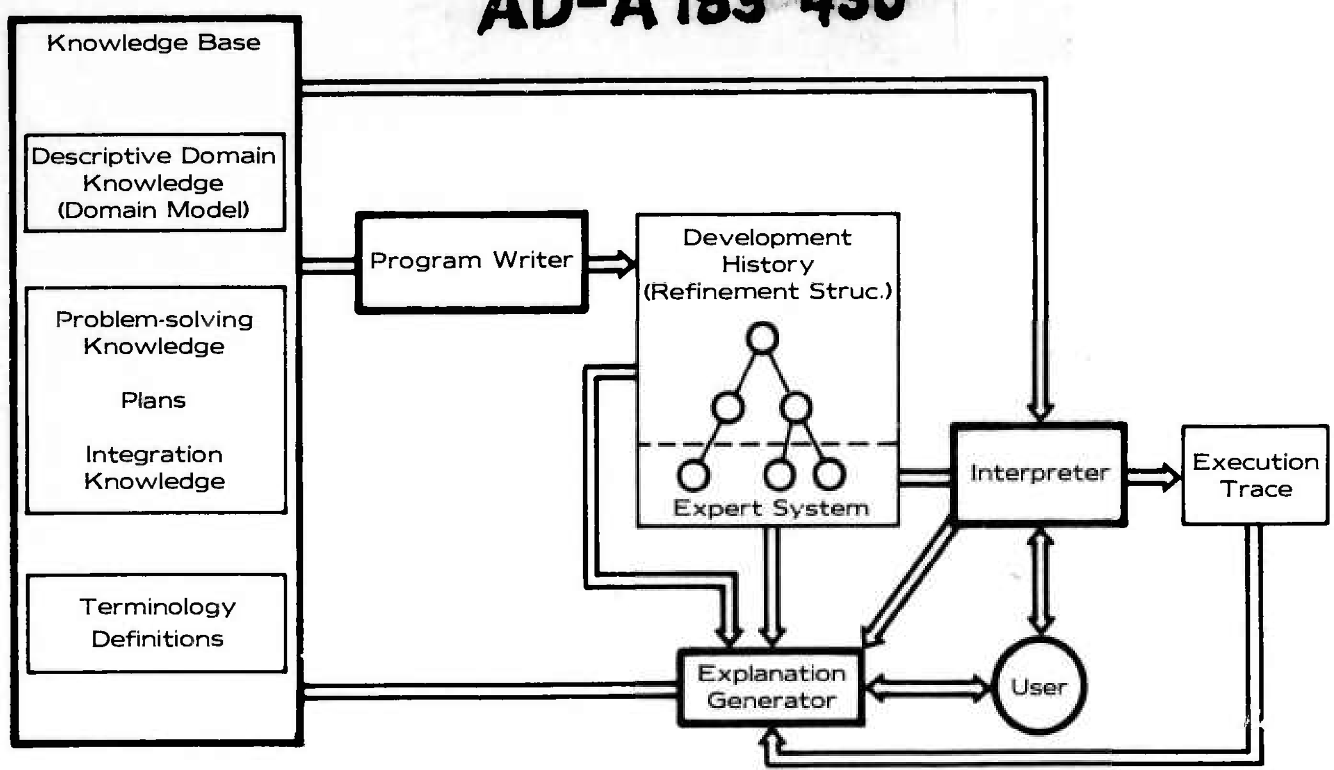
P R O C E E D I N G S

DTIC FILE COPY

Knowledge-based Systems Workshop

April 1987

AD-A183 430



Architecture for EES Version I

This document has been approved for public release and sale; its distribution is unlimited.

DTIC ELECTE AUG 13 1987 A

Sponsored by:

Defense Advanced Research Projects Agency
Information Sciences and Technology Office

87-8-13-136

Knowledge-based Systems Workshop

Proceedings of a Workshop
Held at
St. Louis, Missouri
April 21-23, 1987

Sponsored by the
Defense Advanced Research Agency

Science Applications International Corporation
Report Number SAIC-87/1069
Lee S. Baumann
Workshop Organizer

This report was supported by
The Defense Advanced Research
Projects Agency under ARPA
Order No. 5605, Contract No. N00014-86-C-0700
Monitored by the
Office of Naval Research

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.

