallelism is introduced eases matters considerably. Poligon has attempted to learn from this and has pure applicative semantics in a number of areas but takes a different approach to the finding of parallelism in programs. It attempts to execute everything in parallel that it can and leaves it to the programmer to find any serial dependencies.

When the parallelism in a program is user-defined, problems can result from an inappropriate match between the granularity of the parallelism expressed in the program and the granularity of the underlying machine. In systems of the size and complexity of a typical Poligon application such a match would be particularly difficult to find because of the large number of processors involved and because it would be difficult for the user to keep track of the location of his data in the processor array. These characteristics are a consequence of the highly variable and data dependent state of the solution space in such programs. Poligon, because of its structure, should be able largely to obviate such granularity mismatches because parallelism is defined and controlled by the system and the Poligon system is closely matched to the granularity of the underlying system.

It is often thought that problems suitable for solution by means of the blackboard model tend to partition their solution spaces into what look rather like pipe-lines. Pipe-lines are, of course a well known form of parallelism. In practice pipes in such systems are not pipes in the normal sense, since they are more like "leaky" pipes. It is one of the prime objectives of these systems to reduce the amount of data as it percolates up through the abstraction hierarchy of the solution space. Because of the reduction in the data rate flowing in these pipes the contention problems that one might expect when pipes are connected into trees, as they often are, are alleviated.

A significant limitation of the performance of pipelines is that, at best, the parallelism that they can produce is proportional to the length of the pipe. This would typically be only of the order of half a dozen sections. This is clearly not the "orders of magnitude" of performance improvement that we all hope for. In practice, though, given a large enough problem, it is often possible to set up a large number of these pipes side-by-side. It is one of the major objectives of the Poligon language to encourage, facilitate and reward the decomposition of problems so that this form of independence can be exploited, so that such pipes will be created by the system.

3.2. Language Requirements

Poligon is a language which is by no means directed at general computation. It is nevertheless intended to be used for the solution of large, complex problems on distributed-memory parallel hardware. The following is a brief list of the ways in which Poligon attempts to address some of the primary requirements of programming languages.

- The language should provide a tangible method of expressing the ideas of the programmer.

  The Poligon language has been written with considerable input from those with experience in problem solving systems in the application domains at which it is targeted. It is therefore intended to match the ideas of the "Expert", whose knowledge is to be encoded. but in a domain independent way.

- The compiler[2] should provide a mapping between the language and the underlying systems, be they hardware or software.

  Poligon's compiler compiles Poligon language source into code understood by the underlying *Lisp* system and the concurrent object-oriented operating system running on its target hardware.

- The language should abstract the programmer from its underlying systems.

  The Poligon system shields the user from all aspects of the underlying hardware such as the topology of the processor network, the message-passing behaviour of the hardware and the location of any code or data within the network.

- The language should provide mechanisms for the exploitation of the underlying systems to good effect.

  The underlying hardware and software systems are exploited in a number of ways in Poligon. Firstly the language encourages the user naturally to decompose his problem into a form which will map efficiently onto the underlying hardware. Secondly the language offers a number of application-independent, high-level constructs, which are designed to exploit the hardware to the full. These topics are covered more fully in § 4.

- The language should allow the development of software faster than would be the case if it were to be developed in a less abstract form.

  Considerable effort has been spent on making the Poligon language a high level way to describe the solutions to parallel knowledge based system problems. A high level language with such features as infix, user-definable operators and user definable syntax, provides a natural way for the expert to implement his knowledge.

  Much effort has been spent also on integrating the Poligon system cleanly into the program support environment of the Lisp Machines on which it runs. For instance, incremental compilation is supported from within the editor.

- The language should assist the development of reliable, maintainable and modular software.

  Language features are provided to minimise the possibility of inconsistent modifications to the source code and the structure of the language and its semantics are defined in a manner which min-

[2] The term *Compiler* is used in its most general sense here, perhaps an interpreter or a machine which is clever enough to execute the language specified directly.

imises the probability of complex bugs being introduced by asynchronous side-effects.

A sophisticated set of debugging facilities is provided. A system that emulates the semantics of full, parallel Poligon programs as closely as possible in a serial environment has been produced. The user is able to debug his program serially to remove all possible serial bugs and bugs due to the non-deterministic execution order of Poligon programs before it is ported to the full parallel environment.

In addition to these requirements a language targeted at parallel hardware should have a number of atuibutes which reflect the parallel nature of the target hardware.

- The language should address the granularity of the hardware.

  Poligon is closely matched to the granularity of the hardware at which it is targeted. It is generally expected that the solution space of the problems addressed by Poligon programs will have of the order of thousands of nodes. This is of the same order as the granularity of the hardware.

- The language should provide a mechanism for the extraction of parallelism from programs and from the programmer.

  Poligon extracts parallelism from programs and the programmer in two main ways. First the decomposition of the problem is encouraged to be as modular as possible. Secondly the semantics of Poligon programs are such that almost all of the program can be executed in parallel without changing their behaviour from that seen during serial execution. This allows the system to execute most operations in parallel if it has the resources to do so.

- The language should, where appropriate, shield the programmer from those details of the hardware which are particular to parallel computing engines, such as topology.

  The hardware, on which Poligon programs runs, causes Poligon programs to have to cope with communication between solution space elements on different processor sites. All such message passing is hidden from the user. In fact the Poligon language has no concept of message-passing at all.

  Futures are used for all remote operations in the user's program. The hardware implements these such that there is no efficiency penalty associated with creating futures for such remote accesses. The Poligon language copes with these invisibly to the programmer.

As can be seen quite easily from the above one of the factors that must be well understood before a language is designed is the general purpose of the language and the level of generality that is expected of programs written in it. A language, whose sole purpose is the expression of solutions to huge matrix problems on systolic hardware might well be justified in expecting the programmer to express, at quite a low level, the mapping of the program onto the hardware provided. This is less likely to be a reasonable expectation of a language targeted at the solution of large, complex problems of an unpredicatable, dynamically-varying or data-dependent nature. Poligon is a fairly general purpose programming language with a very definite bias.

4. Abstractions in Poligon

To cope with Poligon's view of parallelism and with the chaotic execution of rules (see § 1) a number of linguistic abstractions are provided.

Poligon provides abstractions for knowledge representation, control, data, parallelising, real-time and side-effect control. These will be described briefly in this section.

4.1. Knowledge Representation

Knowledge is traditionally represented in blackboard systems in a number of ways, listed below.

- *Declarative Knowledge* is encoded in *Rules.*
- *Procedural Knowledge* is encoded in procedures.
- Knowledge concerning the sequencing of activities is encoded in the scheduling mechanism.
- Knowledge about the structure of the solution space is encoded by the definition of the structure of the blackboard.
- Knowledge about relationships between the objects in the system is often encoded using a Link mechanism.

These all represent knowledge about the application domain. In addition, there is in any program a large body of implicit knowledge concerning the semantics of assignment, sequencing and the system's function as a whole, especially in for systems with poor formal properties. This will not be discussed here. The Poligon language does, however, go to considerable effort to make the semantics of the Poligon system as clear as possible.

4.1.1. Declarative Knowledge

The encoding of *Declarative Knowledge* in blackboard systems is conventionally done in *Rules*[3], which exist within scheduling units known as *Knowledge Sources.* Poligon also has the concept of Rules and Knowledge Sources, though their meaning is somewhat different. Unlike serial blackboard systems, the rules in a Poligon system are activated autonomously and asynchronously.

Existing blackboard systems usually suffer from a confusion and overloading in the semantics and purpose of knowledge sources. It is useful to collect one's knowledge of one subject together into one chunk. These chunks are knowledge sources. Sadly, the implementors of blackboard system frameworks often think of knowledge sources as scheduling units and thus design their scheduling strategies around the idea of the "invocation of knowledge sources", even though it is by no means necessarily the case that it is appropriate to schedule all of knowledge in a chunk at the same time. This has a detrimental effect on the modularity of the system.

In Poligon, knowledge sources are used as linguistic and software engineering abstractions provided for the programmer in order to allow him to collect related knowledge together. There are no scheduling semantics associated with knowledge sources in Poligon. Because of the underlying system's daemon-like rule triggering mechanism the rule writer is allowed completely to decouple the concept of *scheduling* from the concept of *chunks of knowledge.*

Rules are activated as a result of "events" happening to the fields of nodes (see § 4.3.1). These events can be caused ei-

[3] The term *Rule* is used here in the sense of "Pattern/Action pairs". It should be noted that these are quite unlike the structures called rules used, for instance, in **Prolog**. Pattern/Action rules move towards a solution to their problem by performing side-effects on their environment, in this case the blackboard, not through unification.

ther by a write operation to a field, by a semaphore being waved at a field or by the real-time clock.

A powerful *Expectation* mechanism is provided, which allows the dynamic placement and specialisation of rules. An Expectation is a way of expressing model-based knowledge. Given a particular model of the behaviour of a system, certain changes might be expected if the model's interpretation of the world is correct. Expectations allow such changes to be watched and even allow their associated rules to be triggered if the changes do not happen in a given time. Such expectations can be placed to watch for events happening, or not happening, in specific places on the blackboard, at specific times. Expectations provide a focussing mechanism[4] and, coupled with the system's ability to trigger[5] rules and "time-out" unsatisfied Expectations on the basis of the real-time clock, Poligon allows complex time-critical knowledge to be expressed and applied simply.

An example rule is shown in figure 4-1.

4.1.2. Procedural Knowledge

Procedural Knowledge is an all encompassing term usually used indiscriminately to describe both knowledge about the relationships between values (*Functions*) and the mechanisms for performing side-effects and for sequencing events (*Procedures*). This is often a result of such systems being built on top of Lisp systems, which fail to draw distinctions between procedures with side-effects and those without. Poligon does not allow the encoding of arbitrary knowledge into procedures. Only side-effect free functions are allowed. Side-effects are permitted only in the bodies of rules, where they can be controlled.

4.1.3. The Sequencing of Activities

In most blackboard systems knowledge of the required sequencing of events at a macroscopic level is expressed by the implementation of the system's scheduler. In many cases, such as AGE [Nii 79] this scheduler has fixed characteristics and the application has a fixed interface to it. In others, such as MXA [Rice 84], the user can specify the characteristics of the scheduling of knowledge sources. Poligon provides no such mechanism. Since all rules are activated as daemons, entirely asynchronously, the only analogue of scheduling is the implicit sequencing of the activation of rules due to some rules causing changes that trigger other's rules.

4.1.4. The Structure of the Solution Space

Poligon is unlike most blackboard systems in this respect. Most blackboard systems partition the blackboard into *Levels*, which represent the hierarchy of abstraction in the solution space. Poligon uses a much more general representation which is like that of some *Frame* systems, providing a "Class" mechanism with user defined classes and metaclasses, and compile-time and run-time inheritance. The functionality of the class mechanism in Poligon is a superset of that of the levels provided by most blackboard systems. The programmer can, of course, represent his solution simply using classes as levels in Poligon if he wishes. Classes are discussed more in § 4.3.1.

The following is a trivial example rule, which shows a small set of the features of Poligon. This rule could be interpreted as saying: *"If the most recent two phonemes that hove been seen ore "oo" and "ph" then the word is "foo"*. Having concluded this the rule finds the set of sentence components, which represent potential conclusions of the word "foo", and sets them so that they are no longer marked as hypothetical. It also makes a Sentence-Component type node, which represents the word "foo", which has been found.

```
Rule : Find-the-word-Foo
    Clasa : Phoneme
{ Class of nodea with which the rule will be associated }
    Field : uncorralated-phonemes
{ Try to activate this rule when this field is changed }

Definitions :
    all-phonemea-in-order ≡
        Tha-Phoneme⊕↑uncorrelated-phonemes
{ The operator "⊕↑" returns all values in a field in }
{ time order.  The-Phoneme repreaents the node. that }
{ triggered this rule }
    most-racent-phonama ≡
        all-phonemaa-in-order·Haad
    next-most-racant-phoneme ≡
        all-phonemes-in-order·Tail·Head
{ Head and Tail are like CAR and CDR only they operata }
{ on lists. Lazy lists and Baga }

Condition Part :
    When : all-phonamea-in-order·length-of-list ≥ 2
{ The "Whan" part is a locally evaluable pracondition }
    If   : moat-recent-phoneme·Sound = "oo"
           And next-most-recent-phoneme·Sound = "ph"
{ The precondition for tha Rula }

Action Part :
    Definitiona :
        new-sentence-component ≡
            New Instance of Sentence-Component
{ The creation of the new Santence-Component noda }
        hypothetical-fooa ≡
{ A Bag of words, which ara "foo" }
            Subset of Words which satisfies
                λ(a-word)
                    a-word·hypothetised And a-word·letters
                    = [ f o o ]
                Endλ

{ Process all elamenta in the Bag hypothetical-fooa }
    Changes :
        In Parallel for each a-word in hypothetical-foos
            Change Type    : Update
            Updated Node   : a-word
            Updated Fielda : hypothetised ← nil

{ Sat fields of new aentence component in }
{ parallel with updating the elements in the Bag }
    Changes :
        Change Type    : Update
        Updated Node   : new-aentence-component
        Updated Fields : letters ← [ f o o ]
                         conatituenta ←
                           List(next-most-recent-phoneme,
                                most-recant-phoneme)
```

All of the actions taken by this rule are performed in parallel, since they are independent of one another, though there is, of course, a serial dependency between the condition part and the action part of the rule.

Figure 4-1: An example Poligon rule

4.1.5. Knowledge about Relationships

Relationships between entities in blackboard systems are often expressed by a form of *Link* mechanism. Sometimes this link is not so much a part of the system as a reflection of the fact that fields in nodes can have as their values other nodes in the system. Other systems have more sophisticated mechanisms that express links explicitly and allow property inheritance along links, e.g. BB1, or the propagation of likelihood, e.g. MXA.

Poligon has a number of system defined relationships; "Is an Instance of", "Is a part of" and "Is a subclass of". The user can define arbitrary relationships between nodes on the black-

[4] It should be noted that the term *Focussing mechanism* is used in a more general sense than by many blackboard systems. There can be any number of such foci all acting in parallel in a Poligon program. The expectation mechanism is another way of applying knowledge in order to take advantage of some local circumstances in order to solve a problem more efficiently or cleanly.

[5] A rule is said to have been *Triggered* when it is activated so that it tries to evaluate its preconditions and body.

board. These links allow property inheritance and are, themselves, represented as nodes and so can have attributes in the same way that any other nodes can. Links are therefore first-class citizens in Poligon and they allow Poligon programs to act like semantic nets.

4.2. Control Abstractions

The flow of control is a rather evanescent concept in a Poligon program. Any rule can be triggered at any time. It is important not to think of the control flow in a Poligon program in the same terms as that of a conventional serial program. There is a well defined flow of control within rules; the action part of a rule is activated after the condition part, upon which it is predicated. Apart from this, however, there is no flow of control in any normal sense. It should be noted also that what little flow of control there is only specifies the strict ordering of activities. The execution of a sequence of actions can be interrupted at any time. The size of the atoms for Poligon's atomic actions is very small.

The triggering of rules is controlled by the user associating rules with particular fields of nodes or classes of nodes on the blackboard. The triggering of rules occurs when a field, which is being watched in such a manner, is updated or is semaphored. A semaphore mechanism is provided to allow rules to be triggered without a field being updated. This provides a form of explicit event-based programming, if it is needed.

Clearly one of the objectives of the design of the Poligon language is to provide a language in which it is simple to express logically distinct pieces of knowledge, independent of other such pieces of knowledge. The decomposition of the problem in this manner causes the system to appear to iterate towards the solution of its problem by small, simple and discrete steps, rather than by complex, giant leaps.

4.3. Data Abstractions

Poligon provides a number of distinct data abstractions. One is characteristic of other blackboard systems, one of pure functional languages and one is rather novel.

- The structure of the blackboard is characterised by being made of *Nodes*, elements in the solution space. These have a user-defined, record-like structure.
- Lazy evaluation is supported.
- Bags are supported as data structures, which parallelism enhancing.

Numerous operations are defined for these data abstractions, particularly a number of generic operations which can be applied to lists, lazy lists and bags, which shield the user from the underlying data structures used by the system or by other segments of his program.

4.3.1. The Structure of the Solution Space

The most obvious data abstraction provided by Poligon is similar to that provided by conventional blackboard systems, that is, the *Node* on the blackboard as an element in the solution space. Such nodes are record-like internally. They have named fields, which can often contain multiple values to be associated with that name. Poligon provides this but also goes beyond it.

Conventional blackboard systems, such as AGE, tend to provide nodes on a blackboard divided into groups, often called "Levels". "Levels" themselves are not represented. Arbitrary use of global data, held in global variables, distinct from the blackboard is also allowed.

Poligon has a much more regular representation for data. The nodes are represented as instances of *Classes*. The Classes themselves are represented as Nodes, which "control" their instances. Knowledge concerned with classes as a whole can be associated with these nodes. Shared, global variables are not allowed in Poligon.

Poligon also provides;

Superclasses — Classes that provide characteristics to the instances of classes. These can be thought of as templates for the instances.

Metaclasses — Classes that provide characteristics to the classes themselves. These can be thought of as templates for the classes.

Thus the classes are themselves instances of metaclasses, which can be user defined, such that instances of a given class can have any number of superclasses, i.e. component templates, and any number of metaclasses, i.e. component templates for their parent class. It is possible to instantiate classes any number of times, as well as their instances.

Automatic property inheritance allows shared data to be located on locally central nodes, which are immediately visible to the interested parties. This distributes shared data in such a manner as will, hopefully, minimise hot-spotting.

An example class declaration, the specification of a template for a class of nodes, is shown below. The declaration defines a class of nodes called *Words*, each instance of which has two fields (slots) called *Letters* and *Sound*.

```
Class Words :
    Fields :
        Lettera
        Sound
```

Extensions to this sort of syntax allows the definition of superclasses and metaclasses within class declarations. The following example defines the class *Sheep*. Each instance of the class *Sheep* will have the characteristics defined for sheep and for mammals. The class called *Sheep* (an instance, in fact, of the class *Meta-Sheep*) has the characteristics of *types of animals*.

```
Class Types-of-animals :
    Fields :
        Rate-Of-Breeding

Class Mammals :
    Fields :
        Colour-of-fur
        Number-of-legs : 4

Class Sheap :
    Metaclasses  : Typas-of-animals
    Superclassea : Memmals
    Fielda :
        Thickness-of-wool
        Flock
```

4.3.2. Lazy Evaluation

Lazy Evaluation is supported in the guise of *Lazy Lists*, *Lazy Function Arguments* and in the form of the lazy association of expressions with names. The following is an example of the lazy association of a name with a value. The name A-Meaningful-Name is associated with the value of the call to the function *An-Expensive-Function*[6].

```
Dafinitions :
    A-Meaningful-Nama ≡
        An-Expensive-Function(an-arg, another-arg)
```

[6] Suitable *Force* operations are provided so that the time of eveluetion can be controlled by the progrem if necessery. These force operetors ellow the progrem to perform *Eager Evaluation* if it is needed.

The value of an item defined in a *Definitions* construct is always a future if it is possible to evaluate it as a future.

4.3.3. Bags

One abstraction suited particularly to the parallel mode of execution of Poligon programs is the *Bag* data type. Bags are implemented in Poligon so that they are formed as the result of efficient parallel operations and can be processed in parallel efficiently. Even when the elements of Bags are processed serially they perform efficiently. The lack of a defined ordering in the Bag means that the system can always return the first satisfied Future out of a Bag of Futures, causing minimum waiting for values. Similarly, when a program attempts to extract an element from a bag and there are no satisfied elements the process in which this happens will go to sleep until the next available future is satisfied.

A Bag is generated, for instance, as the value of the following expression. It is a Bag, which contains all of the *Words*, whose *Sound* is *"phoo"*[7].

```
Subset of Words For Which Element · Sound = "phoo"
```

4.4. Parallelising Abstractions

Poligon supports data representations which are designed to give the user a high level handle on the exploitation of parallelism. Most values computed in Poligon are derived as Futures. Computation is decoupled from the expressions which reference values. Futures are, however, completely invisible to the user in Poligon. It understands which functions are strict in their arguments and so waits for the satisfaction of a Future only when it is required. The programmer can, of course, declare his own non-strict functions and operators. All *DeFuturing* coercions are performed automatically by the Poligon system. Thus the following expression will deliver a list with two elements, one of which is the value of *a* and one of which is the sum of *b* and *c*. The first will be a future, if *a* is. The second will be the DeFutured value *b+c*.

```
List(a, b+c)
```

The efficient use of the bandwidth of the processor interconnection network is enhanced by the use of *Broadcast* and *Multicast* operations. Broadcast messages allow messages to be sent to every node in the system in a single operation. Multicast messages allow messages to be sent to a collection of nodes in a single operation. The Poligon system uses these extensively in the processing of the Bag data type and in the execution of groups of actions in parallel. It uses the same mechanisms to provide an efficient implementation for searching a collection of nodes on the blackboard for patterns, which tends to cause significant slowing of serial implementations because of the combinatorial nature of such searches. It allows the blackboard to be searched for bags of matching nodes in a single, fast operation. This provides a significant improvement over the serial construction of such collections.

4.5. Real-time processing

Real-time processing brings its own problems. Poligon provides a simple and regular mechanism for defining the interface between the Poligon system and its signal data. This data can be from an arbitrary number of different types of sources and is posted on the blackboard asynchronously.

Poligon also provides a mechanism by which each datum is timestamped from the time that it enters the system. These timestamps are propagated automatically by the system so that it is trivial for the programmer to manipulate time-ordered collections of values. This mechanism is required because the conventional implicit time ordering of data in lists cannot apply here and the non-ordered nature of Bags is sometimes not sufficient.

4.6. The control of assignment

Assignment is something which is likely to cause significant problems in any parallel system. Poligon constrains assignment in a number of ways. Side-effects are only permitted on the fields of nodes. All side-effects can be monitored by rules that might be interested in the changes to values. This removes the possibility of the knowledge base getting confused because of surgical side-effects to data structures at arbitrary times and at arbitrary places in the processor network. Assignment is also constrained so that all of the updates to the fields of a given node are done atomically, before any rules which might be triggered by these changes are allowed to trigger. Such atomicity helps to preserve the consistency of the system.

An example of a collection of updates to fields of a given node is given below. In this example the node an-instance-of-words is having two of its fields updated; *Sound* and Letters. Operators, such as "←", allow different sorts of modifications to be made to fields. Such operations might be "add this value to the values in this field" or "replace all of the values in the field". This avoids complex and potentially expensive expressions in the old value of the field being evaluated non-locally.

```
Change Type    :  Update
Updated Node   :  an-instance-of-words
Updated Fields :  Sound   ← "phoo"
                  Letters ← [ f o o ]
```

5. Conclusions

This paper has described Poligon, a language and system for the investigation of problem solving on distributed-memory, parallel hardware. The language was described in the context of related work in the field and in terms of the abstraction mechanisms provided. No significant description of the underlying run-time support has been given.

The Poligon system is still young. Only recently have applications been mounted on it in earnest. Two distinct applications in the field of real-time signal processing are now being implemented and more applications are likely to be started in the near future. Poligon has proved to be well suited to these applications as far as they have gone. No results from the simulation process regarding the performance of Poligon programs are yet available. Significant problems have been found in the simulation of the fine-grained parallelism required by the Poligon metaphor. Such simulations are very time consuming, prone to bugs in the underlying system software and simulator, and are difficult to debug. It is for these reasons that Poligon also has a serial version, Oligon, which accurately emulates the behaviour of the parallel system but without true parallelism. A simulated processor array of 256 processors has recently been made available to the users of Poligon. This simulation will allow more satisfactory investigation of the properties of Poligon programs in the future.

6. Further Reading

For a significantly more detailed treatment of the Poligon language and system the reader is encouraged to consult [Rice 86].

[7] The expression "Element · Sound" denotes extracting one of the values associated with the "Sound" field of the potential element in the bag. "·" is an operator that selects which of the values associated with the field is to be delivered.

The following topics were not described or discussed but are relevant to the work described above. The reader is encouraged to consult the following for further information;

- [KSL 85] for a description of the Advanced Architectures Project of which Poligon is a part.
- [Delagi 86] for a description of CARE, the hardware simulator used by Poligon, and of the particular hardware being simulated.
- [Schoen 86] for a description of CAOS, the concurrent object oriented system running on the CARE machine, which Poligon uses as its operating system.
- [Ensor 85], [Lesser 83], [Aiello 86] and [Fennel 77] for other approaches to parallel problem solving using blackboard systems.

References

[Aiello 86] Aiello, Nelleke.
The Cage User's Manual.
Technical Report KSL-86-23, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

[Davies 86] Davies, Byron.
Carel: A Visible Distributed Lisp.
Technical Report KSL-86-??, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

[Davis 77] Davis, R. and J. King.
An Overview of Production Systems.
In E.W. Elcock and D. Michie (editor), *Machine Intelligence 8: Machine Representation of Knowledge*, . John Wiley, New York, 1977.