87 8 13 136

DISCLAIMER NOTICE

THIS DOCUMENT IS THE BEST QUALITY AVAILABLE.

COPY FURNISHED CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: SAIC-87/1069
2. GOVT ACCESSION NO.: AD-183-430
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): KNOWLEDGE-BASED SYSTEMS Proceedings of a workshop, April 1987
5. TYPE OF REPORT & PERIOD COVERED: ANNUAL TECHNICAL April 1986-April 1987
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): LEE S. BAUMANN (Ed.)
8. CONTRACT OR GRANT NUMBER(s): N00014-86-C-0700
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, Virginia 22209
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA ORDER No. 5605
11. CONTROLLING OFFICE NAME AND ADDRESS:
12. REPORT DATE: April 1987
13. NUMBER OF PAGES: 192
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 Knowledge-Based 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 21-23 April 1987 in St. Louis, Missouri.

DD FORM 1473, 1 JAN 73 — EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

Page

FOREWORD i

AUTHOR INDEX iii

TECHNICAL PAPERS

Semantically Sound Inheritance for a Formally Defined .. 1
Frame Language with Defaults
    Robert Nado and Richard Fikes
    IntelliCorp

Module-Oriented Programming in ABE: Modules and 7
Abstract Datatypes
    Jay S. Lark
    Teknowledge, Inc.

Annual Report of the Experimental Knowledge Systems 20
Laboratory
    Paul R. Cohen
    University of Massachusetts

On Making Expert Systems More Like Experts 47
    William R. Swartout, Stephen W. Smoliar
    USC Information Sciences Institute

The Loom Knowledge Representation Language 59
    Robert Mac Gregor, and Raymond Bates
    USC Information Sciences Institute

A Framework for Situation Assessment: Using Best- 76
Explanation Reasoning To Infer Plans from
Behavior
    John R. Josephson
    The Ohio State University

Concurrency in Abductive Reasoning 86
    Ashok Goel, John R. Josephson, and P. Sadayappan
    The Ohio State University

An Experiment in Knowledge-Based Signal Understanding .. 93
Using Parallel Architectures
    Harold D. Brown, Eric Schoen, Bruce A. Delagi
    Stanford University

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS (Cont'd)

Page

TECHNICAL PAPERS (Cont'd)

An Instrumented Architectural Simulation System 106
Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd
Stanford University

Considerations for Multiprocessor Topologies 119
Greg Byrd and Bruce Delagi
Stanford University

RUM: A Layered Architecture for Reasoning with Uncertainty 123
Piero P. Bonissone, Steven S. Gans, Keith S. Decker
GE Corporate Research and Development Center

Progress in Reasoning with Incomplete and Uncertain Information Part I: Reasoning with Uncertainty Part II: Analogical Reasoning Part III: Reasoning with Incomplete Information 132
Piero P. Bonissone, Gilbert B. Porter, III, Allen L. Brown, Jr.
GE Corporate Research and Development Center

An Algebraic Foundation for Truth Maintenance 135
Allen L. Brown, Jr., Dale E. Gaucas, and Dan Benanav
GE Corporate Research and Development Center

Logics of Justified Belief 143
Allen L. Brown, Jr.
GE Corporate Research and Development Center

Using T-norm Based Uncertainty Calculi in a Naval Situation Assessment Application 150
Piero P. Bonissone
GE Corporate Research and Development Center

A Role for Assumption-Based and Nonmonotonic Justifications in Automating Strategic Threat Analysis 158
Dale E. Gaucas and Allen L. Brown, Jr.
GE Corporate Research and Development Center

TABLE OF CONTENTS (Cont'd)

Page

TECHNICAL PAPERS (Cont'd)

A Mathematical Theory for Diagnosis Based on the MONAD Concept 166
G.B. Porter, III
GE Corporate Research and Development Center