[Delagi 86] Bruce Delagi.
CARE User's Manual
Heuristic Programming Project, Stanford University, Stanford, Ca. 94305, 1986.

[Ensor 85] Ensor, J. Robert and Gabbe, John D.
Transactional Blackboards.
Proc. of IJCAI 85 :340 - 344, 1985.

[Fennel 77] Fennel, R. D. and Lesser, V. R.
Parallelism in AI problem solving: a case study of Hearsay-II.
IEEE Trans on Computers, C-26 :98-111, 1977.

[Gabriel 84] Gabriel, Richard P. and McCarthy, John.
Queue-based Multi-processing Lisp.
Proceedings of the ACM Symposium on Lisp and Functional programming :25 - 44, August, 1984.

[Halstead 84] Halstead, Robert H. Jr.
Implementation of Multilisp: Lisp on a Multiprocessor.
Proceedings of the ACM Symposium on Lisp and Functional programming :9 - 17, August, 1984.

[Hayes-Roth 85]Barbara Hayes-Roth.
Blackboard Architecture for Control.
Journal of Artificial Intelligence 26:251 - 321, 1985.

[Hewitt 73] Hewitt, C., P. Bishop, and R. Steiger.
A Universal, Modular Actor Formalism for Artificial Intelligence.
Proceedings of IJCAI-73 :235 - 245, 1973.

[KSL 85] Knowledge Systems Laboratory.
Knowledge Systems Laboratory 85, incorporating the Heuristic Programming Project.
KSL, Dept of Computer Science, Stanford University, 1985.

[Lesser 83] Lesser, Victor R. and Daniel D. Corkill.
The Distributed Vehicle Monitoring Testbed: A Tool for Investigation Distributed Problem Solving Networks.
The AI Magazine Fall:15 - 33, 1983.

[Nii 79] Nii, H. P. and N. Aiello.
AGE: A Knowledge-based Program for Building Knowledge-based Programs.
Proc. of IJCAI 6 :645 - 655, 1979.

[Nii 86] Nii, H. P.
Blackboard Systems.
AI Magazine 7:2, 1986.

[Rice 84] Rice, J. P.
The MXA user's and writer's companion
Systems Programming Ltd, The Charter, Abingdon, Oxon, UK, 1984.

[Rice 86] Rice, J. P.
The Poligon User's Manual.
Technical Report KSL-86-10, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

[Schoen 86] Schoen, Eric.
The CAOS System.
Technical Report KSL-86-22, Heuristic Programming Project, C. S. Dept., Stanford University, 1986.

The CAOS System

Eric Schoen

Knowledge Systems Lab
Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

The CAOS system is a framework in which multiprocessor expert systems may be developed. This report documents the principal ideas, programming model, and implementation of CAOS. In addition, we describe a working CAOS application, and discuss its performance over a class of (simulated) multiprocessor architectures.

1 Introduction and Overview

This report documents the CAOS system, a portion of a recent experiment investigating the potential of highly concurrent computing architectures to enhance the performance of expert systems. The experiment focuses on the migration of a portion of an existing expert system application from a sequential uniprocessor environment to a parallel multiprocessor environment.

The application, called ELINT, is a portion of a multi-sensor information fusion system, and was written originally in AGE[2], an expert system development tool based on the blackboard paradigm. For the purposes of this experiment, ELINT was reimplemented in CAOS, an experimental concurrent blackboard framework based on the explicit exchange of messages between blackboard agents.

CAOS, in turn, relies on services provided by the underlying machine environment. In the present set of experiments, the environment is a simulation of a concurrent architecture, called CARE [5]. CARE simulates a square grid of processing nodes, each containing a Lisp evaluator, private memory, and a communications subsystem; message-passing is the only means of interprocessor communication.

CAOS is principally an operating system, controlling the creation, initialization, and execution of independent computing tasks in response to messages received from other tasks. Figure 1 illustrates the relationship between the various software components of the experiment.

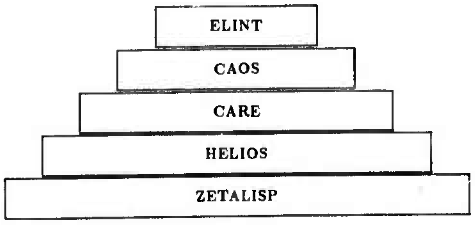


Figure 1: The relationship between ELINT, CARE, and CAOS

The following section briefly describes the salient features of the CARE environment. Section 3 discusses the ideas behind the CAOS framework. Section 4 summarizes the CAOS programming environment, and Section 5 describes its implementation. The final section details the results of our experiments.

2 An Overview of CARE

CARE is a highly-parameterized and well-instrumented multiprocessor simulation testbed, designed to aid research in alternative parallel architectures. It runs executes within Helios, a hierarchical, event-driven simulator which has been described elsewhere [3].

A typical CARE architecture is a grid of processing sites, interconnected by a dedicated communications network. For example, the research discussed in this paper was performed on square arrays of hexagonally connected processors (*e.g.*, each processor is connected to six of its eight nearest neighbors, excluding processors at the edges of the grid).

Each processing site consists of an *evaluator*, a general-purpose processor/memory pair, and an *operator*, a dedicated communications and process scheduling processor which shares memory with the evaluator. Application-level computations take place in the evaluator, a component which is treated as a "black box" Lisp processor. No portion of its interior is simulated; the host Lisp machine serves as the evaluator in each processing site. The operator performs two duties. As a communications processor, it is responsible for routing messages between processing sites. As a scheduling processor, it queues application-level processes for execution in the evaluator (we discuss the scheduling mechanism in greater detail below). The operator is simulated and instrumented in great detail.

CARE allows a number of parameters of the processor grid to be adjusted. Among these parameters are: the speed of the evaluator, the speed of the communications network, and the speed of the process-switching mechanism. By altering these parameters, a single processor grid specification can be made to simulate a wide variety of actual multiprocessor architectures. For example, we can experiment with the optimal level-of-granularity of problem decomposition by varying the speed of both process-switching and communications.

Finally, CARE provides detailed displays of such information as evaluator, operator, and communication network utilization, and process scheduling latencies. This instrumentation package informs developers of CARE applications of how efficiently their systems make use of the simulated hardware.

2.1 The CARE Programming Model

CARE programs are made up of processes which communicate by exchanging messages. Messages flow across *streams*, virtual circuits maintained by CARE. The following services are used by CAOS:

New Process: Creates a new process on a specified site, running a specified top-level function. A new stream is returned, enabling the "parent" of the process to communicate with its "child." Pointers

This research was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Eric Schoen was supported by a fellowship from NL Industries.

to the stream may be exchanged freely with other known processes on other sites.

New Stream: Creates a new stream whose target is the creating process.

Post Packet: Sends a message across a specified stream to a remote process.

Accept Packet: Returns the next message waiting on a specified stream. If no message is waiting when this operation is invoked, the invoking process is suspended and moved into the operator to await the arrival of a message.

Memory in each processing site is private. Ordinarily, intra-memory pointers may not be exchanged with processes in other sites. However, any pointer may be encapsulated in a *remote-address*, and may then be included in the contents of a message between sites. A remote address does not permit direct manipulation of remote structures; instead, it allows a process in one site to produce a local copy of a structure in another site.

Scheduling on a CARE node is entirely cooperative, and is based on message-passing. The message exchange primitives **post-packet** and **accept-packet** form the basis of process scheduling. A process wishing to block (yield control of the evaluator) does so by calling **accept-packet** to wait for a packet to arrive on a stream. The application program's scheduler awakens the process by calling **post-packet** to send a packet to the stream. The process is placed on the queue of processes waiting for the evaluator, and eventually regains control. The CAOS scheduler, which we describe in Section 5.3, is implemented in terms of this paradigm.

3 The CAOS Framework

CAOS is a framework which supports the execution of multi-processor expert systems. Its design is predicated on the belief that future parallel architectures will emphasize limited communication between processors rather than uniformly-shared memory. We expected such an architecture would favor coarse-grained problem decomposition, with little or no synchronization between processors. CAOS is intended for use in real-time data interpretation applications, such as continuous speech recognition, passive radar and sonar interpretation, etc [7,11].

A CAOS application consists of a collection of communicating *agents*, each responding to a number of application-dependent, predeclared messages. An agent retains long-term local state. Furthermore, an arbitrary number of processes may be active at any one time in a single agent.

Whereas the uniprocessor blackboard paradigm usually implies pattern-directed, demon-triggered knowledge source activation, CAOS requires explicit messaging between agents; the costs of automatically communicating changes in the blackboard state, as required by the traditional blackboard mechanism, could be prohibitively expensive in the distributed-memory multiprocessor environment. Thus, CAOS is designed to express parallelism at a very coarse grain-size, at the level of knowledge source invocation in a traditional uniprocessor blackboard system. It supports no mechanism for finer-grained concurrency, such as within the execution of agent processes, but neither does it rule it out. For example, we could easily imagine the *methods* which implement the messages being written in QLisp [8], a concurrent dialect of Common Lisp.

3.1 The Structure of CAOS Applications

A CAOS application is structured to achieve high degrees of concurrency in two principal manners: *pipelining* and *replication*. Pipelining is most appropriate for representing the flow of information between levels of abstraction in an interpretation system; replication provides means by which the interpretation system can cope with arbitrarily high data rates.

3.1.1 Pipelining

Pipelining is a common means of parallelizing tasks through a decomposition into a linear sequence of independent stages. Each stage is assigned to a separate processing unit, which receives the output from the previous stage and provides input to the next stage. Optimally, when the pipeline reaches a steady-state, each of its processors is busy performing its assigned stage of the overall task.

CAOS promotes the use of pipelines to partition an interpretation task into a sequence of interpretation stages, where each stage of the interpretation is performed by a separate agent. As data enters one agent in the pipeline, it is processed, and the results are sent to the next agent. The data input to each successive stage represents a higher level of abstraction.

Advantages of Pipelining Sequential decomposition of a large task is frequently very natural. Structures as disparate as manufacturing assembly lines and the arithmetic processors of high-speed computing systems are frequently based on this paradigm.

Pipelining provides a mechanism whereby concurrency is obtained without duplication of mechanism (that is, machinery, processing hardware, knowledge, etc). In an optimal pipeline of n processing elements, element 1 is performing work on task $t+n-1$ when element 2 is working on task $t+n-2$, and so on, such that element n is working on task t. As a result, the throughput of the pipeline is n times the throughput of a single processing element in the pipeline.

In the case of CAOS applications, the individual agents which compose an interpretation "pipeline" are themselves simple, but the overall combination of agents may be quite complex.

Disadvantages of Pipelining Unfortunately, it is often the case that a task cannot be decomposed into a simple linear sequence of subtasks. Some stage of the sequence may depend not only on the results of its immediate predecessor, but also on the results of more distant predecessors, or worse, some distant successor (*e.g.*, in feedback loops). An equally disadvantageous decomposition is one in which some of the processing stages take substantially more time than others. The effect of either of these conditions is to cause the pipeline to be used less efficiently. Both these conditions may cause some processing stages to be busier than others; in the worst case, some stages may be so busy that other stages receive no work at all. As a result, the n-element pipeline achieves less than an n-times increase in throughput. We discuss a possible remedy for this situation in the following subsection.

3.1.2 Replication

Concurrency gained through replication is ideally orthogonal to concurrency gained through pipelining. Any size processing structure, from individual processing elements to entire pipelines, is a candidate for replication. Consider a task which must be performed on average in time t, and a processing structure which is able to perform the task in time T, where $T > t$. If this task were actually a single stage in a larger pipeline, this stage would then be a bottleneck in the throughput of the pipeline. However, if the single processing structure which performed the task were replaced by T/t copies of the same processing structure, the effective time to perform the task would approach t, as required.

Advantages of Replication The advantages of replicating processing structure to improve throughput should be clear; n times the throughput of a single processing structure is achieved with n times the mechanism. Replication is more costly than pipelining, but it apparently avoids problems associated with developing a pipelined decomposition of a task.

Disadvantages of Replication Our works leads us to believe that such replicated computing structures are feasible, but not without drawbacks. Just as performance gains in pipelines are impacted by inter-stage dependencies, performance gains in replicated structures are impacted by inter-structure dependencies.

Consider a system composed of a number of copies of a single pipeline. Further, assume the actions of a particular stage in the pipeline affects each copy of itself in the other pipelines. In an expert system, for example, a number of independent pieces of evidence may cause the system to draw the same conclusion; the system designer may require that when a conclusion is arrived at independently by different means, some measure of confidence in the conclusion is increased accordingly. If the inference mechanism which produces these conclusions is realized as concurrently-operating copies of a single inference engine, the individual inference engines will have to communicate between themselves to avoid producing multiple copies of the same conclusions. A stringent consistency requirement between copies of a processing structure decreases the throughput of the entire system, since a portion of the system's work is dedicated to inter-system communication.

3.2 An Example

We close this section by describing the organization of ELINT, illustrating the benefits and drawbacks of the CAOS framework applied to this problem. ELINT is an expert system whose domain is the interpretation of passively-observed radar emissions. Its goal is to correlate a large number of radar observations into a smaller number of individual signal emitters, and then to correlate those emitters into a yet smaller number of clusters of emitters. ELINT is meant to operate in real time; emitters and clusters appear and disappear during the lifetime of an ELINT run. The basic flow of information in ELINT is through a pipeline of the various agent types, which we now describe in detail.

Observation Reader The observation reader is an artificat of the simulation environment in which ELINT runs. Its purpose is to feed radar observations into the system. The reader is driven off a clock; at each tick (1 ELINT "time unit"), it supplies all observations for the associated time interval to the proper observation handlers. This behavior is similar to that of a radar collection site in an actual ELINT setting.

Observation Handler The observation handlers accept radar observations from associated radar collection sites (in the simulated system, the observations come from the observation reader agent). There may be a large number of observation handlers associated with each collection site. The collection site chooses to which of its many observation handlers to pass an observation, based on some scheduling criteria such as random choice or round-robin.

Each observation contains an externally-assigned number to distinguish the source of the observation from other known sources (the observation id is usually, but not always, correct). In addition, each observation contains information about the observed radar signal, such as its quality, strength, line-of-bearing, and operating mode. The observation does *not* contain information regarding the source's speed, flight path, and distance; ELINT will attempt to determine this information as it monitors the behavior of each source over time.

When an observation handler receives an observation, it checks the observation's id to see if it already knows about the emitter. If it does, it passes the observation to the appropriate emitter agent which represents the observation's source. If the observation handler does not know about the emitter, it asks an emitter manager to create a new emitter agent, and then passes the observation to that new agent.

Emitter Manager There may be many emitter managers in the system. An emitter manager's task is to accept requests to create emitters with specified id numbers. If there is no such emitter in existence when the request is received, the manager will create one and return its "address" to the requesting observation handler. If there is such an emitter in existence when the request is received, the manager will simply return its address to the requestor. This situation arises when one observation handler requests an emitter than another observation handler had previously requested.

The reason for the emitter manager's existence is to reduce the amount of inter-pipeline dependency with respect to the creation of emitters. When ELINT creates an emitter, it is similar to a typical expert system's drawing a conclusion about some evidence; as discussed above, ELINT must create its emitters in such a way that the individual observation handlers do not end up each creating copies of the same emitter. Consider the following strategies the observation handlers could use to create new emitters:

1. The handlers could create the emitters themselves immediately. Since the collection site may pass observations with the same id to each observation handler, it is possible for each observation handler to create its own copy of the same emitter. We reject this method.

2. The handlers could create the emitters themselves, but inform the other handlers that they've done this. This scheme breaks down when two handlers try simultaneously to create the same emitter.

3. The handlers could rely on a single emitter manager agent to create all emitters. While this approach is safe from a consistency standpoint, it is likely to be impractical, as the single emitter manager could become a bottleneck in the interpretation.

4. The handlers could send requests to one of many emitter managers, chosen by some arbitrary method. This idea is nearly correct, but does not rule out the possibility of two emitter managers each receiving creation requests for the same emitter.

5. The handlers could send requests to one of many emitter managers, chosen through some algorithm which is invariant with respect to the observation id. This is in fact the algorithm in use in ELINT. The algorithm for choosing which emitter manager to use is based on a many-to-one mapping of observation id's to emitter managers.[1]

Emitters Emitters hold some state and history regarding observations of the sources they represent. As each new observation is received, it is added to a list of new observations. On a regular basis, the list of new observations is scanned for interesting information. In particular, after enough observations are received, the emitter may be able to determine its heading, speed, and location. The first time it is able to determine this information, it asks a cluster manager to either *match* the emitter to an old cluster or *create* a new cluster to hold the single emitter. Subsequently, it sends an update message to the cluster to which it belongs, indicating its current course, speed, and location.

Emitters maintain a qualitative confidence level of their own existence (*possible*, *probable*, and *positive*). If new observations are received often enough, the emitter will increase its confidence level until it reaches *positive*. If an observation is not received in the expected time interval, the emitter lowers its confidence by one step. If the confidence falls below *possible*, the emitter "deletes" itself, informing its manager, and any cluster to which it is attached.

Cluster Managers The cluster managers play much the same role in the creation of cluster agents as the emitter managers play in the creation of emitters. However, it is not possible to compute an invariant to be used as a many-to-one mapping between emitters. If ELINT were to employ multiple cluster managers, the best strategy for choosing which of the many managers would still result in the possible creation of multiple instances of the "same" cluster. Thus, we have chosen to run ELINT with a single cluster manager. Fortunately, cluster creation is a rare event, and the single cluster manager has never been a processing bottleneck.

As indicated above, requests from emitters to create clusters are specified as match requests over the extant clusters. Emitters are matched to clusters on the basis of their location, speed, and heading. However, the cluster manager does not itself perform this matching operation. Although it knows about the existence of each cluster it has created, it does not know if the cluster has changed course, speed, and/or

[1] The algorithm computes the observation id modulo the number of emitter managers, and maps that number to a particular manager.

direction since it was originally created. Thus, the cluster manager asks each of its clusters to perform a match.

If either none of the clusters responds with a positive match, a new cluster is created for the emitter; if one cluster responds positively, the emitter is added to the cluster, and is so informed of this fact; if more than one cluster responds positively, an error (or a mid-air collision) must have occured.

Clusters The radar emissions of clusters of emitters often indicates the actual behavior of the cluster. Cluster agents, therefore, apply heuristics about radar signals to determine whether the behaviors of the clusters they represent are threatening or not. This information, along with the course parameters of each radar source, is the "output" of the ELINT system. A cluster will delete itself if all constituent emitters have been deleted.

4 Programming in the CAOS Framework

CAOS is package of functions on top of Lisp. These functions are partitioned into three major classes:

- Those which declare agents.
- Those which initialize agents.
- Those which support communication between agents.

We now describe the CAOS operators for each of these classes.

4.1 Declaration of agents

Agents are declared within an inheritance network. Each agent inherits the characteristics of its (multiple) parents. The simplest agent, **vanilla-agent**, contains the minimal characteristics required of a functional CAOS agent. All other CAOS agents reference **vanilla-agent** either directly or indirectly. Another predeclared agent, **process-agenda-agent**, is built on top of **vanilla-agent**, and contains a priority mechanism for scheduling the execution of messages.

Application agents are declared by augmenting the following characteristics of the base or other ancestral agents:

Local Variables: An agent may refer freely to any variable declared local. In addition, each local variable may be declared with an initial value.

Messages: The only messages to which an agent may respond are those declared in this table. This simplifies the task of a resource allocator, which must load application code onto each CARE site.

Symbolically Referenced Agents: Some agents exist throughout a CAOS run. We call such agents *static*, and we allow code in agent message handlers to reference such agents by name. Before an agent begins running, each symbolic reference is resolved by the CAOS runtimes.

There are a number of additional characteristics; most of these are used by CAOS internally, and we will document these in the next section.

The basic form for declaring a CAOS agent is **defagent**. It has the form illustrated by Figure 2. The first element in each sublist is a keyword; there are a number of defined keywords, and their use in an agent declaration is strictly optional. An agent inherits the union of the keyword values of its parents for any unspecified keyword. Of those keywords which *are* specified, some are combined with the union of the keyword values of the agent's parents, and others supersede the values in the parents. Figure 3 contains the declaration of the emitter agent, one of the most complex examples in ELINT.

As we discuss in the next section, **defagent** forms are translated by CAOS into Flavors **defflavor** forms [4]. CAOS messages are then defined using the **defmethod** function of ZETALISP. These methods are free to reference the local variables declared in the **defagent** expression.

```
(defagent agent-name (parent_1 ··· parent_n)
  (localvars variable_1 ··· variable_n)
  (messages message_1 ··· message_n)
  (symbolically-referenced-agents agent_1 ··· agent_n))
```

Figure 2: The basic form of **defagent**

```
(defagent el-emitter (process-agenda-agent)
  (localvars
    (process-agenda '(el-undo-collection-id-error
                      el-change-cluster-association
                      el-emitter-update-on-time-tick
                      el-initialize-emitter
                      el-update-emitter-from-observation))
    (last-observed -1000000)
    (cluster-manager 'cluster-manager-0)
    manager
    id
    type
    observed
    fixes
    last-heading
    last-mode
    confidence
    cluster
    new-observations-since-time-tick-flag
    id-errors
    gc-flag)
   (messages
    el-update-emitter-from-observation
    el-initialize-emitter
    el-change-cluster-association
    el-undo-collection-id-error)
   (symbolically-referenced-agents
    el-collection-reporter-0
    el-correlation-reporter-0
    el-threat-reporter-0
    el-cluster-manager-0
    el-cluster-manager-1
    el-cluster-manager-2
    el-big-ear-handler
    el-gotcha-handler
    el-emitter-trace-reporter-0))
```

Figure 3: The **emitter** agent

```
(caos-initialize
  ((agent - name1 agent - class site - address)
    ...)
  ((initial - message1)
    ...))
```

Figure 4: The basic CAOS initialization form

```
(caos-initialize
  ((el-observation-reader-0 el-observation-reader (2 2))
   (el-big-ear-handler-1 el-observation-handler (1 1))
   (el-big-ear-handler-2 el-observation-handler (1 1))
   (el-gotcha-handler-1 el-observation-handler (1 2))
   (el-gotcha-handler-2 el-observation-handler (1 2))
   (el-emitter-manager-0 el-emitter-manager (2 1))
   (el-emitter-manager-1 el-emitter-manager (2 2))
   (el-collection-reporter-0 el-collection-reporter (1 2))
   (el-correlation-reporter-0 el-correlation-reporter
                               (1 3))
   (el-threat-reporter-0 el-threat-reporter (1 3))
   (el-emitter-trace-reporter-0 el-emitter-trace-reporter
                                  (3 2))
   (el-cluster-trace-reporter-0 el-cluster-trace-reporter
                                  (3 1))
   (el-cluster-manager-0 el-cluster-manager (2 1)))
  ((post el-observation-reader-0 nil
         'el-open-observation-file
          *elint-data-file*)
   (post el-collection-reporter-0 nil
          'el-initialize-reporter t
          "elint:reports;collections.output")
   (post el-correlation-reporter-0 nil
          'el-initialize-reporter t
          "elint:reports;correlations.output")
   (post el-threat-reporter-0 nil
          'el-initialize-reporter t
          "elint:reports;threats.output")
   (post el-emitter-trace-reporter-0 nil
          'initialize-trace-reporter t
          "elint:reports;emitter.traces")
   (post el-cluster-trace-reporter-0 nil
          'initialize-trace-reporter t
          "elint.reports;cluster.traces")))
```

.Figure 5: The initialization declaration for ELINT.

4.2 Initialization of agents

The initial CAOS configuration is specified by the **caos-initialize** operator, which takes the form illustrated by figure 4; for example, figure 5 is ELINT's initialization form.

The first portion of the form creates the static agents. In figure 5, a static agent named **el-gotcha-handler-1**, an instance of the class **el-observation-handler**, is created on the CARE site at coordinates (1, 2) in the processor grid.

The second portion of the form is a list of LISP expressions to be evaluated sequentially when CAOS's initialization phase is complete. Each expression is intended to send a message to one of the static agents declared in the first part of the form. These messages serve to initialize the application; in figure 5, the initialization messages open log files and start the processing of ELINT observations.

Agents may also be created dynamically. The **create-agent-instance** function accepts an agent class name and a location specification;[2] the **remote-address** of the newly-created agent is returned. While dynamically created agents may *not* be referenced symbolically, their **remote-address**'s may be exchanged freely.

4.3 Communications Between Agents

Agents communicate with each other by exchanging messages. CAOS does not guarantee that messages reach their destinations: due to excessive message traffic or processing element failure, messages may be delayed or lost during routing. It is the responsibility of the application program to detect and recover from lost messages. Commensurate with the facilities provided by CARE, messages may be tagged with routing priorities; however, higher priority messages are not guaranteed to arrive before lower-priority messages sent concurrently.