FOREWORD

The annual review meeting for research and government personnel involved in the DARPA program on Knowledge-Based Systems was held in St. Louis, Missouri on 21-23 April 1987. The purpose of the meeting was to review progress on research efforts undertaken over the past year. Research organizations participating in the workshop included General Electric, Stanford University, Ohio State University, University of Massachusetts, Teknowledge Inc., Intellicorp, and USC Information Sciences Institute. Also present were representatives from DARPA, AFWAL, Lockheed, McDonnell Douglas Aircraft Corporation, Texas Instruments, MIT Lincoln Laboratory, and BBN Laboratories.

On the first day of the workshop, each Principal Investigator involved in the DARPA Knowledge-Based Systems research program outlined the substance of their ongoing program, reviewed results achieved to date, and forecasted plans for future development. This proceeding is intended to document the important progress being made in the knowledge-based systems part of the DARPA Strategic Computing Program which was presented to the attendees at the workshop.

The second day of the workshop was devoted to an exchange of program details between the Knowledge-Based Systems Group and the researchers involved in the DARPA sponsored Pilots Associate Program. This reciprocal review was organized to foster synergism between technolgy base research groups and strategic computing Pilot Associate research groups. Also, on the second day of the workshop, demonstrations were given by Texas Instruments and Intellicorp. Richard Fikes and Richard Treitel presented the Intellicorp demo on their OPUS system using TUXEDO software. Their objective was to show how effectively OPUS and TUXEDO could help program managers prepare budgets so that all constraints were satisfied without leaving out any requirements and without going over budget.

Texas Instrument gave a demonstration on Wednesday of their Robust Speech Recognizer. The presentation demonstrated a connected-speech, speaker dependent recognizer operating in real time. The speaker wore a typical pilot's helment and an oxygen mask with an embedded microphone to demonstrate robustness to breath noise effects and to simulate a sense of realism. The speech system consists of a TI Explorer, a Lisp machine, an Odyssey, and a Signal Processor Board with four TMS 32020

digital processing chips. The system can be tailored to handle pilot's associate simulator applications.

The last day of the workshop consisted of a visit to the McDonnell Aircraft Combat Simulation Facility in St. Louis. The site visit was arranged and hosted by Mr. Vic Lanka, Manager of Marketing for McDonnell Aircraft. Visitors had an opportunity to view the simulation of the F15 crew station as well as the futuristic concept of an advanced cockpit.

The figure used on the cover design was provided by William R. Swartout of USC Information Sciences Institute and was extracted from his paper entitled "On Making Expert Systems More Like Experts". For additional information on this subject, see Swartout's paper included in this proceedings.

Appreciation is extended to Mr. Tom Dickerson of the SAIC Graphic Arts Department for the cover layout and to Mrs. Dianne Williams for her efforts in assisting in the conduct of the workshop and in putting together the papers comprising this proceedings.

Lee S. Baumann
Science Applications
International Corporation
Workshop Organizer

AUTHOR INDEX

NAME	PAGE
Bates, Raymond	59
Benanav, Dan	135
Bonissone, Piero P.	123, 132, 150
Brown, Allen L.	132, 135, 143, 158
Brown, Harold D.	93
Byrd, Greg	106, 119
Cohen, Paul R.	20
Decker, Keith S.	123
Delagi, Bruce A.	93, 106, 119
Fikes, Richard	1
Gans, Steven S.	123
Gaucas, Dale E.	135, 158
Goel, Ashok	86
Josephson, John R.	76, 86
Lark, Jay S.	7
Mac Gregor, Robert	59
Nado, Robert	1
Nishimura, Sayuri	106
Porter, Gilbert B.	132, 166
Sadayappan, P.	86
Saraiya, Nakul	106
Schoen, Eric	93
Smoliar, Stephen W.	47
Swartout, William R.	47

TECHNICAL PAPERS

Semantically Sound Inheritance for a Formally Defined Frame Language with Defaults