Two classes of messages are defined: those which return values (called *value-desired* messages), and those which do not (called *side-effect* messages). The value-desired-messages are made to return their values to a special cell called a *future*. Processes attempting to access the value of a future are blocked until that future has had its value set. It is possible for the value of a future to be set more than once, and it is possible for there to be multiple processes awaiting a future's value to be set.[3]

4.3.1 Sending messages

The CARE primitive **post-packet**, which sends a packet from one process to another, is employed in CAOS to produce three basic kinds of message sending operations:

post: The **post** operator sends a side-effect message to an agent. The sending process supplies the name or pointer to the target agent, the message routing priority, the message name and arguments. The sender continues executing while the message is delivered to the target agent.

post-future: The **post-future** operator sends a value-desired message to the target agent. The sending process supplies the same parameters as for **post**, and is returned a pointer to the future which will eventually by set by the target agent. As for **post**, the sender continues executing while the message is being delivered and executed remotely.

A process may later check the state of the future with the **future-satisfied?** operator, or access the future's value with the **value-future** operator, which will block the process until the future has a value.

post-value: The **post-value** operator is similar to the **post-future** operator; however, the sending process is delayed until the target agent has returned a value. **post-value** is defined in terms of **post-future** and **value-future**.

4.3.2 Detecting Lost Messages

It is possible to detect the loss of value-desired messages by attaching a timeout to the associated future. The functions **post-clocked-future** and **post-clocked-value** are similar to their untimed counterparts, but allow the caller to specify a *timeout* and *timeout action* to be performed if the future is not set within the timeout period. Typical actions include setting the future's value with a default value, or resending the original message using the **repost** operator.

4.3.3 Sending to Multiple Agents

There exist versions of the basic posting operators which allow the same message to be sent to multiple agents.[4] **multipost** sends a side effect message to a list of agents; **multipost-future** and **multipost-value** send a value-desired message to a list of agents. In the latter case, the associated future is actually a list of futures; the future is not considered set until all target agents have responded. The value of such a message is an association-list; each entry in the list is composed of an agent name or **remote-address** and the returned message value from that

[2] Currently, agents may be created at or near specified CARE sites. CAOS makes no attempt at dynamic load balancing.

[3] Futures were also used in QLisp and Multilisp [9]. The HEP Supercomputer [6] implemented a simple version of futures as a process synchronization mechanism.

[4] Neither CAOS nor CARE currently support a *predicated multicast* mode, wherein messages would sent to all agents satisfying a particular predicate; messages can only be sent to a fully-specified list of agents.

agent. There exist clocked versions of these functions (called, naturally, **multipost-clocked-future** and **multipost-clocked-value**) to aid in detecting lost multicast messages.

4.4 Communications Between Processes

Processes in each agent communicate using the shared local variables declared in the agent. Besides sharing previously computed results this way, processes may also share the results of ongoing computations.

Consider the following scenario: within an agent, some process is currently computing some answer. At the same time, another process begins executing, and realizes somehow that the answer it needs to compute is the same answer the other process is already computing. The second process could take one of two actions: it could continue computing the answer, even though this would mean redundant work, or it could wait for the first process to complete, and return its answer. The second approach is feasible, but it does tie up resources in the form of an idle process.

The CAOS operators **attach** and **my-handle** offer a third alternative solution. If a process knows it may ultimately produce an answer needed by more than one requesting agent, it obtains its "handle" (Section 5.4) by calling **my-handle**, and places it in a table for other processes to reference. Any other process wishing to return the same answer as the first calls **attach**, with the first process's handle as argument. The first process returns its answer to all requesting agents waiting for answers from the other processes, and the other processes return no value at all.

4.5 What CAOS Offers Over CARE

CAOS is a large system. It is reasonable to ask what advantages there are to programming in CAOS as opposed to programming in CARE. We believe there are three major advantages:

Clarity: The framework in which an agent is declared makes explicit its storage requirements and functional behavior. In addition, the agent concept is a helpful abstraction at which to view activity in a multiprocessing software architecture. The concept lets us partition a flat collection of processes on a site into groups of processes attached to agents on a site. CAOS guarantees the only interaction between processes attached to different agents is by message-passing.

Convenience: The programmer is freed from interfacing to CARE's low-level communications primitives. As we said earlier, CAOS is basically an operating system, and as such, it shields the programmer from the same class of details a conventional operating system does in a conventional hardware environment.

Flexibility: Currently, CARE schedules processes in a strict first-in, first-out manner. CAOS, on the other hand, can implement arbitrary scheduling policies (though at a substantial performance cost; we discuss this in Section 6).

5 The Runtime Structure of CAOS

CAOS is structured around three principal levels: site, agent, and process. Two of these levels–site and process–reflect the organization of CARE; the remaining (agent) level is an artifact of CAOS. We discuss first the general design principles underlying CAOS, and then describe in greater detail the functions and structure of each of CAOS's levels.

5.1 General Design Principles

The implementation of CAOS described in this paper is written in ZETALISP, a dialect of Lisp which runs on a number of commercially available single-user Lisp workstations. ZETALISP includes an object-oriented programming tool, called Flavors, which has proved to be a very powerful facility for structuring large Lisp applications.

In Flavors, the behavior of an object is described by templates known as *classes*. An *instance*, a representation of an individual object, is created by instantiating a class. Instances respond to messages defined by their class, and contain static local storage in the form of *instance variables*. Classes are defined within an inheritance network; each instance contains the instance variables and responds to the messages defined in its class, as well as those of the classes from which its class inherits.

An appropriate usage for Flavors is the modelling of the behavior of objects in some (not necessarily real) world. For example, CAOS site and agents structures are realized as Flavors instances. The characteristics to be modelled are codified in instance variables and message names. In a well-designed application, messages and variables are consistently named; thus, the implementation of a particular behavior is totally encapsulated in the anonymous function which responds to a message.

5.1.1 Extending the Notion

In some sense, a Flavors instance is an abstract data type. The instance holds state, and provides advertised, public interfaces (messages) to functions which change or access its state. The internal data representation and implementations of the access functions are private.

In Flavors, the abstract data type notion is unavailable within an individual instance. Frequently, the individual instance variables hold complex structures (such as dictionaries and priority queues) which ought to be treated as abstract data types, but there exist no common means within the standard Flavors mechanism for doing so.

CAOS, however, supports such a mechanism, by providing a means of sending messages to instance variables (rather than to the instances themselves). The instance variables are thus able to store anonymous structures, which are initialized, modified, and accessed through messages sent to the variable. Similar mechanisms exist in the Unit Package [14] and in the STROBE system [13], both frameworks for representing structured knowledge.

The CAOS environment includes a number of abstract data types which were found to be useful in supporting its own implementation. The most commonly used are:

Dictionary: The dictionary is an association list. It responds to put, **get**, **add**, **forget**, and **initialize** messages.

Sorted Dictionary: The sorted-dictionary is also implemented as an association list, and responds to the same messages as does the standard dictionary. However, the sorted-dictionary invokes a user-supplied priority function to merge new items into the dictionary (higher-priority items appear nearer the front of the dictionary). This dictionary is able to respond to the **greatest** message, which returns the entry with the highest priority, and to the **next** message, which returns the entry with the next-highest priority as compared to a given entry.

The sorted-dictionary is used primarily to hold time-indexed data which may be collected out-of-order (e.g. when data for time $n+1$ may arrive before data for time n).

Hash Dictionary: The hash-dictionary is implemented with a hash table, and responds to the same messages as the unsorted association list dictionary.

Queue: The queue data type is a conventional first-in, first-out storage structure. The put message enqueues an item on the tail of the queue, while the **get** message dequeues an item from the head of the queue.

Priority Queue: The priority-queue data type supports a dynamic heap-sort, and is implemented as a partially-ordered binary tree. It responds to put, **get**, and **initialize** messages. Associated with the queue is a function which computes and compares the priority of two arbitrary queue elements; this function drives the rebalancing of the binary tree when elements are added or deleted.

Monitor: A monitor provides mutual exclusion within a dynamically-scoped block of Lisp code. It is similar in implementation to the monitors of Interlisp-D and Mesa [10].

If the monitor is unlocked, the **obtain-lock** message stores the caller's process id as the monitor's owner, and marks the monitor as locked; otherwise, if the monitor is locked, the `obtain-lock` message places the caller's process id on the tail of the monitor's waiting queue, and suspends the calling process.

The `release-lock` message removes the process id from the head of the monitor's waiting queue, marks the monitor's owner to be that id, and reschedules the associated process.

Monitors are normally accessed using the `with-monitor` form, which accepts the name of an instance variable containing a monitor, and which cannot be entered until the calling process obtains ownership of the monitor. The `with-monitor` form guarantees ownership of the monitor will be relinquished when the calling process leaves the scope of the form, even if an error occurs.

5.2 The CAOS Site Manager

The site manager consists of a Flavors instance containing information global to the site–information needed by all agents located on the site. In addition, the site manager includes a CARE-level process which performs the functions of creating new agents and translating agent names into agent addresses, as described below.

The following instance variables are part of the site manager:

`incoming-stream`: This instance variable contains the CARE input stream address on which the site manager process listens for requests. Agents needing to send messages to their site manager may reference this instance variable in order to discover the address to which to direct site requests.

`static-agent-stream-table`: This instance variable is a dictionary which maps agent names into the CARE streams which may be used to communicate with the agents. The entries in this dictionary reflect statically-created agents; new entries are added as the result of `new-initial-agent-online` messages directed to the site (see below). The dictionary is used to resolve agent name-to-address requests from agents created locally.

`unresolved-agent-stream-table`: The site manager keeps track of agent names it is not able to translate to addresses by placing unsatisfiable `request-symbolic-reference` requests in this dictionary. The keys of the dictionary are unresolvable agent names. As the agent names become resolvable, the unsatisfied requests are satisfied, and the corresponding entries are removed from the dictionary.

After the initialization phase of a CAOS application has completed, there will be no entries in this dictionary in any of the sites.

`local-agents`: This instance variable is a dictionary whose keys are the names of agents located on the site, and whose values are pointers to the Flavors instances which represent each agent. `local-agents` is used only for debugging and status-reporting purposes.

`free-process-queue`: When a CARE process which was created to service a request finishes its work, it tries to perform another task for the agent in which it was created. If the agent has no work to do, the process suspends itself, after enqueuing identifying information in this instance variable, which holds a queue abstract data type. When any agent on the same site needs a new process to service some request, it checks this queue first; if there are any suspended (free) processes waiting in this queue, it dequeues one and gives it a task to perform. If this queue is empty, the agent asks CARE to create a new process.

The site manager responds to the following messages:

`new-initial-agent-online`: As each static agent starts running during initialization of a CAOS run, it broadcasts its name and CARE input stream to every site in the system, using this message. The correspondence between the sending agent's name and address is placed in the `static-agent-stream-table` dictionary for future reference by agents located on the receiving sites. If any agents have placed requests for this new agent in the `unresolved-agent-stream-table`, messages containing the new agent's name and address are sent to the waiting agents.

`request-symbolic-reference`: Whenever a static agent is created, it runs an initialization function, which among other tasks, caches needed agent name-to-address translations. For each translation, the agent sends this message to its site manager. If the site manager can resolve the name upon receipt of the message, it responds immediately; otherwise, it queues the request in the `unresolved-agent-stream-table`, and defers answering until it is able to satisfy the request. The requesting agents waits until it has received the answer before requesting another translation.

`make-new-agent`: This message is sent to a site to cause a new agent to be created during the course of a CAOS run. The site manager creates the new (dynamic) agent and returns the agent's input stream to the sender of this message. The newly-created agent is *not* placed in the `static-agent-stream-table`; thus, the only way to advertise the existence of such a dynamically-created agent is by the creator of an agent passing the returned input stream to other agents.

5.3 The CAOS Agent

As discussed above, CAOS agents are implemented as Flavors instances. Their class definitions are defined by translating `defagent` expressions into `defflavor` expressions. CAOS itself defines two basic agent classes: `vanilla-agent` and `process-agenda-agent`. `vanilla-agent` defines the minimal agent; `process-agenda-agent` is defined in terms of `vanilla-agent`, but adds the ability to assign priorities to messages.[5] These basic agents are fully-functional, but lack domain-specific "knowledge," and cannot be used directly in problem solving applications.

As stated in the previous section, a CAOS agent is a multiple-process entity. Most of these processes are in created in the course of problem-solving activity; we refer to these as *user processes*. At runtime, however, there are always two special processes associated with each CAOS agent. One of these processes monitors the CARE stream by which the agent is known to other agents. The other participates in the scheduling of user processes. We shall refer to the first of these processes as the agent *input monitor*, and to the second of these processes as the agent *scheduler*. We explain in detail the functioning of these two processes in the next subsection.

We describe here the role of important instance variables in a basic CAOS agent:

`self-address`: This instance variable is an analogue of Flavors' `self` variable. Whereas `self` is bound to the Flavors instance under which a message is executing, `self-address` is bound to the stream of the agent under which a CAOS message is executing. Thus, an agent can post a message to itself by posting the message to `self-address`.

`runnable-process-stream`: This instance variable points to the stream on which the scheduler process listens. Processes which need to inform the scheduler of various conditions do so by sending CARE-level messages to this stream.

`running-processes`: This variable holds the list of user processes which are currently executing within the agent. The current CARE architecture supports only a single evaluator on each site. CAOS tries to keep a number of user processes ready to execute at all times; thus, the single CPU is kept as busy as possible.

`runnable-process-list`: A priority queue containing the runnable user processes. As a process is entered on the queue, its priority is calculated to determine its ranking in the partial ordering. There are two available priority evaluation functions: the first computes the priority based solely on the time the process entered the system; the second considers the assigned priority of the executing message before considering the entry time of the process. These two functions are used to implement the scheduling algorithms of the `vanilla-agent` and the `process-agenda-agent`, respectively.

[5] This is important for applications in which one agent must respond rapidly to a posting from another agent. Assigning a message a high priority will cause that message to be processed ahead of any other messages with lower priorities.

`scheduler-lock`: The scheduler data structures are subject to modification by any number of processes concurrently. The `scheduler-lock` is a monitor which provides mutual exclusion against simultaneous access to the scheduler database.

5.4 The CAOS Process

In this subsection, we describe the mechanism by which CAOS user processes are scheduled for execution on CARE sites. User processes are created in response to messages from other agents. Associated with each user process is a data structure called a `runnable-item`. The `runnable-item` contains the following fields:

`message-name`, `-args`, `-id`, `-answer-targets`: These fields store the information necessary to handle a message request and send the resulting answer back to the proper agents.

`for-effect`: This field is a boolean, and indicates whether the message is being executed for effect or value. This corresponds directly to the source of the message coming from a `post` operation or a `post-future` operation.

`state`: This field indicates the state of the process. The possible states that a process may enter, and the finite state machine which defines the state transition are discussed in the next subsection.

`context`: This field contains a pointer to the CARE stream upon which the process waits when it not runnable. A process (such as the scheduler) wishing to wake another process simply sends a message to this stream. The suspended process will thus be awakened (by CARE).

`time-stamp`: This field contains the time at which the process entered the system. It is used by the functions which calculate the execution priority of processes.

The CAOS scheduler's only handle on a process is the process's `runnable-item`. In fact, the only communication between a user process and the CAOS scheduler consists of the exchange of `runnable-item`'s.

5.5 Flow of Control

In the following, we detail how a user process, the CAOS input monitor, and the CAOS scheduler interact to process a message request from a remote agent. For purposes of exposition, we assume the following sequence of events:

1. An agent, `agent-1`, executes a `post` operation, with `agent-2` as the target. The posting is for the message named `message-a`.

2. `agent-2` receives and executes the posting. In order to complete the execution of `message-a`, it must perform a `post-value` operation to a third agent, `agent-3`.

We begin at the point where `agent-1` has performed its `post` operation.

5.5.1 Input Processing

The input monitor process handles requests and responses from remote agents. When the message from `agent-1` enters `agent-2`, its input monitor creates a new `runnable-item` to hold the state of the request. The message name, arguments, id, and answer targets are copied from the incoming message into the `runnable-item`. The `runnable-item`'s state is set to `never-run`, and its time stamp is set to the current time. In order to queue the message for execution, the input monitor takes one of two actions.

If the agent's `runnable-process-list` is empty, the `runnable-item` is sent in a message to the agent scheduler process (by sending the item in a message to the stream whose address is found in the agent's `runnable-process-stream` instance variable). When the agent's `runnable-process-list` is empty, the scheduler process is guaranteed to be waiting for messages sent to the scheduler stream, and hence, will be awakened by the message sent from the input monitor. The scheduler then computes the priority of the message, and places the `runnable-item` in its `runnable-process-list`.

If the agent's `runnable-process-list` is *not* empty, the input monitor computes the message's priority and places the `runnable-item` on the `runnable-process-list` itself. When the queue is not empty, it is guaranteed that the scheduler will examine the queue sometime in the future to make scheduling decisions; thus, it is not necessary to send any messages to the scheduler to inform it of the existence of new processes.

5.5.2 Creating Processes

Eventually, the newly-created `runnable-item` will reach the head of `agent-2`'s `runnable-process-list`. At this time, there is still no process associated with the item, so the scheduler creates a process using the facilities of CARE, adds the process to the `running-processes` list, and passes it its `runnable-item`. The process will eventually gain control of the evaluator, and will set the state of its `runnable-item` to `running`. It then begins executing the requested posting.

5.5.3 Requesting Remote Values

At some point, the process executing on `agent-2` requires a value from `agent-3`, and performs a `post-value` operation to acquire it. The process looks up the address of `agent-3`, and posts a message which contains the appropriate message name, arguments, id, and answer target. The `message-id` unambiguously identifies the `future` upon which the process will be waiting for the value to be returned. The answer target is the agent's own `self-address`; when the answer is received by the input monitor process, it will be forwarded to the appropriate future, and the process will be reawakened.

In the meantime, the process sets its state to `suspended`, removes its `runnable-item` from the `running-processes` list, and appends it to the list of processes already waiting for the future to be satisfied. If the `runnable-process-list` is not empty, the suspending process wakes the process at the head of the queue.[6] The suspending process then waits for a message on its wakeup stream, the stream whose address is in the `context` field of its `runnable-item`.

5.5.4 Answer Processing

Some time later, `agent-3` will have completed its computations, and will have returned the desired answer to `agent-2`. The answer will be received by `agent-2`'s input monitor process, which will recognize the input as a value to be placed in a future. The input monitor sets the value field of the appropriate future, and moves the `runnable-items` of the processes waiting on the future to the `runnable-process-list`.

If the queue was previously empty, the agent must have been (or will soon be) entirely idle; thus, the `runnable-items` are sent to the scheduler in a message, causing the scheduler to be reawakened. If the queue was not previously empty, the agent must be busy, so the items are simply added to the queue according to their priorities. In both cases, the `runnable-items` are placed in the `runnable` state.

5.5.5 Reawakening Suspended Processes

When the `runnable` `runnable-item` reaches the head of `agent-2`'s `runnable-process-list`, a message (which contains no useful information) is sent to its associated process's wakeup stream. As a result, process eventually wakes up, gains control of the evaluator, and sets its state to `running`.

5.5.6 Completing Computation

A process may perform any number of `post`, `post-future`, or `post-value` operations during its lifetime. Eventually, however, the process

[6] In effect, the process takes on the role of the scheduler. Although the system would continue to work with only a designated scheduler process performing scheduler duties, this arrangement permits scheduling to take place with minimal latency. As a result, fewer evaluator cycles are wasted waiting for the scheduler process to run the next user process.

will complete, having computed a value which may or may not be sent back to the requesting agent. If the process was suspended for any portion of its lifetime, another process may have attached to it; in this case, the process may have more than one requesting agent to which to return an answer.

Before the process terminates, it examines the head of the `runnable-process-list`. If the queue is empty, the process simply goes away. If the `runnable-item` at the head of the queue is `runnable`, it sends the appropriate message to awaken the associated process. Finally, if the item is `never-run`, the process makes itself the process associated with this new `runnable-item`, and executes the new message in its own context.[7] Barring this possibility, the process "queues" itself on a free process queue associated with the site manager; when a new process is needed by an agent on the site, one is preferentially removed from this queue and recycled before a entirely new process is created. This way, processes, which are expensive to create, are reused as often as possible.

6 Results and Conclusions

The CAOS system we have described has been fully implemented and is in use by two groups within the Advanced Architectures Project. CAOS runs on the Symbolics *3600* family of machines, as well as on the Texas Instruments *Explorer* Lisp machine. ELINT, as described in Section 3.2, has also been fully implemented. We are currently analyzing its performance on various size processor grids and at various data rates.

6.1 Evaluating CAOS

CAOS is a rather special-purpose environment, and should be evaluated with respect to the programming of concurrent real-time signal interpretation systems. In this section, we explore CAOS's suitability along the following dimensions:

- Expressiveness
- Efficiency
- Scalability

6.1.1 Expressiveness

When we ask that a language be suitably *expressive*, we ask that its primitives be a good match to the concepts the programmer is trying to encode. The programmer shouldn't need to resort to low-level "hackery" to implement operations which ought to be part of the language. We believe we have succeeding in meeting this goal for CAOS (although to date, only CAOS's designers have written CAOS applications). Programming in CAOS is programming in Lisp, but with added features for declaring, initializing, and controlling concurrent, real-time signal interpretation applications.

6.1.2 Efficiency

CAOS has a very complicated architecture. The lifetime of a message, as described in Section 5.5, involves numerous processing states and scheduler interventions. Much of this complexity derives from the desire to support alternate scheduling policies within an agent. The cost of this complexity is approximately one order of magnitude in processing latency. For the common settings of simulation parameters, CARE messages are exchanged in about 2–3 milliseconds, while CAOS messages require about 30 milliseconds. It is this cost which forces us to decompose applications coarsely, since more fine-grained decompositions would inevitably require more message traffic.

We conclude that CAOS does not make efficient use of the underlying CARE architecture. A compromise, which we are just beginning to explore, would be to avoid the complex flow of control described in Section 5.5 in agents whose scheduling policies are the same as CARE's (FIFO). In such agents, we could reduce the CAOS runtimes to simple functional interfaces to CARE. We anticipate such an approach would be much more efficient.

6.1.3 Scalability

A system which scales well is one whose performance increases commensurately with its size. Scalability is a common metric by which multiprocessor hardware architectures are judged: does a 100-processor realization of a particular architecture perform 10 times better than a 10-processor realization of the same architecture? Does it perform 5 times better? Only just as well? Or *Worse?* In hardware systems, scalability is typically limited by various forms of *contention* in memories, busses, etc. The 100-processor system might be slower than the 10-processor system because all interprocessor communications are routed through an element which is only fast enough to support 10 processors.

We ask the same question of a CAOS application: does the throughput of ELINT, for example, increase as we make more processors available to it? This question is critical for CAOS-based real-time interpretation systems; our only means of coping with arbitrarily large data rates is by increasing the number of processors. Section 6.2 discusses this issue in detail.

We believe CAOS scales well with respect to the number of available processors. The potential limiting factors to its scaling are *(1)*, increased software contention, such as inter-pipeline bottlenecks described in Section 3.1.2, and *(2)*, increased hardware contention, such as overloaded processors and/or communication channels. Software contention can be minimized by the design of the application. Communications contention can be minimized by executing CAOS on top of an appropriate hardware architecture (such as that afforded by CARE); CAOS applications tend to be coarsely decomposed–they are bounded by computation, rather than communication–and thus, communications loading has never been a problem.

Unfortunately, processor loading remains an issue. A configuration with poor *load balancing*, in which some processors are busy, while others are idle, does not scale well. Increased throughput is limited by contention for processing resources on overloaded sites, while resources on unloaded sites go unused. The problem of automatic load balancing is not addressed by CAOS; agents are assigned to processing sites on a round-robin basis, with no attempt to keep potentially busy agents apart.

6.2 Evaluating ELINT Under CAOS

Our experience with ELINT indicates the primary determiner of throughput and answer-quality is the strategy used in making individual agents cooperate in producing the desired interpretation. Of secondary importance is the degree to which processing load is evenly balanced over the processor grid. We now discuss the impact of these factors on ELINT's performance.

The following three strategies were used in our experiments:

NC: This strategy represents limited inter-agent control. No attempt is made to prevent concurrent creation of multiple copies of the "same" agent (this possibility arises when multiple requests to create the agent arrive simultaneously at a single manager). As a result, multiple, non-communicating copies of an abstraction pipeline are created; each receives a only portion of the input data it requires. The NC strategy was expected to produce poor results, and was intended only as a baseline against which to compare more realistic control strategies.

CC: In this strategy, the manager agents assure that only one copy of a agent is created, irrespective of the number of simultaneous creation requests; all requestors are returned pointers to the single new agent. Originally, we believed the CC (for "creation control") strategy would be sufficient for ELINT to produce correct high-level interpretations.

CT: The CT ("creation *and* time control") strategy was designed to manage skewed views of real-world time which develop in agent

[7] This is another situation in which an application process performs scheduling duties.

ELINT *Performance Dimension*	*Control Type/Grid Size*					
	NC	CC	CC	CT	CT	CT
	4 × 4	4 × 4	6 × 6	2 × 2	4 × 4	6 × 6
FALSE ALARMS	1	0	0	0	0	0
REINCARNATION	49	42	2	0	0	0
CONFIDENCE LEVEL	19	20	90	89	93	95
FIXES	48	42	99	100	100	100
FUSION	0	0	77	85	88	89

Table 1: Quality of ELINT performance of various grid sizes and control strategies (1 ELINT time unit = 0.1 seconds).

Control Type	*Simulated Time (sec)*		
	2 × 2	4 × 4	6 × 6
NC		> 11.19[8]	
CC		10.87	5.12
CT	11.80	8.10	4.17

Table 2: Simulated time required to complete an ELINT run (1 ELINT time unit = 0.1 seconds).

pipelines. In particular, this strategy prevents an **emitter** agent from deleting itself when it has not received a new observation in a while, yet some **observation-handler** agent has sent the **emitter** an observation which it has yet to receive.

Table 1 illustrates the effects of various control strategies and grid sizes. The table presents six performance attributes by which the quality of an ELINT run is measured.

False Alarms: This attribute is the percentage of **emitter** agents that ELINT should not have hypothesized as existing.

ELINT was not severely impacted by false alarms in any of the configurations in which it was run.

Reincarnation: This attribute is the percentage of recreated **emitter** agents (*e.g.*, **emitters** which had previously existed but had deleted themselves due to lack of observations). Large numbers of reincarnated **emitters** indicate some portion ELINT is unable to keep up with the data rate (*i.e.*, the data rate may be too high globally, so that all **emitters** are overloaded, or the data rate may be too high locally, due to poor load balancing, so that some subset of the **emitters** are overloaded).

The CT control strategy was designed to prevent reincarnations; hence, none occurred when CT was employed on any size grid. When CC was used, only the 6 × 6 grid was large enough for ELINT to keep up with the input data rate.

Confidence Level: This attribute is the percentage of correctly-deduced confidence levels of the existence of an **emitter**.

The correct calculation of confidence levels depends heavily on the system being able to cope with the incoming data rate. One way to improve confidence levels was to use a large processor grid. The other was to employ the CT control strategy, since fewer reincarnations result in fewer incorrect (*e.g.*, too low) confidence levels.

Fixes: This attribute is the percentage of correctly-calculated fixes of an **emitter**.

Fixes can be computed when an **emitter** has seen at least two observations in the same time interval. If an **emitter** is undergoing reincarnation, it will not accumulate enough data to regularly compute fixes. Thus, the approaches which minimized reincarnation maximized the correct calculation of fix information.

Control Type	*Message Count*		
	2 × 2	4 × 4	6 × 6
NC		> 16118	
CC		7375	
CT	4516	4703	4616

Table 3: Number of messages exchanged during an ELINT run (1 ELINT time unit = 0.1 seconds).

GRID SIZE	1 × 1	2 × 2	3 × 3	4 × 4	5 × 5	6 × 6
SIMULATED TIME (sec)	9.42	3.20	1.49	0.74	0.52	0.56

Table 4: Overall Simulation Times for CT Control Strategy (1 ELINT time unit = 0.01 seconds, debugging agents turned off).

Fusion: This attribute is the percentage of correct clustering of **emitter** agents to **cluster** agents.

The correct computation of fusion appeared to be related, in part, to the correct computation of confidence levels. The fusion process is also the most knowledge-intensive computation in ELINT, and our imperfect results indicate the extent to which ELINT's knowledge is incomplete.

We interpret from Table 1 that control strategy has the greatest impact on the quality of results. The CT strategy produced high-quality results irrespective of the number of processors used. The CC strategy, which is much more sensitive to processing delays, performed nearly as well only on the 6 × 6 processor grid. We believe the added complexity of the CT strategy, while never detrimental, is only beneficial when the interpretation system would otherwise be overloaded by high data rates or poor load balancing.

Tables 2 and 3 indicate that cost of the added control in the CT strategy is far outweighed by the benefits in its use. Far less message traffic is generated, and the overall simulation time is reduced (In Table 2, the last observation is fed into the system at 3.6 seconds; hence, this is the minimum possible simulated run time for the interpretation problem).

Finally, Table 4 illustrates the effect of processor grid size when the CT control strategy is employed. This table was produced with the data rate set ten times higher than that used to produce tables 1–3; the minimum possible simulated run time for the interpretation problem is 0.36 seconds. The speedup achieved by increasing the processor grid size is nearly linear with the square root of the size; however, the 6 × 6 grid was slightly slower than the 5 × 5 grid. In this last case, we believe the data rate was not high enough to warrant the additional processors.

6.3 Unanswered Questions

CAOS has been a suitable framework in which to construct concurrent signal interpretation systems, and we expect many of its concepts to be useful in our future computing architectures. Of principal concern to us now is increasing the efficiency with which the underlying CARE architecture is used. In addition, our experience suggests a number of questions to be explored in future research:

- What is the appropriate level of granularity at which to decompose problems for CARE-like architectures?
- What is the most efficient means to control the actions of concurrent problem solvers when necessary?
- How can flexible scheduling policies be implemented without significant loss of efficiency? What is the impact on problem solving if alternate scheduling policies are not provided?

[8] This run was far from completion when it was halted due to excessive accumulated wall-clock time.

We have started to investigate these questions in the context of a new CARE environment. The primary difference between the original environment and the new environment is that the *process* is no longer the basic unit of computation. While the new CARE system still supports the use of processes, it emphasizes the use of *contexts*: computations with less state than those of processes.

When a context is forced to suspend to await a value from a stream, it is aborted, and restarted from scratch later when a value is available. This behavior encourages fine-grained decomposition of problems, written in a functional style (individual methods are small, and consist of a binding phase, followed by an evaluation phase).

In addition, CARE now supports arbitrary prioritization of messages delivered to streams. As a result, it is no longer necessary to include in CAOS its complex and expensive scheduling strategy. Early indications are that the new CARE environment with a slightly modified CAOS environment performs between two and three orders of magnitude faster than the configuration described in this paper.

Acknowledgements

My thanks to Harold Brown, Bruce Delagi, and Reid Smith for reading and commenting on earlier drafts of this paper. Bruce Delagi, Sayuri Nishimura, Russell Nakano, and James Rice created and maintain the CARE environment. Harold Brown defined the behavior of the CAOS operators, ported ELINT from AGE to CAOS, and collected the results which appear in Section 6. Finally, I wish to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Lab for their excellent support of our computing environment.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structure and Algorithms.* Addison-Wesley, 1983.

[2] N. C. Aiello, C. Bock, H. P. Nii, and W. C. White. *Joy of AGE-ing.* Technical Report, Heuristic Programming Project, Stanford University, 1981.

[3] H. Brown, C. Tong, and G. Foyster. PALLADIO: An Exploratory Environment for Circuit Design. *IEEE Computer*, 16, December 1983.

[4] H. I. Cannon. *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming.* Technical Report, A.I. Lab, Massachusetts Institute of Technology, 1981.

[5] B. A. Delagi. *The CARE User Manual.* Technical Report, Knowledge Systems Laboratory, Stanford University, 1986. In preparation.

[6] Denelcor, Inc. *Heterogeneous Element Processor: Principles of Operation.* February 1981.

[7] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys*, 12:213–253, June 1980.

[8] R. P. Gabriel and J. McCarthy. Queue-Based Multiprocessing Lisp. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[9] R. H. Halstead, Jr. Implementation of MultiLisp: Lisp on a Multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[10] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[11] V. R. Lesser and D. D. Corkill. The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks. *The AI Magazine*, 15–33, Fall 1983.

[12] E. Y. Shapiro. *Lecture Notes on the Bagel: A Systolic Concurrent Prolog Machine.* Technical Memorandum TM-0031, Institute for New Generation Computer Technology, November 1983.

[13] R. G. Smith. *Structured Object Programming in Strobe.* Technical Report SYS-84-08, Schlumberger-Doll Research, March 1984.

[14] R. G. Smith and P. Friedland. *Unit Package User's Guide.* Technical Report HPP-80-28, Heuristic Programming Project, Stanford University, December 1980.

CAREL: A Visible Distributed Lisp

Byron Davies

Knowledge Systems Laboratory
Stanford University, Palo Alto, California

Corporate Computer Science Center
Texas Instruments, Dallas, Texas

Abstract

CAREL is a Lisp designed for interactive programming of a distributed-memory multiprocessor. CAREL insulates the user from the machine language of the multiprocessor architecture, but still makes it possible for the user to specify explicitly the assignment of tasks to processors in the multiprocessor network. CAREL has been implemented to run on a TI Explorer Lisp machine using Stanford's CARE multiprocessor simulator [Delagi 86].

CAREL is more than a language: real-time graphical displays provided by the CARE simulator make CAREL a novel graphical programming environment for distributed computing. CAREL enables the user to create programs interactively and then watch them run on a network of simulated processors. As a CAREL program executes, the CARE simulator graphically displays the activity of the processors and the transmission of data through the network. Using this capability, CAREL has demonstrated its utility as an educational tool for multiprocessor computing.

1. Context

CAREL was developed within the Advanced Architectures Project of the Stanford Knowledge Systems Laboratory. The goal of the Advanced Architectures Project is to make knowledge-based programs run much faster on multiple processors than on one processor. Knowledge-based programs place different demands on a computing system than do programs for numerical computation. Indeed, multiprocessor implementations of expert systems will undoubtedly require specialized software and hardware architectures for efficient execution. The Advanced Architectures Project is performing experiments to understand the potential concurrency in signal understanding systems, and is developing specialized architectures to exploit this concurrency.

The project is organized according to a number of abstraction layers, as shown in Figure 1-1. Much of the work of the project consists of designing and implementing languages to span the semantic gap between the applications layer and the hardware architecture.

The design and implementation of CAREL depends mainly on the hardware architecture level. At the hardware level, the project is concentrating on MIMD, large grain, locally-connected, distributed memory multiprocessors communicating via buffered messages. This class was chosen to match the needs of large-scale parallel symbolic computing with the constraints imposed by the desire for VLSI implementation and replication. Like the FAIM-1 project [Davis and Robison 85], we consider each processing node to have significant processing and communication capability as well as a reasonable amount of memory -- about as much as can be included on a single VLSI circuit (currently a fraction of a megabit, but several megabits within a few years). Each processor can support many processes. As both application and architecture are better understood, the detailed design of the hardware architecture will be modified to support the needs of the application.

Layer	Research Question
Applications	Where is the potential concurrency in signal understanding tasks?
Problem-solving frameworks	How do we maximize useful concurrency and minimize serialization in problem-solving architectures?
Knowledge representation and inference	How do we develop knowledge representations to maximize parallelism in inference and search?
Systems programming language	How can a general-purpose symbolic programming language support concurrency and help map multitask programs onto a distributed-memory multiprocessor?
Hardware architecture	What multiprocessor architecture best supports the concurrency in signal understanding tasks?

Figure 1-1: Multiple layers in implementing signal understanding expert systems on multiprocessor hardware

The hardware architecture level is implemented as a simulation running on a (uniprocessor) Lisp machine. The simulator, called CARE for "Concurrent ARray Emulator", carries out the operation of the architecture at a level sufficiently detailed to capture both instruction run times and communication overhead and latency. The CARE simulator has a programmable instrumentation facility which permits the user to attach "probes" to any object or collection of objects in the simulation, and to display the data and historical summaries on "instruments" on the Lisp machine screen. Indeed, the display of the processor grid itself is one such instrument.

2. Introduction

The CAREL (for CARE Lisp) language is a distributed-memory variant of QLAMBDA [Gabriel and McCarthy 84] and an extension of a Scheme subset [Abelson and Sussman 85]. CAREL supports futures (like Multilisp [Halstead 84]), truly parallel LET binding (like QLAMBDA), programmer or automatic specification of locality of computations (like Par-Alfl [Hudak and Smith 86] or Concurrent Prolog [Shapiro 84], and both static assignment of process to processor and dynamic spread of recursive computations through the network via remote function call. Despite the length of this list of capabilities, CAREL is perhaps best described as a high-level systems programming language for distributed-memory multiprocessor computing.

The CAREL environment provides both accessibility and visibility. CAREL is accessible because, being a Lisp, it is an interactive and interpreted language. The user may type in expressions directly and have them evaluated immediately, or load CAREL programs from files. If the multiprocessing features are ignored, using CAREL is just using Scheme. The multiprocessing extensions in CAREL are derived from those of QLAMBDA. For example, PARALLEL-LET is a simple extension of LET which computes the values for the LET-bindings concurrently, at locations specified by the programmer or determined automatically.

CAREL gains its visibility through the CARE simulator: CAREL programmers can watch their programs execute on a graphic display of the multiprocessor architecture. Figure 5-1 shows CARE and CAREL with a typical six-by-six grid of processors. A second window on the Lisp machine screen is used as the CAREL listener, where programs are entered. As a CAREL program runs, the simulator illuminates each active processor and each active communication link. The user may quickly gain an understanding of the processor usage and information flow in distributed CAREL programs. CARE instruments may also be used to gather instantaneous and historical data about the exection of CAREL programs.

The rest of the paper is divided into a discussion of the philosophy of CAREL, a description of the language CAREL, and some illustrated examples of CAREL in action on the CARE simulator.

3. Philosophy and Design

The CAREL language was developed with the following assumptions in mind:

1. CAREL (like Multilisp) was designed to augment a serial Lisp with "discretionary" concurrency: the programmer, rather than the compiler or the run-time support system, decides what parts of a program will be concurrent. CAREL provides parallelism through both lexical elaboration and explicit processes [Filman and Friedman 84].
2. Similarly, CAREL was designed to provide discretionary locality: the programmer also decides *where* concurrent routines will be run. A variety of abstract mechanisms are provided to express locality in terms of direction or distance or both.
3. CAREL generally implements *eager* evaluation: when a task is created, it is immediately started running, even if the result is not needed immediately. When the result is needed by a strict operator, the currently running task blocks until the result is available.
4. CAREL is designed to automatically manage the transfer of data, including structures, between processors. CAREL supports general methods to copy lists and structures from one processor to another, and specialized methods to copy programs and environments.
5. CAREL is designed to maintain "architectural fidelity": all communication of both data and executable code is explicitly handled by the simulator so that all costs of communication may be accounted for.
6. CAREL provides certain specialized "soft architectures", such as pipelines and teams, superimposed on the processor network.
7. Through CARE, CAREL graphically displays the runtime behavior of executing programs.
8. Finally, and unfortunately, CAREL ignores resource management, including the problem of garbage collecting data and processes on multiple processors. Resource management is a very important problem, but CAREL doesn't yet have a solution for it. CAREL currently depends on the memory management of the Lisp machine on which it runs in simulation.

4. The Language

This section presents a language description of CAREL and examples -- with graphics -- of its use. The functions and special forms of CAREL were selected roughly as the union of the capabilities of QLAMBDA (as extended for distributed memory) and Par-Alfl. There has been no attempt as yet to create a minimal but complete subset of CAREL.

On top of a Scheme subset, CAREL supports the following functions and special forms:

PARALLEL-LET: a special form for parallel evaluation of LET binding. Optionally, the programmer may specify the locations at which the values for binding are to be evaluated.

PARALLEL-LAMBDA: a special form to create asynchronously running closures. Optionally, the programmer may specify the location where the closure is to reside. The closure may also include state variables so that its behavior may vary over time.

PARALLEL: a parallel PROGN, evaluating the component forms concurrently.

PARALLEL-MAP: a parallel mapping function which applies a single function to multiple arguments at multiple locations, returning a list of the results.

MULTICAST-MAP: a parallel mapping function which evaluates the same form at multiple locations and gathers up the values returned in the order in which they are returned.

FUTURE: a special form specifying a form to be evaluated and the site at which the evaluation should take place. Returns a future encapsulating the value that will eventually be returned.

TOUCH/FORCE: a function to force a future to give up its value.

ON: evaluates a form at a specified location. Equivalent to (TOUCH (FUTURE ...)).

PIPELINE: a special-form to create a software pipeline of processes spread across multiple processors.

TEAM: a special form to create a team of processes spread across multiple processors. Each member of the team executes the same function. The team manager assigns a new task to the least loaded team member.

DEFINE-STRUCTURE: a simple version of DEFSTRUCT.

DEFINE-SERIALIZED-STRUCTURE: a serialized version of DEFINE-STRUCTURE. Each structure created incorporates a queue to serialize access to the structure.

CAREL augments standard Lisp datatypes with the following:

FUTURE-OBJECT: a datatype to encapsulate a value to be returned eventually after computing at a specified location

REMOTE-ADDRESS: a pointer to an object at a remote site

LOCATION: grid coordinates, neighbor/polar coordinates, or a keyword (:ANY, :ANY-NEIGHBOR, :ANY-OTHER)

STRUCTURE: a structure with named slots

SERIALIZED-STRUCTURE: a serialized structure with named slots

The following describes the syntax of CAREL's functions and special forms, and gives illustrated examples of their use. Certain expressions are used repeatedly in the paragraphs that follow, so their definitions appear first:

location-form is any form that evaluates to something that can be interpreted as a location in the CARE network.

body is an arbitrary list of forms.

PARALLEL-LET:

(PARALLEL-LET *parallel? bindings . body*)

parallel? is an arbitrary form, used to control the parallelism of the evaluation

bindings is a list of triples (*variable value-form location-form*)

As in QLAMBDA, *parallel?* is used to control whether the bindings should indeed be evaluated in parallel. If *parallel?* evaluates to () or #!FALSE, then the PARALLEL-LET is evaluated as an ordinary LET, with the bindings being evaluated in (an unspecified) sequence, and the body being evaluated in an environment including those bindings.

If *parallel?* evaluates to T or #!TRUE, then the location-forms are evaluated concurrently and the concurrent evaluation of the value-forms is begun. The variables are immediately bound to the future-objects corresponding to the values to be returned, and the evaluation of thc body is begun. The body may block temporarily on unfinished futures.

In all these cases, the value returned by the PARALLEL-LET is the (forced) value of the last form in the body.

PARALLEL-LAMBDA:

(PARALLEL-LAMBDA *parallel? args*
location-form state-bindings
. body)

Evaluating a PARALLEL-LAMBDA sets up a closure at a remote site specified by *location* and returns a function of the specified arguments. When this function is applied, the list of evaluated arguments is sent to the remote closure, the remote evaluation is initiated, and a future is immediately returned. The remote closure created by PARALLEL-LAMBDA contains some state variables, bound in *state-bindings*. A state variable is changed by applying the PARALLEL-LAMBDA function to the arguments (:SET *variable-name value*).

parallel? is used, as in PARALLEL-LET, to determine whether parallelism is actually employed.

PARALLEL:

(PARALLEL . *body*)

The PARALLEL special form initiates the concurrent evaluation of the forms in the *body*. Control returns from PARALLEL when all of the forms have been evaluated. The value returned by PARALLEL is undefined.

PARALLEL-MAP:

(PARALLEL-MAP *function-form arguments-form*
locations-form)

function-form evaluates to a function of one argument

arguments-form evaluates to a list, each member of which is to be used as an argument to the function

locations-form evaluates to a list of locations.

PARALLEL-MAP, like MAP, applies a function repeatedly to arguments drawn from a list and returns a list of results. Unlike MAP, PARALLEL-MAP performs the function applications concurrently and remotely, and returns a list of futures that will eventually evaluate to the results.

MULTICAST-MAP:

(MULTICAST-MAP *function-form locations-form*)

MULTICAST-MAP invokes a function of no arguments at each location in a list of locations. MULTICAST-MAP immediately returns a list of futures corresponding to the values that will eventually be returned. Since the function called takes no arguments, the values returned can be different only if they depend on the local state of the processor at the location of evaluation, as embodied in the "global" environment of that processor.

MULTICAST-MAP-NO-REPLY:

(MULTICAST-MAP-NO-REPLY *function-form locations-form*)

MULTICAST-MAP-NO-REPLY invokes a function of no arguments at each location in a list, but does not cause results to be returned. The value returned by MULTICAST-MAP-NO-REPLY is undefined.

PIPELINE:

(PIPELINE *stage1 ... stagen*)

where a stage is:

(*name args location-form state-variables . output-forms*)

For each stage expression, PIPELINE establishes a remote-closure at the specified location, and then links the remote closures so that the output of one stage becomes the input of the next stage. The linked closures form the working part of the pipeline. PIPELINE then returns a function which, when applied, passes its arguments on to the first stage of the pipeline and immediately returns a future which will eventually contain the result that comes out of the pipeline. To ensure that the results that comes out of the pipeline correspond one-for-one with the sets of arguments that went in, the future-object to hold the result is created atomically with the entry of the arguments into the pipeline and is passed along with the data through the pipeline.

TEAM:

(TEAM *args location-forms . body*)

The TEAM special form creates a set of closures, called a team, plus a single distinguished closure called the manager of the team. Each closure, or member of the team, is identical, except perhaps for its location within the processor network. When the manager of the team is applied to a list of arguments, the manager selects a member of the team and applies that member to the arguments, immediately returning a future which will eventually contain the value computed.

The purpose of the team is to spread a workload among a number of identical processes. Like the stages of a pipeline, the members of a team are created with a fixed functionality and are statically assigned to processors. Because of this, the overhead of invoking a team member is less than creating and invoking a new process.

DEFINE-STRUCTURE:

(DEFINE-STRUCTURE *structure-name . slot-names*)

DEFINE-STRUCTURE is a simple analog of the Common Lisp DEFSTRUCT. Evaluating a DEFINE-STRUCTURE special form creates:

1. a MAKE-*structure-name* function with required arguments corresponding to the *slot-names.* (MAKE-*structure-name . args*) creates an instance of *structure-name* with slot values specified by *args.*

2. *structure-name-slot-name* functions for each slot. These functions are used to access the slot values of a structure instance.

3. SET-*structure-name-slot-name* functions for each slot. These functions are used to set the slot values of a structure instance.

DEFINE-SERIALIZED-STRUCTURE:

(DEFINE-SERIALIZED-STRUCTURE *structure-name . slot-names*)

DEFINE-SERIALIZED-STRUCTURE is the same as DEFINE-STRUCTURE, except that access to the structure created is serialized. Only one process at a time may modify the structure.

5. Some Examples

PARALLEL-LET:

```
;;; This subroutine concurrently performs trivial
;;; computations at the four corner neighbors of a
;;; given location and collects the results.
;;;
(define (cycle-corners-1 where)
  (parallel-let t
     ((x1 (list 1 2) (neighbor 0 where))
      (x2 (list 3 4) (neighbor 2
                                (neighbor 1
                                          where)))
      (x3 (list 5 6) (neighbor 3 where))
      (x4 (list 7 8) (neighbor 5
                                (neighbor 4
                                          where))))
     (append x1 x2 x3 x4)))


;;; CYCLE calls the subroutine starting at the
;;; current processor
;;;
(define (cycle) (cycle-corners-1 *here*))
```

PARALLEL-MAP (see Figure 5-1):

```
;;; FOUR-CYCLE calls the CYCLE program at
;;; four different locations in the
;;; processor grid.
;;;
(define (four-cycle)
  (parallel-map cycle-corners-1
                '((2 5) (5 2) (2 2) (5 5))
                '((2 5) (5 2) (2 2) (5 5)))
```

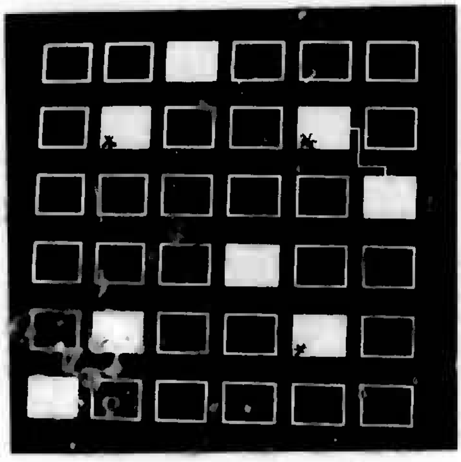


Figure 5-1: PARALLEL-MAP: Execution of the FOUR-CYCLE program. Active processors are displayed in inverse video. Active communications links are drawn as lines joining particular ports of the processor nodes. The processors annotated with asterisks are the cycle centers. Each processor is at a different point in the cycle.

PARALLEL-LAMBDA:

```
;;; This creates a process at some other node in
;;; the network, returning an object which, when
;;; applied as a function to two arguments,
;;; evaluates a linear expression on those
;;; arguments.
;;;
(define (linear-evaluator al bl)
  (parallel-lambda t (x y) ':any-other
                   ((a al) (b bl))
      (+ (* a x) (* b y))))
```

MULTICAST-MAP-NO-REPLY (see Figure 5-2):

```
;;; This activates the processor at each location
;;; in SITES, but does no worthwhile computation.
;;;
(define (activate-locations sites)
  (multicast-map-no-reply (lambda () *here*)
                          sites))
```

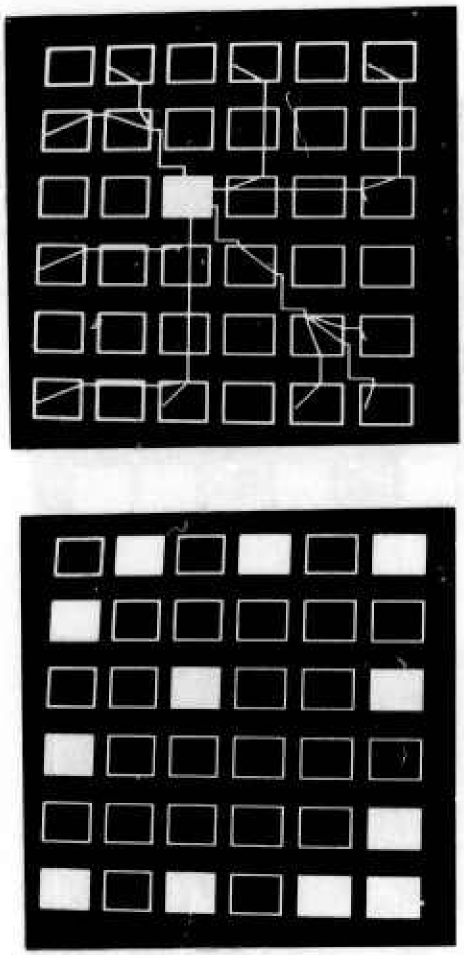


Figure 5-2: MULTICAST-MAP-NO-REPLY: Samples from the execution of the ACTIVATE-LOCATIONS program, showing how the multicast message is distributed and how the processors receiving the message are activated. Since no reply is required, the computation just dies out once the distributed programs are run.

MULTICAST-MAP (see Figure 5-3):

```
;;; This sends a message to each location
;;; in the list SITES, asking it to return
;;; its location.
;;;
(define (identify-yourself sites)
  (multicast-map (lambda () *here*) sites))
```

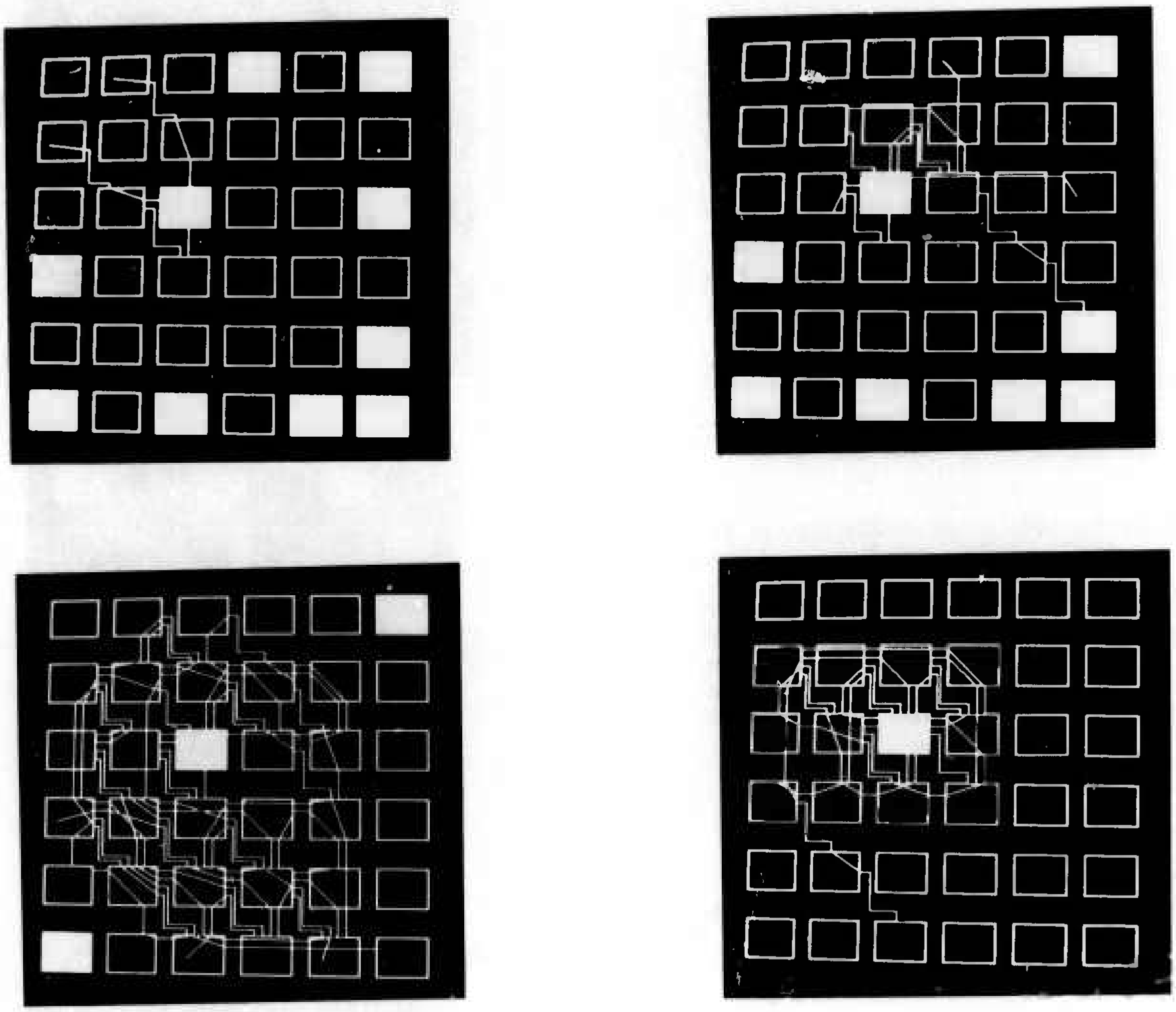


Figure 5-3: MULTICAST-MAP: Samples from the execution of the IDENTIFY-YOURSELF program. The multicast method is distributed as in Figure 5-2, but in this example the processors must send a value back to the requesting process. The network becomes congested as all the processors respond then gradually returns to rest as the messages reach their destination. The notion of a network "hot-spot" is clearly demonstrated.

PIPELINE:

```
;;; This sets up a pipeline across the bottom and
;;; up the right-hand side of the processor array.
;;; This trivial pipeline simply adds 1 to the
;;; input value at each stage and passes the result
;;; on to the next stage.  It also prints out the
;;; result at each stage, using a printing
;;; mechanism "outside" the simulation.
;;;

(define (make-test-pipeline)
  (pipeline
   (s1 (x) '(1 6) ((a 1)) (print (+ a
   (s2 (x) '(2 6) ((a 1)) (print (+ a
   (s3 (x) '(3 6) ((a 1)) (print (+ a
   (s4 (x) '(4 6) ((a 1)) (print (+ a
   (s5 (x) '(5 6) ((a 1)) (print (+ a x)))
   (s6 (x) '(6 6) ((a 1)) (print (+ a x)))
   (s7 (x) '(6 5) ((a 1)) (print (+ a x)))
   (s8 (x) '(6 4) ((a 1)) (print (+ a x)))
   (s9 (x) '(6 3) ((a 1)) (print (+ a x)))
   (s10 (x) '(6 2) ((a 1)) (print (+ a x)))
   (s11 (x) '(6 1) ((a 1)) (print (+ a x)))))
```

Part II, entitled *MONAD: A Hierarchical Model Paradigm for Reasoning by Analogy*, describes a methodology for analogical reasoning. The philosophy for the implementation in progress is described for the problem solving strategy

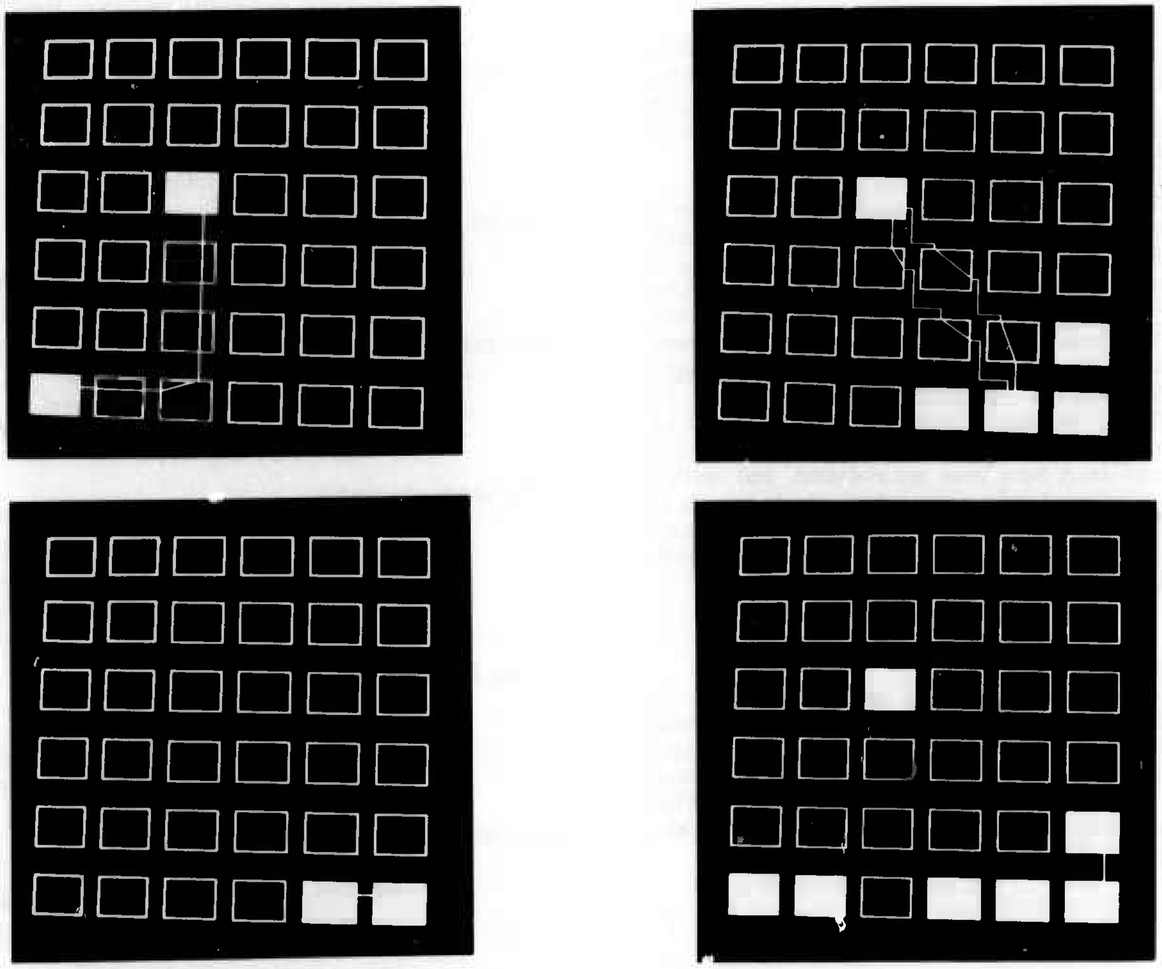


Figure 5-4: PIPELINE: Samples from the execution of programs constructing and using a CAREL software pipeline. The pipeline runs along the bottom and up the right side of the processor array. The pipeline is constructed in two passes. The first pass (a) establishes a process at each site and the second pass (b) links the processes together. The execution of the pipeline on a single argument (c) shows data flowing through the pipeline using only local communication. The last figure (d) shows that multiple data items may flow through the pipeline simultaneously, keeping multiple processors busy.

6. Implementation

CAREL is implemented by a "semicircular"[1] interpreter, implemented in Zetalisp and drawing heavily on the CARE simulator. Details of the representation will appear in a later paper [Davies 86]. These include the representation of CAREL datatypes in terms of Lisp and CARE primitives, the use of a "global" environment (full copies of which exist at each processor) and processor-local environments, and the interface to the CARE hardware simulator.

7. CAREL and Other Languages

CAREL was strongly influenced by three other languages: QLAMBDA [Gabriel and McCarthy 84], Par-Alfl [Hudak and Smith 86], and Actors [Agha 85]. QLAMBDA provided the idea of having two kinds of parallelism (which Filman and Friedman called parallelism by lexical elaboration and parallelism by cexplicit processes). CAREL addresses the question, "What would QLAMBDA look like on a distributed-memory multiprocessor?".

Par-Alfl provided the notion of a dynamic variable $SELF that a process could use, reflectively, to determine where it was executing. The part of CAREL that implements parallelism by lexical elaboration is very similar to Par-Alfl. CAREL adds the ability to deal with processes as first class objects.

CAREL differs from Actors in its emphasis on discretionary parallelism and in its reliance on the programmer to manage process resource allocation. These are consequences of CAREL's design as simple extension of an existing serial Lisp. CAREL's primitives for concurrency and locality are powerful enough to implement a wide variety of interesting programs, but still provide less concurrency, less capability for managing synchroniation, and less theoretical elegance than Actors. For example, CAREL enforces synchronization at the inputs and outputs of a function or closure: when APPLY is invoked, all the arguments must have been pre-evaluated, and multiple outputs are considered to be generated in a single list. In the Actor language SAL described by Agha, the inputs to an Actor may arrive at any time and in any order and outputs likewise may be generated asynchronously.

8. Acknowledgements

Implementation of CAREL was made possible by the existence of the CARE simulator, as implemented by Bruce Delagi and augmented by Eric Schoen. The author further wishes to acknowledge the intellectual support of the Stanford Advanced Architectures Project. Contributors to PARSYM, the netwide mailing list for parallel symbolic computing, have provided fruitful stimulation.

[1] Semicircular, not metacircular, because it is implemented in Lisp, but not in CAREL itself.

References

[Abelson and Sussman 85]
Harold Abelson and Gerald Jay Sussman with Julie Sussman.
Structure and Interpretation of Computer Programs.
MIT Press, Cambridge, Massachusetts, 1985.

[Agha 85] Gul A. Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
Technical Report, MIT AI Laboratory, March, 1985.

[Davies 86] Byron Davies.
CAREL: Implementation of a Distributed Scheme.
Technical Report In preparation, Stanford Knowledge Systems Laboratory, 1986.

[Davis and Robison 85]
A. L. Davis and S. V. Robison.
The Architecture of the FAIM-1 Symbolic Multiprocessing System.
In *Proceedings of IJCAI-85.* 1985.

[Delagi 86] Bruce Delagi.
CARE User's Manual
Heuristic Programming Project, Stanford University, Stanford, Ca. 94305, 1986.

[Filman and Friedman 84]
R. E. Filman and D. P. Friedman.
Coordinated Computing: Tools and Techniques for Distributed Software.
McGraw-Hill, New York, 1984.

[Gabriel and McCarthy 84]
Richard P. Gabriel and John McCarthy.
Queue-based multiprocessing Lisp.
In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984.* 1984.

[Halstead 84] Robert H. Halstead.
Implementation of Multilisp: Lisp on a Multiprocessor.
In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984.* ACM, 1984.

[Hudak and Smith 86]
P. Hudak and L. Smith.
Para-functional programming: A paradigm for programming multiprocessor systems.
In *Proceedings of ACM Symposium on Principles of Programming Languages, January 1986.* ACM, 1986.

[Shapiro 84] E. Shapiro.
Systolic programming: A paradigm of parallel processing.
In *Proceedings of the International Conference on Fifth Generation Computer Systems.* 1984.

MULTI-SYSTEM REPORT INTEGRATION USING BLACKBOARDS

John R. Delaney

Knowledge Systems Laboratory
Stanford University
701 Welch Road, Building C
Palo Alto, CA 94303

ABSTRACT

Blackboards are an AI problem solving methodology. A blackboard system consists of a structured data base (the blackboard) holding input and derived inferences and a collection of procedures for deriving inferences (knowledge sources). Each knowledge source is specialized to operate on some portion of the blackboard. The knowledge sources are invoked opportunistically as the information on the blackboard increases.

The best known applications of the blackboard methodology have been in speech understanding and passive sonar data interpretation. The inputs in these cases were a single form of raw sensor data. But the methodology is also well suited to integrating multiple streams of fully reduced and qualitatively different data such as active radar track reports, passive electronic intelligence reports, and human intelligence reports about enemy intentions.

This paper sketches the nature of the blackboard problem solving methodology with an emphasis on those features suiting it to such applications. The sketch is illustrated with examples from a relatively simple multi-system report integration problem. Relevant applications currently under development at Stanford's Knowledge Systems Laboratory are also described.

INTRODUCTION

"Multi-System Report Integration" is an odd phrase. An alternative would have been "Sensor Data Fusion". But that phrase often implies a less reduced form of information to integrate than is intended here. The reporting systems in this paper are presumed to reduce the data they sense as fully as is practical with only that data available. The degree of processing can vary from s, stem to system. For a radar tracking system, the reports would be samples of on-going tracks integrating all measurements up to the present. For an ELINT system dealing with intermittent emissions, the reports might be just current emitter and bearing characteristics. And for a human intelligence gathering system, the reports might be informed guesses about near-term enemy intentions.

"Sensor Data Fusion" also usually implies that the information to be integrated appears at comparable time intervals or is static. But the reporting systems in this paper are presumed to provide reduced data over a wide range of time intervals. The radar, ELINT, and "humint" systems mentioned above could produce reports at very different intervals with very different degrees of regularity. Assuming that some reports are locally of comparable frequency while others are locally static information is Procrustean.

"Blackboards" refers to a particular AI problem solving methodology. The best known applications of the blackboard methodology are HEARSAY-II, a speech understanding system (2), and the HASP/SIAP sonar data interpretation system (4,5). These applications effectively processed regular streams of data from a single sensor, treating any other information as locally static. But the blackboard methodology is more generally applicable. In particular, it provides a convenient framework for integrating maximally reduced information from multiple sources with different temporal characteristics. Just what is needed for multi-system report integration.

In the first section below, the fundamental features of blackboard systems are described abstractly. A consistent set of examples are used in the following section to clarify those features in context of multi-system report integration. The next section reviews those aspects of the blackboard methodology particularly suited to multi-system report integration. The last section briefly describes work in progress at Stanford's Knowledge System Laboratory on two more ambitious examples. It also explains how that work is embedded in a larger effort.

NATURE OF BLACKBOARDS

The blackboard problem solving methodology originated approximately 10 years ago and has been evolving ever since. The hallmarks of a blackboard system are:

- A global data store holding input data and hypotheses about the solution of the problem derived from that data. Related information is kept together. This data store is known as the blackboard.
- A collection of procedures for deriving hypotheses about the solution of the problem from the input data and/or from other hypotheses. Each procedure is specialized to operate on a particular portion of the blackboard. These procedures are known as knowledge sources.
- A mechanism for invoking a knowledge source on relevant parts of the blackboard. A knowledge source is invoked on a particular piece of the blackboard when the invocation would incrementally advance the solution of the problem. This mechanism is known as the control structure.

Each of these hallmarks is described abstractly in the remainder of this section with simple examples appearing in the next.

The blackboard holds the state of the problem solving system as the solution evolves. In conventional terms, the dimensionality of the state varies with time. The elements may be discretely or continuously valued. And the

----

This work was supported by the Defense Advanced Research Projects Agency, the NASA-Ames Research Center, Boeing Computer Services, and the National Institutes of Health.

elements change values at discrete times. But such observations miss the most significant feature of the blackboard. It structures the information it holds.

Closely related input data or hypotheses are collected together in the form of blackboard nodes having certain attributes and values for those attributes. Related nodes form blackboard levels. All the nodes in a given level having the same attributes but (potentially) different attribute values. Levels can in turn form hierarchies of analysis or abstraction, usually with input data nodes at the base of each hierarchy. The most common nodal attributes are links between nodes on different levels. Such links connect hypotheses to input data or other hypotheses which support them. They can be links up and down levels within a hierarchy or they can be across hierarchies.

Knowledge sources transform the state of the problem solving system by adding nodes to the blackboard, by removing them, or by modifying their attribute values. Knowledge sources are effectively parametric procedures for transforming the state. A knowledge source could be invoked on any node at a given level or a tuple of nodes at one or more levels. It operates only on the node(s) upon which it is invoked plus those nodes linked directly or indirectly to them. Knowledge sources are also effectively typed procedures; a knowledge source can be invoked only on a node of a particular level or on a tuple of nodes, each of a particular level. This feature of knowledge sources provides them with a degree of modularity. In particular, knowledge sources do not interact directly.

The procedure carried out by a knowledge source expresses knowledge of how to advance the problem solution. It is expressed in the creation, modification, and/or elimination of particular sorts of hypotheses in the form of nodes of particular levels. In this sense, a knowledge source is a specialist in the solution of some part of the overall problem. The details of the procedure can be expressed in any form. A typical form is a set of production rules and a policy for using them.

Each production rule specifies a logical condition on the attribute values of the node(s) upon which the knowledge source is invoked and an action to be carried out if that condition is true. Both the condition and action can be compound. The value of a compound condition is TRUE if the values of all its component conditions have TRUE values. A compound action is simply a sequence of individual nodal creations, deletions, or modifications. Evaluating a logical condition or modifying a node may require the application of complex numeric functions to attribute values. In this way, production rules mix symbolic and numeric computations.

Different policies for using a set of production rules allow at most one action to occur, or multiple actions but never the same one twice, or the same one repeatedly. In the first case, the rules are scanned in order of definition with the scan terminating immediately if a rule's action is carried out. In the second case, the logical conditions of the rules are all tested before any actions take place. Then any actions are carried out in parallel. The third case is simply the second case repeated until no logical condition is TRUE. While this style of programming many seem bizarre at first, it has proved quite successful in past and existing blackboard systems.

A knowledge source describes the procedure by which it changes the blackboard when invoked. It also describes when it is invocable. The most general form of this description is a (possibly compound) logical condition on attribute values of the node(s) upon which it could be invoked. In this manner, a knowledge source resembles a production rule. The condition is parametric in the same sense that each knowledge source is parametric. As a result, the same knowledge source may be invocable on several nodes or tuples of nodes simultaneously. Each such combination of a knowledge source and a node or tuple of nodes is called a potential invocation. At any time, there are typically many potential invocations. The control structure determines the set of potential invocations, picks one, and causes it to be carried out.

Many blackboard systems do not use the most general form to describe when a knowledge source is invocable. They use events and logical combinations thereof. An event is a summary of a blackboard change. A knowledge source posts the appropriate event or events when it completes. A pointer to the affected node is associated with each event. These systems may also use events for an additional purpose as explained below.

The control structure is intended to operate in an opportunistic manner analogous to the manner in which people solve jigsaw puzzles. Initially, the puzzle solver scans for pieces with singular small-scale characteristics. If two such pieces have similar characteristics, they are tested for fit. Gradually, clusters of pieces accrete as the puzzle solver continues to scan through the unused pieces. Once the clusters become sufficiently large, scanning the pieces is replaced by searches for specific pieces to extend a cluster. But pieces plausibly belonging another cluster are tested for fit there if they are chanced upon during a search. Eventually, large clusters are recognized as connected on the basis of large scale characteristics and are jointed. If progress while searching for specific pieces bogs down, the puzzle solver reverts to scanning for pieces with similar characteristics for a time. It choses that activity which, at the moment, seems likely to make the best contribution to the overall solution of the problem.

A variety of techniques are used by the control structures of different blackboard systems to decide which potential invocation would, if carried out, make the best contribution to the overall solution. The topic is being actively researched. One system has an additional blackboard for handling hypotheses about the best choice (3) and another allows all potential invocations to be carried out in parallel (6).

Several blackboard systems use events in their control structures. After a particular event or sequence of events, particular knowledge sources are preferred to others. And they are prefered for invocation on the affected node or nodes. These same systems also use events to describe when a knowledge source is invocable. So the control structures of these systems need only attend to events and not to the blackboard nodes themselves.

Some of these blackboard systems also use expectations in their control structures. Expectations are posted by knowledge sources just as events are posted. Generally speaking, they are instructions to invoke a particular knowledge source on a particular node or nodes when, if ever, a certain event or pattern of events occurs involving the node(s). Expectations can also be negative. Such expectations cause a particular knowledge source to be invoked if a certain event or pattern of events does not occur within a specified time interval.

BLACKBOARDS ILLUSTRATED

Consider the problem of producing a situation map of aircraft flying over an area of interest. The situation map is based on track reports from an air surveillance radar tracking system, emitter/bearing reports from an ELINT system sensing airborne radar emissions, and warnings from a human intelligence system. The warnings are that particular aircraft or groups of aircraft may soon enter the area of interest with particular objectives in mind. The situation map should identify the type of each aircraft as well as its current position and velocity. The radar track reports are regular for aircraft in the area of interest. The ELINT reports are intermittent by comparison. There are no reports unless an emitter is on. And the detection range of an active emitter can depend on its type and, in some cases, on the aircraft's aspect. ELINT reports are also less

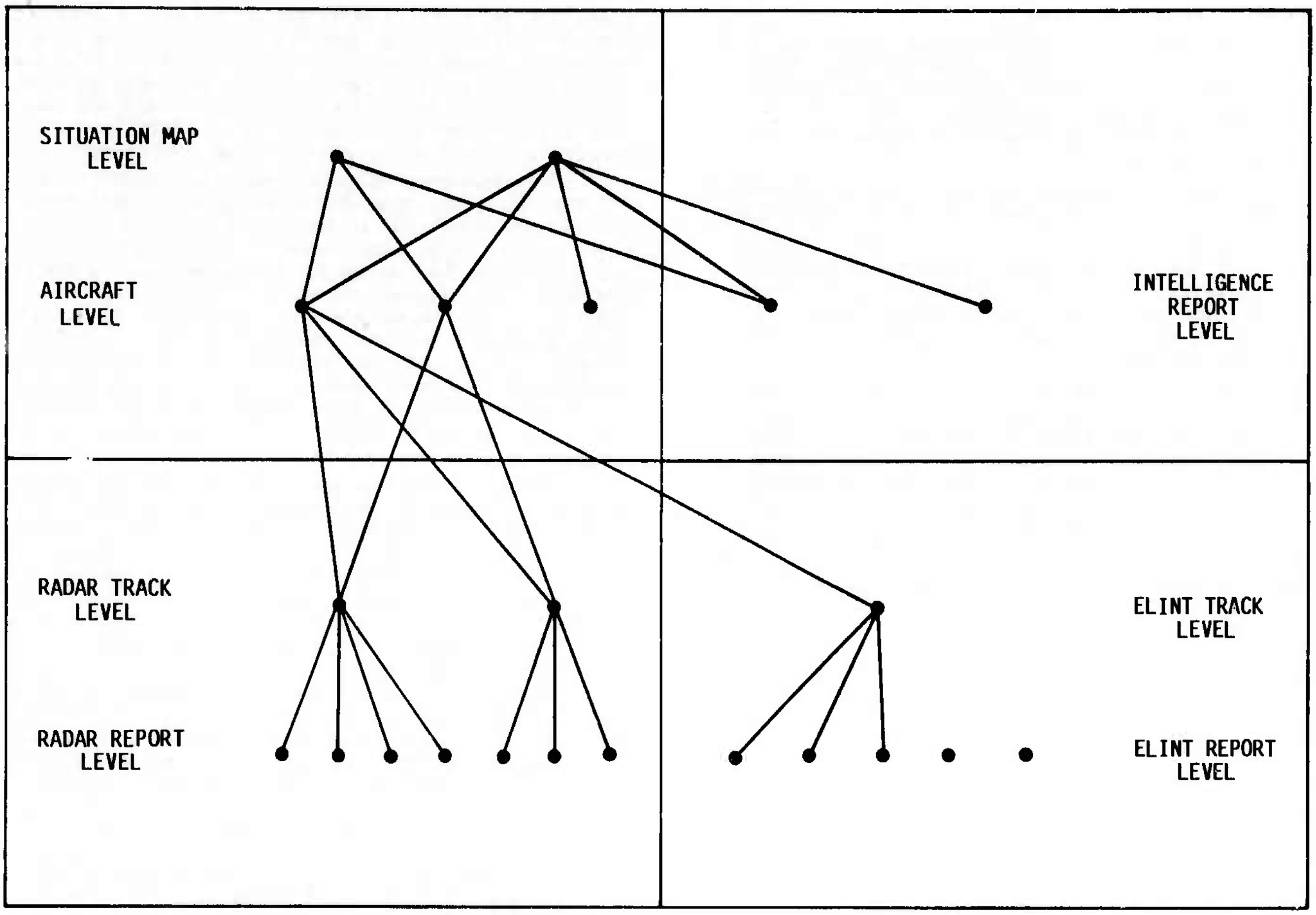


Figure - A Blackboard with 7 levels of nodes in 4 hierarchies

accurate geometrically than radar reports. Intelligence reports are generally less frequent than the ELINT reports, but can be updated rapidly on occasion.

Figure 1 illustrates a possible blackboard configuration during the course of solving this problem. There are seven levels on the blackboard, a typical number. The situation map and aircraft levels form one hierarchy of levels. Nodes on these two levels hierarchically express alternative hypotheses about the map of aircraft in the area of interest. Two situation map hypotheses exist in this case, both including the same two hypothetical aircraft and one including a hypothetical third aircraft as shown by links between the corresponding nodes in the figure. One attribute of a situation map node is thus a set of component aircraft nodes. Hypothesis credibility is also a situation map node attribute. *A posteriori* probability would be a reasonable credibility measure. The value of that attribute is a function of the credibilities of the supporting aircraft hypotheses.

The intelligence report level is treated as a separate, degenerate hierarchy in the figure. The figure shows two intelligence report nodes. Links indicate that one of these reports supports both situation map hypotheses while the second report supports only one of them. The credibility attribute value of each situation map node is also a function of the credibility of each intelligence report node linked to it.

The radar track and radar report levels form another hierarchy. So do the ELINT track and ELINT report levels. A sequence of report nodes is linked to a corresponding track node to represent the hypothesis that they were all caused by the same object, aircraft or emitter. Similarly, the links between the aircraft nodes and both kinds of track nodes represent the hypothesis that the tracks are all of the same aircraft. The credibility of an aircraft hypothesis is a function of the credibilities of the two kinds of track hypotheses supporting it.

It will prove useful later to have explicit definitions of certain attributes of radar report and radar track nodes. We do so in pseudo-computerese as follows:

```
Level: radar-report
Attributes: report-time
            track-identifier
            state-estimate
                North position
                East position
                North velocity
                East velocity
            state-covariance
            ...
            associated-tracks

Level: radar-track
Attributes: last-associated-report
            report-history
            track-credibility
```

The names of the attributes suggest their intended meanings. But attributes are given pragmatic meaning by the way the attributes are manipulated by knowledge sources. They are analogous to the elements of a state vector in this sense.

Knowledge sources embody knowledge about how to solve a problem. Consider the following fragment of knowledge about radar tracking:

> A sequence of radar reports caused by a particular aircraft usually have the same track identifier. An exception may occur if two aircraft approach closely at some time, in which case the track identifiers are swapped at roughly the time of closest approach.

It can be converted into the following fragments of knowledge about collecting radar reports into radar tracks:

> Given a radar report node that is not associated with any radar track node and given a radar track node, if the radar report node's track identifier is the same as that of the radar track node's last associated radar report node, then associate them.
>
> Given two radar track nodes, if their histories of associated radar report nodes indicate a close approach, then create two new radar track nodes with histories composed by splitting the original track nodes' histories at the time of closest approach and rejoining them with the track identifiers swapped after that time.

A knowledge source based on the first of these fragments is expressed in pseudo-computerese as follows:

```
Applies-to:
        a-radar-track , a-radar-report

Invocation-condition:
        associated-tracks of a-radar-report =
                empty-set

Use-policy:
        all-true-once

Production-rule 1:
Condition:
        track-identifier of last-associated-report
                        of a-radar-track =
                track-identifier of a-radar-report

Action:
        last-associated-report of a-radar-track
                :=  link to a-radar-report ;
        report-history of a-radar-track
                :== link to a-radar-report ;
        associated-tracks of a-track-report
                := link to a-radar-track
```

Here ":=" symbolizes assignment, ":==" signifies addition to a set, and ";" sequences simple actions in a compound one.

The knowledge source is quite simple, with just one production rule. That is atypical. Knowledge sources using production rules typically employ between ten and thirty production rules. A knowledge source realizing the second fragment would be more complex. It would include one or more production rules used to determine whether a possible close approach occurred and when.

The details of any particular control structure are complex. And the motivation for that complexity is not apparent in an example involving just one or two knowledge sources and a few nodes. So no attempt is made to include control structure details in this illustration. A sketch of the blackboard changes one would prefer under particular circumstances provides a better feel for the control structure's gross behavior. It also illustrates how the different components of a blackboard system can come together to solve a problem.

Assume that no reports have been received of any sort by the blackboard system. Then one situation map node exists with no links to aircraft nodes. This represents the hypothesis that no aircraft are in the area of interest. Then an intelligence report is posted on the blackboard. It warns that some number of aircraft of a particular type or types are expected to enter the area during a specified time interval across a specified portion of the area's boundary. Aircraft nodes are then created with the appropriate types, all linked to a new situation map node. The credibility of this new situation map node is the same as that of the intelligence report. The credibility of the old situation map node is appropriately adjusted downward.

The radar track attribute of each new aircraft node is not filled in at this point. There are no radar track nodes yet. But an expectation is established that later examines newly created radar track nodes. If one is created in the appropriate time interval and the appropriate place, a link to that radar track becomes the value of the associated track attribute. If the expectation goes unsatisfied, the aircraft node is deleted and the credibility of each associated situation map is reduced. Whenever the credibility of a situation map node slips below a certain level, that node is also deleted. Any aircraft nodes linked only to that situation map node are also deleted. The credibilities of all remaining situation maps are then re-normalized.

Receipt of the first few radar track reports causes them to be posted on the blackboard, but no more. Only when three report nodes having the same track identifier appear on the blackboard is a radar track node created to represent the hypothesis that they are from a single aircraft. In this manner, the creation of false radar track nodes based on radar false alarms is largely avoided. The resulting node may then be linked to an existing aircraft node by the aforementioned expectation.

Failing that, a new aircraft node is created to which the new radar track node is linked. Then the cross-product is formed of the old situation map hypotheses and the pair of hypotheses that the radar track was or was not caused by an aircraft. One new situation map node is created corresponding to each existing one. The new situation map nodes are copies of the old nodes, each with a link to this aircraft node added. Some portion of the credibility of each old situation map hypothesis must also be transferred to the corresponding new hypothesis. At this point, the knowledge source which removes insufficiently credible situation map nodes is again applied to reduce the number of situation map hypotheses maintained.

The accretion of ELINT reports into ELINT tracks is similar to that of radar reports into radar tracks. But the creation an of ELINT track does not satisfy any expectations or trigger the creation of an aircraft node. Rather it triggers a search for aircraft nodes of a type which could produce the sensed emission and which has a history of estimated positions (implicit in the radar tracks' report history) consistent with the ELINT track's history of bearings (similarly implicit). The ELINT track node is linked with any and all such aircraft nodes. The credibility of any such aircraft nodes is increased appropriately to reflect evidence that the hypothesis it represents is correct. Such a credibility increase must also be propagated up to the situation map nodes. Creation of a new aircraft node triggers a similar search for supporting ELINT tracks.

Prioritization among the knowledge sources carrying out the aforementioned actions can be relatively simple. The arrival of a new input datum should trigger a locus of activity on the blackboard which propagates up the network of levels, with pauses to spread down along different hierarchies as appropriate. All of the activity directly triggered by one datum should be completed before the next input datum is posted. To keep the amount of inter-input processing reasonable, the diversity of hypotheses created in the normal course of processing must be limited. Thus as additional radar reports arrive, the posted nodes are simply associated with radar tracks on the basis of track identifiers as in the above knowledge source example. It would be possible to create track nodes expressing all possible hypothetical combination of track reports without regard to track identifiers. But the processing required to create, qualify, and eventually delete most of these nodes would be wasteful given the number of possible combinations.

But when should the control structure invoke the knowledge source which tests for a close approach of two aircraft and creates new track nodes to reflect a possible confusion of track identifiers? One answer would be after the completion of every invocation of the knowledge source associating a new radar report with an existing radar track. But that would mean frequent invocations, usually producing no change. An alternative is to invcke that knowledge source only when some other, less frequent, occurrence suggests the possibility of a close approach by two aircraft and consequent track identifier confusion be considered.

In the scheme described above, ELINT tracks are associated with an aircraft if they are consistent with the aircraft's hypothesized type and with the radar track. If the tracks are geometrically consistent but the nature of the tracked emission is inconsistent with the aircraft type, one possibility is that the aircraft hypothesis was wrong with regard to type and should be discarded or modified. But another possibility is that the radar track history actually corresponds to two different aircraft at two different times due to a track identifier confusion during a close approach. If ELINT tracks are already linked with the aircraft node as support for the hypotheses, the possibility of a close approach should be investigated first.

The above sketch does not reflect the only manner in which the example problem might be solved. It reflects various options for incrementally advancing the problem solution. Choosing which option to use in a particular situation can require subtlety if one wishes to be computationally efficient. Not illustrated are the additional subtleties of advising the control structure how to achieve that sequencing. Experience is required to make such choices wisely. Experience is also important in the construction of knowledge sources, the choice of blackboard levels, and the selection of nodal attributes. Simple examples can only suggest the subtleties involved.

SUITABILITY OF BLACKBOARDS

The above sketch of possible blackboard changes illustrates a major reason why the blackboard problem solving methodology is suitable for multi-system report integration. The ordering of changes adapts appropriately to the arrival of very different sorts of input data in different orders.

If any intelligence report involving a particular aircraft arrives after radar track reports corresponding to it, the hypothesis that it exists will still have been formed. The credibility of the situation map hypotheses supported by that aircraft hypothesis will be increased once the intelligence report is incorporated into the support for those situation map hypotheses. ELINT reports are not discarded immediately if they do not confirm an existing aircraft hypothesis. They are saved for possible confirmation in the future. And exceptional occurrences need be considered only when evidence suggests they occur. The close approach of two aircraft leading to track identifier confusion being the case in point.

This adaptability in the operation of a blackboard system is a consequence of the control structure's opportunistic invocation of knowledge sources, the knowledge sources' modularity of forming or altering hypotheses, and the blackboard's structured composition of hypotheses. Any knowledge source can be invoked after any other completes, depending on the state of the blackboard, i.e., of the problem's solution, at that point in time.

The blackboard methodology also provides a means for managing the complexity of large multi-system report integration problems. Knowledge sources are modular in their applicability to all nodes of a given level, or tuples of given levels, but only to those nodes. Modularity is also achieved by expressing a partial problem solution as hypotheses supported by a hierarchy, or a set of linked hierarchies, of sub-hypotheses ultimately based on input data. Solution to individual parts of a particular multi-system report integration problem can be conceptualized and implemented without dwelling on the details of how the results of solving one part are used in the solutions of other parts.

Standard algorithms can be used where appropriate to solving part of the problem. But special pre- or post-processing may be required. Such pragmatic features of a standard algorithm's use in a particular context can be isolated from the algorithm itself by encapsulating them in separate knowledge sources. Explicitly separating formal and heuristic aspects of a problem's solution can highlight the heuristic aspects. It illuminates the assumptions, explicit or implicit, upon which they are based. Modifying the heuristic aspects without compromising the formal aspects also becomes easier.

WORK IN PROGRESS

The Heuristic Programming Project Group of Stanford's Knowledge System I aboratory is trying to

- realize a new generation of software architectures using parallel computation to speed up AI applications and
- specify multiprocessor system architectures for carrying out those computations efficiently.

Among the issues being investigated are

- recognition of opportunities for parallelism in the solution to a problem and
- expression of that potential parallelism in a problem solving framework that can exploit it.

In particular, this effort is focusing on signal understanding problems and blackboard-like frameworks.

Blackboard systems appear to be intrinsically parallel. At any time, there can be many potential invocations of knowledge sources. Those involving different nodes seem eligible for parallel execution. Within knowledge sources, production rule conditions could be evaluated in parallel. And some production rule actions could be safely executed in parallel. Currently two different blackboard systems are under development, each investigating a different approach to expressing opportunities for parallel computation or requirements for serial computation. Applications of these experimental systems used in evaluating their effectiveness.

The focus on signal understanding problems follows in large part from the focus on blackboard systems. The two mate well. But signal understanding problems are important in their own right. When signal understanding is defined broadly, it includes sensor data fusion and multi-system report integration. That class of problems is large and of considerable interest to the military.

Two signal understanding problems have been investigated so far as part of the current project. They are referred to as the TRICERO/ELINT and AIRTRAC problems. While generally similar, each problem is expected to push the research into recognizing opportunities for, and expressing, parallel computation in different directions.

In the TRICERO/ELINT problem, streams of ELINT emitter/bearing measurements must be combined to estimate the flight paths and operating modes of non-cooperating aircraft. The problem is named after ESL's TRICERO blackboard system for solving a problem of which this one is just a component. The knowledge of how to solve the TRICERO/ELINT problem has already been worked out, albeit without attention to opportunities for parallel computation. So work on this problem is further along.

The AIRTRAC problem is recognizing aircraft flying across a national border and heading for particular airfields used by smugglers. The smugglers' aircraft must be picked out of the normal air traffic across that border. To solve the problem, aircraft destinations must be recognized, not just flight paths and types. Streams of radar reports from multiple radar systems are available. But the low altitude coverage of those radars is assumed to be limited and the smugglers are assumed to know the coverage limits. So smugglers can try to avoid detection. They can also maneuver their aircraft evasively to disrupt tracking. Such behavior is a sure sign of a smuggler's aircraft, but makes the recognition of a destination difficult.

To complicate the AIRTRAC problem further, distributed aeroacoustic tracking systems using modest batteries of acoustic sensor arrays(1,7) are placed across large holes in radar coverage. These systems provide tracking reports within their limited coverage. Because such systems are passive and readily moved, the smugglers are assumed to be unaware of their coverage and so unable to avoid detection by these systems. These systems also use acoustic signature information to provide aircraft class estimates along with tracking reports.

Initial solutions to both problems should be completed in both experimental blackboard systems by the end of the year. Moreover, each solution should have been applied to several problem scenarios on realistic simulated multiprocessors. These experiments will determine how much parallelism was realized and may suggest alternative ways of realizing more parallelism.

REFERENCES

(1) J.R. Delaney and R.R. Tenney, "Broadcast Communication Policies for Distributed Aeroacoustic Tracking", Proceedings of the 8th MIT/ONR Workshop on C^3 Systems, Cambridge, MA, July, 1985, pp.195-199.

(2) L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy, "The HEARSAY-II Speech Understanding System: Integrating Knowledge To Resolve Uncertainty", Computing Surveys, v. 12, December 1980, pp. 213-253. Also reprinted in (8).

(3) B.Hayes-Roth, "A Blackboard Architecture for Control", Artificial Intelligence, vol. 26, no. 3, July 1985, pp. 251-321.

(4) H.P. Nii and E.A. Feigenbaum, "Rule-Based Understanding of Signals", in D.A. Waterman and F. Hayes-Roth, Pattern-Directed Inference Systems, Academic Press, San Francisco, 1978, pp. 483-501.

(5) H.P. Nii, E.A. Feigenbaum, J.J. Anton, and A.J. Rockmore, "Signal-to-Symbol Transformation: HASP/SIAP Case Study", AI Magazine, vol. 3, no. 2, Spring 1982, pp. 23-35.

(6) J. Rice, "POLIGON: A System for Parallel Problem Solving", Knowledge Systems Laboratory Technical Report 86-19, Stanford University, 1986

(7) R.R. Tenney and J.R. Delaney, "A Distributed Aeroacoustic Tracking Algorithm", Proceedings of the 1984 American Control Conference, San Diego, CA, June 1984, pp. 1440-1450.

(8) B.L. Webber and N.J. Nilsson (eds.), Readings in Artificial Intelligence, Tioga Press Company, Palo Alto, 1981.

AIDE : A Distributed Environment for Design and Simulation

*** Working Paper ***

Nakul P. Saraiya
Knowledge Systems Laboratory
Department of Computer Science
Stanford University
April, 1986

Abstract

AIDE is an environment that provides facilities for the design and simulation of systems, specifically multiprocessor computer systems. In addition, AIDE has facilities to do distributed simulation of such a system using a network of hosts. We are currently evaluating the performance of the distributed simulation algorithm on a network of workstations for a simulated multiprocessor system.

1. Introduction

A design system is expected to provide a *framework* for a designer to adequately implement representations of certain interesting physical or abstract entities that perform some function. In doing so, it must provide a suitably precise formalism and an integrated set of tools allowing the designer to conveniently specify, modify and evaluate such representations [1, 9]. AIDE[2] is an attempt to provide such a framework.

AIDE evolved in the context of the Advanced Architectures for Expert Systems project of the Heuristic Programming Project. The project requires simulating a large distributed-memory message-passing MIMD architecture (CARE [5]) running several additional software layers (for example, CAOS, POLIGON, and ELINT). This led naturally to investigating the utility of distributed simulation both as a means of reducing simulation turnaround time and in ensuring that the simulated machine was being programmed fairly (without making use of the *real* shared memory available on the host machine). Furthermore, implementing the distributed simulation algorithm was in itself a useful exercise in symbolic programming of a multiprocessor system, addressing some of the same concerns as an application written for CARE.

This document describes the essential aspects of the AIDE system. The first part of the document concerns design representation and capture, and the second part deals with design validation, specifically sequential and distributed simulation. More detailed documentation for the system is contained in the user's manual [7].

2. Design Capture

Design capture denotes the process of specifying a representation of an abstract entity to a design system. Below we discuss the formalism and supporting tools provided by AIDE to facilitate this process.

2.1. Representation

Every real-world or abstract entity may be characterized by its **structure** and its **behavior**. A structural view of an entity is any organizational view of the entity that decomposes it into (functionally or otherwise) semi-independent components. The behavior of an entity is a conceptual formalization of the way certain interesting properties of the entity change over time; different formulations (possibly emphasizing different concerns) lead to different specifications of behavior. A design or model in AIDE is exactly the totality of its specified structure and behavior.

The process of design is "partially-structured" [1]; designers often work both top-down and bottom-up. AIDE provides a structural formalism that supports this notion.

2.1.1. Hierarchical Partitioning

The well-known technique of hierarchical decomposition is one of the ways in which a designer makes the process of designing a complex system more tractable. For example, PALLADIO [1] viewed the process of circuit design as the incremental refinement of a functional description of the circuit into its physical realization. Here the basic design refinement step was partitioning the circuit at some abstract structural level into constituent components specified at either the same level or a less abstract level.

AIDE supports hierarchical partitioning directly and simply by allowing the designer to define a component[3] structurally in terms of arbitrary (perhaps incompletely specified) subcomponents.

2.1.2. Design Libraries

Complementing hierarchical partitioning is the use of *prototypes* to build on previous work [4, 1]. This allows the designer to rapidly create new designs by modifying existing components or by applying new composition rules to extant components. AIDE supports this idea through the use of *libraries*, which are collections of prototypical components that the may be stored between sessions and re-used in the creation of new components.

2.1.3. Behavior

Component behavior specifications must be efficient both in expression and simulation. AIDE uses the ZETALISP [10] language and programming environment directly in addressing both these concerns, paying the penalty of expecting the user to be a reasonably competent LISP programmer.

[1] Support for this work was provided by the following : DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

[2] AIDE Is-a Distributed Environment.

[3] A component is the basic unit of design in AIDE.

2.1.4. Implementation

It is natural to use the object-oriented programming paradigm to implement the components of a design, directly mapping from entities in some "real" world (of the designer's choosing) to the data objects manipulated by the design system; AIDE uses the object-oriented programming facilities provided by the FLAVOR system [10]. Every component is an **instance** of some component **class**, where the class defines a component type and is implemented as a *flavor*.[4] Structure is specified in terms of these flavors and behavior in terms of *methods* relevant to them.

2.2. Structure

To the design system, a component's structure consists of two parts :

- the component's own properties, and,
- the component's relationships with other components.

2.2.1. Component Properties

The designer sees a component as a "black box" of a particular type that has a collection of local named attributes with associated values. The allowable attributes of a component are defined by its type, while the *values* on these attributes may (and usually do) differ for each component instance. A subset of these properties[5], the **state** properties, are used by the behavior of the component. Special state properties known as **ports** (input and output) constitute a component's interface to its environment. Other automatically inherited properties are used by the system to maintain and display components.

AIDE provides the `defcomponent` form for a designer to define the properties of a new component type and it has a graphical editor to capture and alter display properties held by components.[6]

Figure 2-1 is a simple example of the component class declaration for an abstracted D-type flip-flop. Each instance of `d-flip-flop` has three input ports (named `d`, `clock`, and `clear`), one output port (named q), and no internal state.

```
    (defcomponent D-Flip-Flop
      (:input D Clock Clear)
      (:output Q)
      (:documentation "Class of positive-edge-triggered D-type
   flip-flop witb direct clear.  Uses 'higb, 'low and 'x
   logic signals.  Has unit delay between an input transition
   and stable output."))
```

Figure 2-1: Definition of the `d-flip-flop` Component Class

For a complete description of the `defcomponent` form see [7]; suffice it to say here that it translates into the appropriate FLAVORS declarations.

2.2.2. Structural Relationships

There are two structural relationships that hold between components :

- **Composition**. Any component may be a subcomponent of exactly one component and every component may be composed of any number of subcomponents. When a component is *composite* (made up of subcomponents), it may share its ports, for behavioral purposes, with those of its subparts through the "connection" relation.
- **Connection**. This relation holds between individual ports of two components and is specified by **lines** which *connect* the relevant ports. Lines may connect an output port of some component to an input port of another component except when connecting ports between a composite component and one of its subcomponents, in which case the connected ports are of the same type (port sharing). Usually a line connects just two ports; **contacts** are special entities that provide fan-in and fan-out capabilities for lines.

These structural relationships are captured by AIDE through its graphical structure editor.

2.2.3. Prototypes

Traditionally, object (frame) systems have had difficulty in implementing a general mechanism for capturing complex relationships that must hold between sets of instances of various classes. The "connection" structural relation is just such a relation - it is difficult to declare this information in the class definition of a composite component. The solution we have adopted in AIDE is to store connectivity information about a composite component type as a "canonical" instance of the relevant component class; this canonical instance is called the **prototype** of its class. The structure of a component class is therefore fully specified by the existence (in the environment) of both a `defcomponent` declaration and a prototype.

2.2.4. The Editor

A component in AIDE may be accessed through the graphics-based, menu-driven interface which provides operations for viewing and selecting components. Top-level components (*devices*) are maintained in book-keeping entities known as *worlds*, each of which may have several windows (*viewports*) viewing the relevant device. The editor uses the graphics-based interface in providing operations to create new devices and edit their structure, allowing the designer to create, alter and delete components, lines, ports and contacts. There are also facilities to copy devices into permanent file storage, prototize devices for inclusion in libraries, and load devices and libraries from file. A complete desciption of the operations provided by the editor may be found in [7].

2.3. Behavior

Behavior is defined by AIDE to be the interaction of a component with its environment over (simulated) time. A behavioral specification applies to a class of component; it is implemented by a *method* on the class that interacts with the simulator to generate the time-varying behavior of a component of that class. Since the simulator in AIDE is event-driven, this interaction takes the form of the consumption and production of **events**, which are encapsulations of the time-stamped state changes in the simulated system. Behavior for a component is therefore simply a specification that relates values on input ports with values on output ports over (simulated) time; components whose output values depend on a history of input values make use of their internal state properties.

[4] In the usual inheritance network.

[5] We use the term "properties" loosely to mean the collection of attributes and their values.

[6] A large part of the graphical interface was modelled after that used by HELIOS and PALLADIO.

AIDE provides the **defbehavior** form to declare the behavior of a componcnt class. Events relevant to a component are consumed when the simulator propagates the specified state change and then invokes the relevant component's behavior method; the simulator is informed of new events through the execution of the **assert** function within a behavior method, which specifies a change that will be true of some state of the component at some future simulated time.

2.3.1. An Example

Figurc 2-2 is an abstract behavioral specification for the d-flip-flop component class. The signal on the d input is transferred to the q output when the clock input goes from low to high. If, however, the clear input goes low, then so does the q output. The q output is unaffected by the d input whenever the clock is stable. The clock period is two simulated time units, and input setup time is ignored.

```
(defbehavior D-Flip-Flop (ignore state signal now)
  ;;  Clear  Clock  D      |  Q
  ;;  ---------------------|------
  ;;  low    x      x      |  low
  ;;  high   ↑      high   |  high
  ;;  high   ↑      low    |  low
  ;;  high   low    x      |  Q0
  (selectq state
    (Clock
     (when (eq (state-value (port-signal Clear)) 'high)
       (when (eq signal 'high)
         (when (< (- now (state-time (port-signal D)) 2))
           (assert Q (state-value (port-signal D))
                   (1+ now))))))
    (Clear
     (when (eq signal 'low) (assert Q 'low (1+ now))))))
```

Figure 2-2: Behavior Declaration for the d-flip-flop Class

There are a couple of points worth noting in the cxample of Figure 2-2.

- The style illustrates one of the benefits of event-driven simulation : only the state *changes* are propagated as opposed to recomputing the state of the entire system at every step [8].

- The declaration has an explicit notion of the passage of time; simulated time units have user-defined semantics and it is up to the designer to ensure that the units be used consistently by different components.

- The state changes specified by the cvents for a given simulated time are *all* made before behavior methods are invoked on the events. (This, however, excludes zero-delay events generated by the behavior methods, which must be dealt with more carefully. These are not considered in this report, but are handled by AIDE.) Hence, there is no need to specify a clause to handle a change in d occurring at the same simulated time as a clock transition from low to high, where the clock event is "processed" earlier in real time than the d event.

2.3.2. Composite Behavior

The benefits to be gained by hierarchical simulation are well-known; once the behavior of a multi-component system is verified, the designer may reduce simulation turnaround time by abstracting this behavior into a less detailed behavior that realizes the same function. AIDE directly supports this by allowing a designer to specify whether a composite component's behavior is its own defined behavior ("top-level") or the compounded behaviors of its connected subcomponents ("internal"). For example, if we designed a shift-register from D-type flip-flops, we might initially verify the design using the "internal" behavior of the shift-register, that is, the composite behavior of its flip-flops; later, when using a shift-register in the design of a control-unit, we might use a "top-level" characterization of its functionality.

How does composite behavior work? During simulation, events on output ports are immediately transformed into events on the furthest participating connected input ports (if any), and then forwarded to the simulator to be consumed by the relevant component at the specified simulated time.[7]. Hence, the effects of a local change propagate through the system along connection paths, achieving the required overall system behavior.

2.3.3. Behavior Requirements

A top-level behavioral specification is usually required to satisfy the following properties [2, 5] :

1. **Functionality**. Events generated on output ports of a component depend only on events consumed on its input ports and internal states.

2. **Realizability**. An event generated for simulated time t cannot depend on any events consumed by the component for simulated times greater than t. This simply reflects the notion that no real system can predict the future.

3. **Finite Delay**. An event on an input port or internal state with simulated time t cannot generate events on output ports with simulated time less than t. This reflects the idea that no real system can alter the past.

A quick inspection of Figure 2-2 should verify that the behavior specified for d-flip-flop satisfies these properties.

3. Design Validation

Once a design has been specified to a design system, the designer must be able to validate it by ensuring that it meets both its functional and performance goals. In the absence of formal verification methods, simulation is a common technique to establish the *functionality* of a design [8]. Furthermore, since simulation (unlike emulation) automatically carries with it an explicit notion of time[8] it can also be used to compare the *performance* of a design with other designs or real systems that realize the same function; this is often as important to the designer as verifying its functionality [2].

3.1. Discrete Event Simulation

While there are various types of simulation (see [6] for a good characterization of simulation methods), we are concerned here only with discrete-time, event-driven simulation. Before proceeding with our discussion, it is useful to consider some definitions.

3.1.1. Consistency and Acceptability

An **event** is an atomic state change in the simulated system during the execution of a simulation. It is represented as a record consisting of (1) a component. (2) the state or port of the component that changes, (3) the value that it gets, and (4) the simulated time of this change. Two events are equivalent if they are isomorphic (thus they represent the same state change to the simulated system, though for different executions of the simulation).

[7] Event transformation is done cooperatively by the components themselves through message-passing

[8] As construed by the designer.

Simulated time is the designer's abstraction of real time, so that the state of the real system (device) at any real time corresponds to the state of the simulated system (device) at the corresponding simulated time [6]. Simulated time takes on non-negative, discrete, and, for convenience, integer values.

The **simulation** of a component (device) refers to the *execution* of a simulation of a component (device) under the control of some simulation algorithm which regulates the consumption and production of events relevant to that component (device) over real time. For a given simulation, there is an associated set of events. We say that two simulations are equivalent if they produce equivalent event sets (given that the device being simulated is deterministic); two simulation algorithms are **consistent** if any two simulations under the control of each algorithm, respectively, are equivalent. The actions of a simulator to achieve consistency (using a simulation algorithm) are collectively called **synchronization**; hence the algorithm is often called a *synchronization algorithm*.

Lastly, we call a synchronization algorithm **acceptable** if it is consistent with itself and if it accurately reflects the behavioral specification of the simulated system. Intuitively, this means that a synchronization algorithm is acceptable if it always generates all and only those events induced by the initial state (including initial events) of the simulated system and the behaviors of the components being simulated.

3.1.2. Synchronization

Acceptability is the goal of every synchronization algorithm. Since almost every implementation of a simulator (including AIDE) depends directly on side-effects to changeable state[9], acceptability operationally means that the simulation algorithm must *control the consumption of events during execution so that behavior-generating code is invoked in the correct context*. (This is not necessarily the case; for example, a simulation system that uses a strict logic programming system to implement structural and behavioral specifications need not concern itself with this issue since all "state changes" will persist in such a system; of course, the burden of storage management has now been thrust upon the logic programming system.) With this implementation model in mind, we provide below an informal relation on events that will be useful in analyzing the acceptability of synchronization algorithms.

An event e_i **preempts** another event e_j if either of the following is true :

1. e_i and e_j specify a change to the same state entity but the simulated time of e_i is greater than the simulated time of e_j;

2. the state change specified by e_i overwrites information that is used by e_j and the behavior of the relevant component to generate an event.

Two events are **independent** if neither preempts the other.

We claim that an acceptable simulation algorithm is one that generates an event set such that for every e_i and e_j in the set, if e_i preempts e_j then e_i is "processed after" e_j.

In theory a simulator has to run the entire simulation to determine the set of preemption relationships between every two events; in practice, however, it computes a set of *possible* event preemptions, with the requirement that this set be a superset of the set of actual event preemptions. The problem of synchronization (distributed or otherwise) is thus essentially the problem of dynamically determining potential event preemptions and processing those events that cannot be preempted.

3.2. Sequential Simulation

We discuss briefly the mechanism by which sequential simulation works in AIDE.

3.2.1. Synchronization Using Simulated Time

The standard sequential synchronization algorithm makes use of the simulated time of an event and the requirement that the device is *realizable* to achieve acceptability. Events with lower simulated times are always "processed before" events with higher simulated times; therefore, whenever an event is processed, all the events that could possibly have preempted it have already been processed.

The main advantage of this synchronization algorithm is that it is simple and easily implementable in a serial system. However, it is too conservative in its computation of possible event preemptions to be viable in a distributed environment.

3.2.2. Implementation

AIDE implements a simulator as a flavor-instance that maintains a simulated-time-ordered *eventlist* and an associated *global clock* for a given device. At every step, the simulator removes the event at the head of the eventlist, moves the clock to the specified simulated time, makes the appropriate state change, and invokes the behavior method of the relevant component. Events generated by the behavior of a component are passed back to the simulator, which sorts them into the eventlist to be processed when they get to its head.

AIDE uses the graphical interface to allow the designer to access the simulator associated with a device. It provides operations to reset, initialize, and run a simulation with or without breakpoints [7].

Current facilities for "observing" a simulation are limited; a general instrumentation interface is under design.

3.3. Distributed Simulation

The motivation for distributed simulation is doing event processing in parallel using multiple machines to gain a reduction in the overall simulation turnaround time as compared to a sequential simulation. Thus, synchronization algorithms for distributed simulation systems seek ways of processing non-preemptable events in parallel. These algorithms must trade off the cost of determining potential event preemptions against the cost of processing the events themselves in minimizing the total execution time of the simulation. Such costs, naturally, depend on various factors, including the target machine environment. Our discussion below assumes a machine environment that consists of small number of fairly powerful machines (*Symbolics* 3600s) communicating over a shared network (the ETHERNET).

Though there are various classes of synchronization algorithms for a distributed environment [6], we only consider those which distribute control of the simulation to the participating machines, that is, algorithms that are run individually by each machine.

[9] There is a direct correspondence between a state variable in the specification and one in the implementation of the specification, dictated by storage management considerations.

3.3.1. Partitioning

Decomposition is not only a powerful tool in design, but also in distributed problem-solving. It is therefore natural to consider various ways of partitioning the problem of simulation into subproblems which may be tackled by the participating machines individually. In doing this partitioning, we must keep in mind that we would like each machine to operate as autonomously as possible and also that there are costs associated with communicating information between machines which we would like to minimize.

Usually the structure of a device (system) directly reflects its functionality. Given the nature of the design representation, this implies that the subcomponents of the device themselves behave fairly autonomously. This in turn points to the obvious utility of partitioning the simulation problem by assigning to each machine the subproblem of simulating some subset of the components of the device (system) as this will tend to reduce the gross interactions (and shared state) between the machines, thus reducing the costs associated with communicating and keeping consistent such information. This partitioning will also allow each machine to operate reasonably independently. Furthermore, if the system being modelled *itself* exhibited concurrent activity (a multiprocessor computer system, for example), then this partitioning scheme may enable the overall simulation of the system to directly exploit the natural parallelism visible in the events that represented the "actual" concurrency. The above are, in fact, basic assumptions of the AIDE distributed simulation approach, as they are of most other distributed simulation schemes [5, 2, 6].

3.3.2. Using the Device Specification in Synchronization

Synchronization based on the simulated times of events alone unnecessarily (and, in most cases, severely) restricts the amount of exploitable parallelism by assuming that an event with simulated time t could be preempted by *any* event for simulated time less than t. Very few abstract models (for example, CARE, which mixes detailed simulation of inter-processor communication with more abstract simulation of processing activities) exhibit such synchronicity at the event level, thus there will be very few opportunities for parallel processing in their simulation. The device specification and the behavior requirements provide additional information for better estimating potential event preemptions.

Since the preemption relation applies between events, the more information within an event used by the synchronization algorithm, the closer its synchronization activities come to using the results of the simulation itself, and the better the estimation of preemption relationships. We may organize these pieces of information in terms of the "fields" of an event.

1. *Simulated time* is already essential, as the definition of preemption and the implementation of a behavior specification suggests. We may make use of the property that two events with the same simulated time are always independent to find inherent parallelism.

2. The *component* is also useful within the partitioning scheme we have chosen. Since each component has a minimum (non-zero) simulated time *delay* between consuming an event and generating one on an output port, and since it is also directly *connected to* only some small subset of the other components, an event for that component will have a simulated time "lag" before it may preempt an event on a component more "distant" in terms of connections. This enables a machine simulating the "distant" component to process existing events for it up to "lag" simulated time units beyond the event for the original component in parallel. Connection information is available in the structural specification of a device; minimum delays may be extracted from behavior specifications.

3. The *state* being changed within a component is useful when a component has a number of internal states that affect its output ports with varying delays. This gives better bounds for "lag" on a per-event basis within such a component, thereby giving a better overall approximation of possible preemptions. Such information can be determined as for the component itself..

Much of the above information can be efficiently "compiled" before the actual execution of a partitioned simulation. However, part of it must still be computed dynamically by the machines, communicated between them and finally used by them, perhaps undercutting the increased opportunities for parallelism.

3.3.3. The First Cut

We describe here the first synchronization approach used in AIDE, which reflects a particular choice of only the first two information sources described above for implementation simplicity.

Distributed simulation in AIDE starts with the designer selecting the partitioning level for the device in terms of its subcomponents. At this level, the component and all its subcomponents form a *logical process* or *lp* within which simulated time will be consistent. Different *lps* may have different simulated times during a simulation, even within a machine. Thereafter, the designer partitions the simulation by assigning lps to machines.

At this point, AIDE compiles synchronization information and distributes components to *simulation servers* on each (previously obtained) machine; a server is essentially a sequential simulator plus support for synchronization.

The synchronization information compiled here is at two levels. Between machines, the system first computes a table that represents gross minimum delays along connections between any lp on one machine and any lp on another. Within a machine, lps are organized in terms of simulated time *windows*. Window-out(lp_i) is the minimum simulated time delay before an event consumed at lp_i could generate an event at for any non-local lp. Similarly, window-in(lp_i) is the minimum simulated time delay before an event consumed by an "edge" lp (one with a direct connection from any remote lp) could generate an event for lp_i. These quantities are static for a given partition and are directly computed from the structure and a predeclared minimum delay for each component type.

During execution, a simulation server runs a cycle with two phases.

- **Synchronize**. If the server was active (processed some local events) in the last step, it computes the minimum time that an existing local event could affect any remote server. This quantity is the next event time (NET) of the server and is equal to the minimum over all the local events of the simulated time of the event plus the window-out of the lp specified in that event. It sends this time to every other server in a *synchronization message*.

  Each server now waits for all servers active in the last step to send their NETs. Then it uses the compiled inter-machine delay table to form the next set of active servers as follows :

For server s_i, $\forall s_j$ $\{\mathrm{NET}(s_i) < [\mathrm{NET}(s_j) + \mathrm{delay}(s_j,s_i)]\} \Rightarrow$ active(s_i).

Each server also computes the local *preemption time*.

$\mathrm{PT}(s_{\mathrm{local}}) = \mathrm{minimum}\{\forall s_j[\mathrm{NET}(s_j) + \mathrm{delay}(s_j,s_{\mathrm{local}})]\}$.

- **Simulate.** Each server processes all the existing local events that cannot be preempted by an event that occurs at an *edge* lp with time not less than $\mathrm{PT}(s_{\mathrm{local}})$ (using the window-in of lps). Events from remote machines may be asynchronously received for input ports of local edge components, but they will be for simulated times greater than $\mathrm{PT}(s_{\mathrm{local}})$. Similarly, events on local edge output ports may be transmitted asynchronously to remote input ports.

The calculations of activity and preemption times ensure that the set of event preemptions computed by each server is acceptable. Deadlock is avoided by requiring non-zero delays within components [6]. Lastly, inconsistent information regarding NETs is avoided by using the same reliable stream to transmit events as well as synchronization messages between servers and by having each server include in the synchronization messages the minimum times of remote events generated for every server during that step.

3.3.4. Evaluation and Implications

To evaluate the AIDE algorithm, we use a probabilistic model of a simulated multiprocessor (CARE) induced from its event history in a serial simulation. As mentioned earlier, CARE exhibits clusters of "communication" events (representing packet routing between nodes) that are localized in simulated time as well as over the processor grid intermixed with slower "computation" events (representing processing activity within a node) that have larger, more varying simulated time periods. In using a probabilistic model, we bypass many of the additional issues involved in distributing CARE programs while still retaining information that allows us to predict the performance of a distributed simulation of the model.

Preliminary runs of this model and others using a small number of machines (1 to 4) indicate that the implementation does attain speedup *when concurrency is available*. In the probabilistic CARE model, the "window" mechanism seems to reduce synchronization points by a factor roughly proportional to the number of components on a machine. However, we also observe that there are very few opportunities for parallelism across the machines; rarely is more than one machine active during any given step. This immediately places an upper bound (somewhere between 1 and 2) on the speedup that may be gained by the distributed simulation.

We can suggest two reasons for the above. The first is that the probabilistic CARE model was generated from early applications that did not themselves demonstrate much low-level concurrency atop CARE. The second (more probable) is that the preemption calculations were too simplistic.

We are taking steps to alleviate the above difficulties. One step is to use a probabilistic model extracted from the event history of a demonstrably concurrent application in CARE. Another is to increase the complexity of compiled synchronization information in attempting to increase the number of active machines at any step. The latter involves the following specific actions :

- make use of the third facet of information present in an event in determining preemptions, either through declarations or by "wiring" such information into the behavior specification of a component class;
- at compile time, compute better lower-bound delays between machines by searching connection paths between all machines (the first implementation only did it for neighbors and then used a plane assumption);
- compute preemption times on a per-machine basis as opposed to the conservative strategy of using only the "most dangerous" machine.

We anticipate that the above changes will result in increased parallel activity for the machines (a necessary condition for speedup); thereafter, we will determine whether the added cost of maintaining and using this information will negate (or worse) this gain.

References

[1] Harold Brown, Christopher Tong, and Gordon Foyster.
Palladio: An Exploratory Environment for Circuit Design.
Computer Magazine 16(12):41-58, December, 1983.

[2] Randall E. Bryant.
Simulation of Packet Communication Architecture Systems.
Technical Report MIT/LCS/TR-188, Laboratory for Computer Science, Massachusetts Institute of Technology, November, 1977.

[3] Bruce Delagi, Jerry Yan.
CARE User Manual.
HPP Report, Stanford University, Department of Computer Science, 1986.
[in preparation].

[4] Gordon Foyster.
HELIOS User's Manual.
HPP Report HPP 84-34, Stanford University, Department of Computer Science, August, 1984.

[5] Jay Misra.
Distributed Simulation.
Tutorial notes, ICDCS, IEEE Computer Society.

[6] J. K. Peacock, J. W. Wong, and E. Manning.
Distributed Simulation using a Network of Processors.
Computer Networks 3(1):44-56, February, 1979.

[7] Nakul P. Saraiya.
AIDE User Manual.
HPP Report, Stanford University, Department of Computer Science, 1986.
[in preparation].

[8] Narinder Singh.
MARS: A Multiple Abstraction Rule-Based Simulator.
HPP Memo HPP 83-43, Stanford University, Department of Computer Science, December, 1983.

[9] Narinder Singh.
Corona: A Language for Describing Designs.
HPP Report HPP 84-37, Stanford University, Department of Computer Science, September, 1984.

[10] Daniel Weinreb and David Moon.
LISP Machine Manual.
Symbolics, Inc., 1981.

RECENT DEVELOPMENTS IN NIKL

Thomas S. Kaczmarek
Raymond Bates
Gabriel Robins

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

Abstract

NIKL (a New Implementation of KL-ONE) is one of the members of the KL-ONE family of knowledge representation languages. NIKL has been in use for several years and our experiences have led us to define and implement various extensions to the language, its support environment and the implementation. This article reports on the extensions that we have found necessary based on using NIKL in several different testbeds. The motivations for the extensions and future plans are also presented.

1. Introduction

Our work on NIKL is motivated by a desire to build a principled knowledge representation system that can be used to provide terminological competence in a variety of applications. To this end, we have solicited use of the system in the following applications: natural language processing, expert systems, and knowledge-based software. Our research methodology is to allow application needs, rather than theoretical interests, to drive the continued development of the language. This methodology has allowed us to perform an empirical evaluation of the strengths and weaknesses of NIKL. Also it has helped us identify some requirements for any knowledge representation tool that would be used in a wide range of intelligent systems.

We classify the improvements that we have made or plan to make into three broad categories:

1. Expressiveness - enhancements to the terminological competence represented in NIKL and the inferences NIKL can make regarding the subsumption relationship,

2. Environment - enhancements to the tools that accompany NIKL for both maintaining knowledge bases (knowledge acquisition) and reasoning about the terminology defined in the knowledge base, and

3. Support - enhancements to user documentation, the reliability and the availability of the implementation.

This paper will concentrate on enhancements made to the expressiveness of NIKL but will also describe some improvements and additions made to the NIKL environment. An Introduction to NIKL will be included as background material and enhancements to the support of NIKL will be mentioned for the sake of completeness.

2. Background

KL-ONE was designed by [Brachman 78] to "circumvent common expressiveness shortcomings." It was designed to embody the principles that *concepts* are formal representational objects and that *epistemological* relationships between formal objects must be kept distinct from conceptual relations between the things that the formal objects represent. KL-ONE defined an "epistemologically explicit representation language to account for this distinction."

A KL-ONE concept is described by "a set of functional roles tied together by a structuring gestalt." Concept definitions "capture information about the functional role, number, criteriality and nature of potential roles fillers; and 'structural conditions', which express explicit relationships between the potential role fillers and give functional roles their meaning." A overview of the KL-ONE system has been published by [Brachman and Schmolze 85].

2.1. The classifier

An important consequent of the well-defined semantics of KL-ONE is that it is possible to define a *classification* procedure to determine the subsumption relationship for concepts in a KL-ONE network. A detailed description of the semantics of the KL-ONE classifier have been published by [Schmolze and Lipkis 83]. The classifier for KL-ONE deduces "that the set denoted by some concept necessarily includes the set denoted by a second concept but where no subsumption relation between the concepts was explicitly entered." Classifiers for KL-ONE and NIKL have been developed at ISI.

The desirable properties for the classification algorithm are soundness (no incorrect inference is made), completeness (all correct inferences are made), and totality (the algorithm always halts). Theoretical analysis work done by [Brachman and Levesque 84] has determined the limits on the expressiveness if completeness of the classification algorithm is to be maintained. Work on NIKL has concentrated on the issue of soundness, forgoing completeness in favor of Increased expressiveness. An efficient implementation has also been a goal of the NIKL effort and the NIKL classifier is in fact nearly two orders of magnitude faster for large networks than the KL-ONE classifier.

2.2. Classification-based reasoning

The NIKL classifier provides a general weak method for categorizing descriptions of objects. It is insufficient as the sole inference mechanism for an intelligent system but it can be used very effectively (and efficiently) in what we have termed *classification-based reasoning*.

Most uses of KL-ONE and NIKL rely heavily on this kind of reasoning. It consists of a classification-reasoning cycle. The application first creates a new description of some partial result and then classifies this in a static network describing knowledge of the problem domain. Based on the result of classification, additional inferences are drawn about the partial result and a new description is constructed. These inferences are the result of some rule or procedure that examines the network looking for inferences that it is capable of making. The new description that results may achieve the goal of the reasoning cycle, in which case reasoning terminates. More typically, further classification and redescription are required and there is a continuation of the reasoning cycle.

One way of thinking about this reasoning cycle is to think of the classifier as selecting applicable rules based on the terminology that is used to describe the task domain and the problem at hand. The selection of the rules is within the terminological system, i.e., based on the definitions of terms. However, the rules are outside the terminological component and expressed in some other language.

2.3. NIKL's evolution from KL-ONE

NIKL's name is evidence of the fact that it is thought of as a New Implementation of KL-ONE. Despite this, there are major differences between NIKL and KL-ONE. These are in addition to the emphasis on the efficiency of the classification algorithm already mentioned. Many of the differences are a direct result of the influence of work on KRYPTON by [Brachman, Fikes, and Levesque 83]. Close cooperation between the NIKL design team and the KRYPTON designers resulted in many system similarities despite a strong distinction on the issue of completeness.

The major difference between NIKL and KL-ONE involves the representation and use of roles.[1] At the time NIKL was designed, use of KL-ONE had uncovered a need for revisions of the ideas about roles. For example, explicit structural conditions were no longer used to define the meaning of roles partially because of the inadequacy of the original formalization and lack of useful consequences of these conditions. In addition, the notation required in KL-ONE for relating roles in concepts (which included relations such as modifies, differentiates, and individuates) were cumbersome. The idea of thinking of roles as two-place relations and concepts as one-place relations emerged, and roles took on a new significance. Roles were defined as having a domain and a range, organized in a separate taxonomy, thought of as representations of relations, and assumed to be used consistently.

3. The status of NIKL

A NIKL implementation was first developed approximately two years ago. Since then it has been in use principally at ISI and at Bolt, Beranek, and Newman Inc., which contributed to the design of the system. Several "browsing" tools, syntactic support, and graphing tools have been developed and used to construct and maintain knowledge bases. A natural language paraphraser to assist users in understanding networks was also developed but has not been heavily used. Various inference mechanisms driven by the classifier have also been implemented.

Applications of KL-ONE and NIKL have been in the areas of natural language processing (see the publications of [Bobrow and Webber 80, Sondheimer 84, Sidner 85, Mark 81]), expert systems (see the work of [Neches, Swartout, and Moore 85]), and software description (see the publications of [Kaczmarek, Mark and Wilczynski 83, Wilczynski 84]). Large networks, in excess 1500 concepts, have been developed in these environments.

This experience with NIKL has led us to consider certain extensions to the language, its environment, and the implementation. The following sections will describe the extensions we consider important and explain the motivation and status of each. The extensions have been divided into roughly three categories: terminological competence, environment, and implementation.

3.1. Terminological Competence

By terminological competence we mean the ability of the system to *represent* and *reason about* various distinctions that a modeler might need to capture in defining concepts. For example, the ability to restrict the range and number of role fillers for a particular functional role adds to the terminological competence. Inferring that if a person has at least one son, then the person has at least one child (based on the fact that son is a specialization of child) is another example of terminological competence. The following sections will describe our efforts in this area.

3.1.1. Disjointness and covers

One addition to NIKL that was absent in KL-ONE is support for disjoint and covering sets. A collection of concepts can be declared as being disjoint, i.e., have no common extensions in the world. A collection can also be declared as a cover of another concept, i.e., all extensions of the covered concept must be described by at least one of the members of the covering. These two declarations can be combined to form partitions.

NIKL supports limited inferences based on these notions.

- As a result of disjoint classes, NIKL can determine if a concept is coherent or not. For example, a person all of whose children are both males and females, would be marked as being incoherent if male and female were declared as being disjoint. An incoherent description is admissible in NIKL but is assumed to not have any extension in the world.

- With respect to covers, a simple inference procedure is available to deduce the existence of other covers. For example, suppose male and female cover sex, spouses have a sex role that is restricted to sex, and that husband and wife are specializations of spouse. Further assume the only difference between husband and wife is a restriction of the sex role to male and female respectively, then NIKL can infer that husband and wife cover spouse. Needs for this kind of reasoning have come about in using NIKL for expert systems where certain methods of problem solving are applicable only when some covering exists. NIKL's current inferential capabilities for covers are limited to simple cases such as the one presented in this example. Plans call for expanding these capabilities as needed by applications.

[1] Actually there are significant differences beyond those having to do with roles if one takes KL-ONE to be defined by the original formalization rather than the then current implementation, which did not support much of the formalism.

3.1.2. Reasoning about role restrictions

The inclusion of an explicit role hierarchy in NIKL allows the system to infer certain properties of concepts. The example of calculating minimum number restrictions for the son and child roles presented above illustrates one kind of inference. In that example, we have propagated a minimum number restriction up to a more general role. Obviously, we can also propagate a maximum down to a specializing role. These are two inferences that we have recently added to NIKL.

Another inference involves value restrictions for roles. It is illustrated by the network definition seen in Figure 3-1. The NIKL specification for this example can be paraphrased as follows:

- doctors, famous, and rich are primitive concepts[2],
- surgeons are a primitive specialization of doctors,
- very famous is a primitive specialization of famous,
- all the rich cousins of an "A" must be doctors,
- all the famous cousins of any "B" must be surgeons and all the rich relatives of any "B" must be very famous,
- relative is a primitive relation,
- cousin is a primitive specialization of relative
- any concept that fills the role of famous cousin must fill the role of cousin and be famous,
- any concept that fills the role of rich relative must fill the role of relative and be rich and,
- any concept that fills the role of rich cousin must fill the roles of rich relative and cousin.

```
(DEFCONCEPT Doctor primitive)
(DEFCONCEPT Famous primitive)
(DEFCONCEPT Rich primitive)
(DEFCONCEPT Surgeon primitive
    (specializes Doctor))
(DEFCONCEPT Very-Famous
    (specializes Famous))

(DEFRELATION Relative primitive)
(DEFRELATION Cousin primitive
    (specializes Relative))
(DEFRELATION Famous-Cousin
    (specializes Cousin) (range Famous))
(DEFRELATION Rich-Relative
    (specializes Relative) (range Rich))
(DEFRELATION Rich-Cousin
    (specializes Rich-Relative Cousin))

(DEFCONCEPT A
    (restrict Rich-Cousin (VR Doctor)))
(DEFCONCEPT B
    (restrict Rich-Relative (VR Very-Famous))
    (restrict Famous-Cousin (VR Surgeon)))
```

Figure 3-1: Example of role reasoning

From this specification, NIKL infers the following:

- all of A's rich cousins are rich doctors,
- all of B's rich cousins are rich and very famous surgeons,
- all of B's famous cousins are famous surgeons, and
- all of B's rich relatives are rich and very famous.

Figure 3-2 graphically depicts the network after classification has been performed.

The conclusions illustrated in the figure are derived from the following line of reasoning. All of B's rich cousins are rich relatives and therefore very famous, so they are all also surgeons (since all the famous cousins of B are surgeons), making them doctors as well. It follows then that B specializes A since all of its rich cousins are rich and very famous surgeons, which is a specialization of rich doctors. The current classifier for NIKL supports this kind of reasoning based on the role hierarchy.

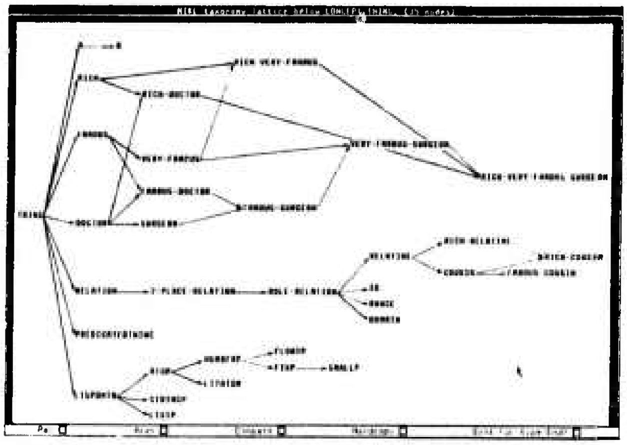


Figure 3-2: Graph of taxonomy defined in Figure 3-1

Our plans for enhancing reasoning about role restrictions include adding logic to account for coverings and disjointness in the role hierarchy. For example, if we knew that the roles, son and daughter, are disjoint and that they cover the role, child (i.e., form a partition) then we can determine the maximum and minimum number restrictions for child based on the number restrictions for son and daughter. Similar kinds of inferences can be made involving the value restrictions.

3.1.3. Roles and relations

One of the criticisms of KL-ONE and NIKL was an incomplete treatment of roles. In KL-ONE the semantics for roles was determined only by other constructs that were described for concepts. In previous versions of NIKL, all roles were primitive. Work in natural language text generation has pointed out the need for a more uniform treatment because sometimes a sentence needs to describe the relationships that exist between concepts. This requires giving relations the same status as concepts in the network and establishing a correspondence between restrictions of roles at a concept and the relations those restrictions refer to.

[2] A primitive concept or relation corresponds to the notion of a "natural kind", i.e., a predication that can only be determined by an oracle. To NIKL this means that no concept may be placed beneath this one in the hierarchy unless the concept specification explicitly says to do so.

We have thus adapted a position where roles are thought of as two place relations that are defined in the concept hierarchy. We have implemented this strategy by allowing the user to define relations that may then be used as roles. The example above in Figure 3-1 illustrates thls capability. Under this new implementation, relations are represented as concepts in the same hierarchy with all other concepts. All relations have at least two roles, a range and a domain.

One implication of this support is that it has allowed the user a simple way to say things such as "a car, one of whose tires is flat." In the previous implementation, the user would have to specify and name a primitive role that specialized the tire role for a car and then restrict the value of that role.

A more significant improvement results from removing an unfortunate consequence of this old procedure (which resulted from the primitiveness of the role). The result of that procedure was that nothing would classify as a kind of the concept being defined unless the user added the same role (presumably by referring to it by name) and restricting if to the same range (or some specialization of it). In the current implementation, we "gensym" a relation that specializes tire and restrict its range to flat. Any other similar or more specialized relation resulting from a restriction, for example, the one generated by "a car with a blown-out tire," will either merge with the gensymed relation or classify as a specialization of it. Thus, classification of a car with a blown-out tire under a car with a flat tire can happen without having to refer to a specific (and primitive) flat-tire role in the specification of the car with a blow-out.

Since relations are now part of the concept hierarchy, we can define other properties for roles and declare disjointness and coverings. One consequence of this is that we have simplified the development of support for reasoning about number and value restrictions for roles based on these notions. Another is that we can specify more completely the meaning of a relation.

3.1.4. Cycles in the network

The current NIKL classifier cannot reason effectively about cycles in the network. A cycle occurs whenever one classification depends on another. In general, the classifier stops trying to draw inferences about any of the concepts in a cycle when one is encountered. Typically a large collection of static concept specifications are presented to the classifier. It recursively descends the known hierarchy to find and classify those new concepts that have no dependencies on any other new concepts. It then unwinds the recursion and forms the newly classified hierarchy as a result. If it discovers a cycle, it simply declares the concepts classified and warns the user about the existence of the cycle.

The exception to this processing involves cycles that result from roles being defined as concepts. For example, if the son relation is used to define a person, then person cannot be classified until the relation son has been classified. But if the domain of son is person, then it cannot be classified until person is classified. Obviously, a cycle results. The current NIKL classifier detects this special case of a cycle and marks the relation as being classified and it continues to attempt to classify the concept that used the relation.

A more sophisticated classification control strategy could obviously result in a more complete classification. We have designed, and are in the process of implementing and testing, what we call the *incremental classification control strategy*. Under this regime, the classifier will maintain dependency links for all concepts and use an iterative approach to classification. When a cycle ls encountered, the classifier will do the best it can with the concept with the fewest dependencies. It will then classify all those concepts that depend on that one and eventually (because of the cycle) try to reclassify the original concept after having done its best on the dependent concepts. This approach obviously cycles and needs a termlnation condition. The incremental classifier will stop classification when the network has reached a quiet state, i.e., no new inferences can be drawn, or some user-settable number of dependency cycles have been completed. This strategy will allow more inferences to be made by the classifier and will also provide the basis for a much improved knowledge acquisition environment. Details of the implications for acquisition will be presented later in Section 3.2.4.

3.1.5. Partial orderings

One glaring shortcoming of KL-ONE and NIKL has been an inability to define sequences. Requests for this capability have come from nearly all applications[3]. We have examined the requirements and designed a more general capability that supports partial orderings on roles.

The partial orderings in NIKL represent relations that exist between role fillers. Support includes knowledge (in the classifier) about the reflexive, antisymmetric, and transitive nature of partial orderings. One partial ordering may be a specialization of another and they are defined in the concept hierarchy like all other relations.

The NIKL user can make several different kinds of statements about the partial orderings of the role fillers. One states that all the fillers of a particular role must be ordered by a particular relation. For example, the statements of a computer program are ordered by the lexically-before relation. A second kind of statement is that all the fillers of one role are related to all the fillers of another role by a particular ordering. An example is a statement that the initialization steps of a while loop come before the termination tests, which in turn come before the steps in the body. The final kind of statement declares that the fillers of one role are the *immediate* predecessors (or successors) of the fillers of another. An example is the statement that one statement of a program is immediately lexically-before another.

Classification will involve the determination of subsumption between partially ordered sets (posets), which is a fairly expensive operation. The expense includes the construction of the representation of posets as graphs and the determination of whether one graph is a subgraph of another. The design of the implementation is such that overhead caused by this enhancement will be minimal for concepts that do not involve use of this feature.

[3] Various extra-NIKL schemes have been adopted in past work to handle this problem. In past applications, it was not necessary for the classifier to deal with sequences so a special purpose sequence reasoner could be used.

3.1.6. Necessary and sufficient conditions

The NIKL classifier represents a particular kind of classification, one that depends on certain logical properties. There are other kinds of classification that depend on domain specific knowledge. One such kind of classification involves the definition of *sufficient conditions*. The idea is that the presence of certain evidence is sufficient to draw a conclusion if there is no contradictory evidence. For example, one might be willing to say that any mammal with a human DNA structure must be a kind of human unless there is evidence to the contrary even though we do not have evidence for upright posture, opposing thumbs and so forth.

Such reasoning has heretofore been unavailable in NIKL and KL-ONE. In light of this one can characterize the definitions of current NIKL concepts as stating necessary conditions (since no part of the description could be missing) and sufficient conditions (since the presence of them is sufficient evidence for the classifier to draw specialization conclusicns). The exception to this is for concepts marked as primitive, which indicates that no set of sufficient conditions can be found.

The proposal for adding sufficient conditions would allow the user to state that some collection or collections of roles were sufficient. For example, if you know that an animal has four legs and a trunk or a finger on the end of its nose (and there is no contradictory evidence, such as it lives in a tree) then it is an elephant. Still in question is the proper handling and possible inclusion of other constructs of the description language, such as structural descriptions and partial orderings. Our plan is to proceed with defining sufficient conditions in terms of roles and role sets and see if applications will require more complex support. An initial investigation indicates that this limited support will suffice.

3.1.7. Negation

Negation is a problem for the classification algorithm as has been shown by the work of [Brachman and Levesque 84]. Nevertheless, it is a notion that nearly all applications find useful. Since we cannot admit negation and maintain decidability for the classifier, we have provided other mechanisms and conventions that seem to satisfy most users. One convention is the use of zero as the minimum and maximum number restriction for a role restriction. For example, a verb phrase with no time modifier can be modeled this way.

The ability to define partitions as disjoint covers provides a way to talk about complements, which are akin to negation. This is another addition to NIKL that was the result of expressed desires for negation. The strategy exemplified in these two capabilities, namely, providing something different than what the user asked for but which meets the requirements of the application is very much a part of our methodology for continuing the evolution of NIKL.

3.2. The Environment

The NIKL environment consists of tools that aid in knowledge acquisition and reasoning. Our experience has led to the generation of tools in both of these areas.

3.2.1. Assertions

Recording and reasoning about extensions of the terminological knowledge represented in NIKL is considered to be outside of NIKL itself and part of the environment. An *ad hoc* assertional mechanism[4] was developed for use with the CUE and Consul applications (see, [Kaczmarek, Mark, and Sondhelmer 83]). A more systematic approach has led to the development of a major tool for reasoning about assertions by [Vilain 84] of Bolt Beranek and Newman. This tool, KL-TWO, combined the RUP package of [McAllester 82] with NIKL. KL-TWO provides a truth maintenance package that is very useful in some applications. However, it is inappropriate for large data bases and for certain kinds of applications where efficient implementations of the assertions are required.

To correct these deficiencies (for certain applications) we have planned two other hybrid systems. The first involves coordination between the conceptual hierarchy defined in NIKL with the schemata for a commercial relational data base. With this scheme we plan to use NIKL in applications requiring the kinds of semantic browsing techniques found in the work of [Patel-Schneider, Brachman, and Levesque 84] and [Tou, Williams, Fikes, Henderson and Malone 82]. The second involves using NIKL in coordination with the knowledge representation aspects of a knowledge-based software development paradigm. Here we are actively involved in using NIKL to define a type hierarchy and relations for the AP5 language of [Cohen and Goldman 85].

3.2.2. Reformulation

As was previously mentioned, classification-based reasoning is a common mode of use of NIKL. The terms, reformulation and mapping, have been used in KL-ONE applications to refer to this kind of activity. Currently there is a reformulation facility available that is used in the expert system research of [Neches, Swartout, and Moore 85]. This mechanism is used to satisfy goals by expanding plans. Within the paradigm of their project, reformulation is used to generate an expert system based on a knowledge of the domain and expert problem solving knowledge. In this methodology, goals, methods, and plans are all expressed in NIKL and the expert system shell uses these to generate the expert system for a particular domain and set of goals and methods. While the facility provided was designed for a particular use, the mechanism is generic and can be applied to any number of other applications.

3.2.3. Graphic-based editing

The KL-ONE community has a rich tradition of drawing pictures with "circles and arrows." A graphical representation of concepts and networks has always been a part of the language. As the expressiveness of NIKL has increased, the cleanliness of the graphs has diminished, but nevertheless, the graphs remain useful.

We have developed an integrated set of acquisition tools in a window-based workstation environment. The tools include a graph of the concept hierarchy, an EMACS editing window, and a LISP interaction window. Within the LISP interaction window, the environment can produce highly formatted ("pretty-printed") descriptions of concepts. The atoms in these formatted displays, which refer to concepts and relations, as well as the nodes of the graph and the text in the edit buffer are all mouse sensitive and known to be NIKL constructs by the environment. This allows the

[4] This scheme was built around the KL-ONE notion of a *nexus*

user to move from one window to another in a coordinated way. It also allows the user to refer to a NIKL object simply by pointing at it in any of the various views of the network. A natural language paraphraser has also been added to this environment to assist in the understanding of the network.

We also have a tool to graph the definition of a particular concept. This tool has proven to be less useful than originally thought. While drawing concept specifications on paper with a pencil is extremely useful, we haven't been able to duplicate the free flowing expressiveness of that mode of design. Work on the human factors of the tool and the inclusion of higher level operations (the current level is, for example, add a role) are anticipated. However, the tool is useful in terms of providing a graphic presentation of a concept. The deficiencies become obvious in creating or editing a concept definition.

3.2.4. Incremental classification

A major problem with the NIKL environment arises from the batch nature of the classifier. The example in Figure 3-1 illustrates some of the many inferences that the classifier makes. For example, deciding that the user really meant rich cousins to be rich doctors, not just doctors. This kind of inference can be particularly troublesome for the user because NIKL frequently needs to generate new concepts that the user hasn't explicitly defined. Usually NIKL cannot pick an appropriate name for the concepts it generates. In many cases the need to generate a new concept arises from the fact that the user has inadvertently omitted the concept or made some modeling error. A better acquisition environment can be obtained by having the classifier interact with the user whenever such a concept must be generated. The user could then choose an appropriate name, decide there is an error, or tell NIKL that the concept will be defined later.

The example of interaction arising from new concepts being generated is just one case in which interaction during classification can improve the modeling environment. The control strategy that will be employed in the incremental classifier will be much more supportive of the kind of interaction that knowledge acquisition requires.

The dependency information that the incremental classifier will keep can also be used to enhance the modeling environment. This information is particularly useful for editing a concept definition and then making sure the network is properly updated and for supporting various kinds of analysis tools.

3.2.5. Surface language support

As part of our efforts we have used a general lexical analysis and semantic interpretation package developed by [Wile 81]. This package gives a flexible surface language that allows easy modifications to accommodate extensions to NIKL as we develop them. It also opens up the possibility of defining highly application dependent surface languages.

3.3. The Implementation

The current version of NIKL is in Common LISP and we have experimented with its use on a variety of workstations and mainframe implementations of Common LISP. The integrated acquisition environment depends on some specific tools found in the Symbolics ZETALISP environment. We are actively pursuing the development of similar facilities that rely on a Common LISP implementation of a form and graphics package that requires only modest customizations for various graphic environments.

4. Summary

NIKL is an evolving knowledge representation tool based on KL-ONE. The experiences gained in a variety of applications have shaped the current implementation. Principal enhancements made to NIKL that were in direct response to applications needs were: the representation of roles more uniformly with concepts, support for negation, a connection to an assertional truth maintenance system, support for domain specific reasoning (triggered by classification), and more complete inferences drawn as a result of having a relation hierarchy. Further enhancements have also been suggested and continue to be developed. They include: the representation of sequences and orderings, the availability of sufficiency reasoning in the classifier, more complete inferences regarding cycles in the models, and coordination with an assertional component that supports efficient data base access.

In addition we have implemented and continue to develop tools for the knowledge acquisition environment. This work has also been sensitive to the needs that have arisen out of several application environments. Principle developments include a Common LISP implementation, and an integrated tool set that features graphic representations, formatting and paraphrasing tools, and flexible lexical analysis support. The addition of a more interactive editing style and various analysis tools is forthcoming.

5. Acknowledgement

The development of NIKL and our plans have been the result of interactions with a number of AI researchers. Many of these were developers or experienced users of KL-ONE or one of its variants. The rest were potential or new users. The contributors represent many different organizations and research interests. The following have all made, and in many cases continue to make, significant contributions: Don Cohen, Neil Goldman, Bill Mann, Norm Sondheimer, Bill Swartout, Bob Neches, Don Voreck, Steve Smoliar, Ron Brachman, Victoria Pigman, Peter Patel-Schnieder, Richard Fikes, Ramesh Patil, Jim Schmolze, Rusty Bobrow, Marc Vilain, Bill Mark, David Wilczynski, Mark Feber, and Tom Lipkis.

References

[Bobrow and Webber 80] Robert Bobrow and Bonnie Webber, "Knowledge Representation for Syntactic/Semantic Processing," in *Proceedings of the National Conference on Artificial Intelligence*, AAAI, August 1980.

[Brachman 78] Ronald Brachman, *A Structural Paradigm for Representing Knowledge*, Bolt, Beranek, and Newman, Inc., Technical Report, 1978.

[Brachman and Levesque 84] Ronald J. Brachman and Hector J. Levesque, *The Tractability of Subsumption in Frame-Based Description Languages*, Fairchild Research Laboratories, Technical Report, 1984.

[Brachman and Schmolze 85] Brachman, R.J., and Schmolze, J.G., "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, August 1985, 171-216.

[Brachman, Fikes, and Levesque 83] Ronald Brachman, Richard Fikes, and Hector Levesque, "KRYPTON: A Functional Approach to Knowledge Representation," *IEEE Computer*, September 1983.

[Cohen and Goldman 85] Cohen, D and Goldman N., Efficient Compilation of Virtual Database Specifications, 1985.

[Kaczmarek, Mark and Wilczynski 83] Kaczmarek, T., W. Mark, and D. Wilczynski, "The CUE Project," in *Proceedings of SoftFair*, July 1983.

[Kaczmarek, Mark, and Sondheimer 83] T. Kaczmarek, W. Mark, and N. Sondheimer, "The Consul/CUE Interface: An Integrated Interactive Environment," in *Proceedings of CHI '83 Human Factors in Computing Systems*, pp. 98-102, ACM, December 1983.

[Mark 81] William Mark, "Representation and Inference in the Consul System," in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, IJCAI, 1981.

[McAllester 82] D.A. McAllester, *Reasoning Utility Package User's Manual*, Massachusetts Institute Technology , Technical Report, April 1982.

[Neches, Swartout, and Moore 85] Robert Neches, William R. Swartout, and Johanna Moore, "Explainable (and Maintainable) Expert Systems," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 382-389, International Joint Conferences on Artificial Intelligence and American Association for Artificial Intelligence, August 1985.

[Patel-Schneider, Brachman, and Levesque 84] Peter F. Patel-Schneider, Ronald J. Brachman, and Hector J. Levesque, *ARGON: Knowledge Representation meets Information Retrieval*, Fairchild Research Laboratories, Technical Report 654, September 1984.

[Schmolze and Lipkis 83] James Schmolze and Thomas Lipkis, "Classification in the KL-ONE Knowledge Representation System," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, IJCAI, 1983.

[Sidner 85] Candace L. Sidner, "Plan parsing for intended response recognition in discourse," *Computer Intelligence* 1, 1985.

[Sondheimer 84] Norman K. Sondheimer, Ralph M. Weischedel, and Robert J. Bobrow, "Semantic Interpretation Using KL-ONE," in *Proceedings of Coling84*, pp. 101-107, Association for Computational Linguistics, July 1984.

[Tou, Williams, Fikes, Henderson and Malone 82] Tou, F.F., M.D. Williams, R, Fikes, A. Henderson, and T. Malone, "RABBIT: An Intelligent Database Assistant," in *Proceedings AAAI-82*, pp. 314-318, 1982.

[Vilain 84] Marc Vilain, *KL-TWO, A Hybrid Knowledge Representation System*, Bolt Beranak and Newman, Technical Report 5694, September 1984.

[Wilczynski 84] David Wilczynski and Norman Sondheimer, Transportability in the Consul System: Model Modularity and Acquisition, 1984.

[Wile 81] David S. Wile, *POPART: Producer of Parsers and Related Tools System Builders' Manual*, 1981.