Robert Nado and Richard Fikes

IntelliCorp
1975 El Camino Real West
Mountain View, California 94040-2216

Abstract

Most frame languages either are glaringly deficient in their treatment of default information or do not represent it at all. This paper presents a formal description of a frame language that provides semantically sound facilities for representing default information and an efficient serial algorithm for inheriting default information down class-subclass and class-member hierarchies constructed in that language. We present the inheritance algorithm in two forms. In the first form, the algorithm provides justifications to a TMS, which then manages the inherited information. In the second form, the algorithm performs its own, special-purpose truth maintenance and therefore is useable in a system that does not include a general-purpose TMS.[1]

I. Introduction

The common-sense reasoning required in many knowledge system applications relies heavily on the ability to use general information that is subject to exceptions: what has been called prototypic or default information. Although frame-based representation languages have become increasingly popular for expressing the domain-specific information on which the functionality of knowledge systems is based [Fikes and Kehler, 1985], most such languages either are glaringly deficient in their treatment of default information (as argued, for example, in [Brachman, 1985] and [Touretzky, 1984]) or do not represent it at all (e.g., KL-ONE [Brachman and Schmolze, 1985] and KRYPTON [Brachman *et al.*, 1983]). Thus, an important step in the advancement of knowledge system technology is the development of a frame language that provides semantically sound facilities for representing and efficiently processing default information. This paper presents a formal description of such a frame language (based on the frame language in the KEE^{TM} system[2]) and an efficient serial algorithm for inheriting default information down class-subclass and class-member hierarchies constructed in that language. The language has been implemented at IntelliCorp in a system called OPUS.

As observed by Touretzky [Touretzky, 1986], the "shortest-path" ordering of defaults used by most inheritance systems (e.g., FRL [Roberts and Goldstein, 1977] and NETL [Fahlman, 1979]), does not always successfully provide the desired preference of more specific defaults over less specific defaults. Problems arise in some cases of multiple inheritance, where nodes are allowed to have more than one parent link. An example, adapted from Touretzky, is depicted in Figure 1. The typical inheritance algorithm correctly prefers White over Grey as a default color for a royal elephant, because the default from RoyalElephants has a "shorter path" than the default from Elephants. However, in the situation shown in the figure, Clyde has a redundant class membership link to Elephants. Clyde, then, inherits both the default White from RoyalElephants and the default Grey from Elephants *along paths of equal length*. Thus, shortest-path algorithms are not sufficient to correctly handle this situation.[3] This, and other shortcomings of existing algorithms are overcome in the OPUS algorithm presented here.

An additional motivation for this work is to enable "truth

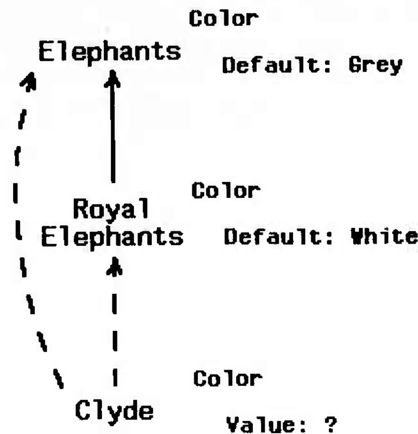


Figure 1: A Problem with the "Shortest Path" Ordering

maintenance" (or, "reason maintenance" as it is sometimes called) capabilities to be incorporated into frame-based representation systems. Truth maintenance algorithms provide an automatic means of managing derived results as changes are made in a model [Doyle, 1979]. In addition, a truth maintenance system (TMS) can be used as the basis for a context mechanism that enables a frame system to model and compare multiple hypothetical situations (as was done, for example, in the KEEworlds^{TM} facility [Morris and Nado, 1986]).

Inheritance mechanisms add derived results to a model. They also typically provide an efficient special-purpose form of truth

[1] This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. F30602 85 C 0065. The views and conclusions reported here are those of the authors and should not be construed as representing the official position or policy of DARPA or the U.S. government.

[2] KEEworlds, KEE and Knowledge Engineering Environment are trademarks of IntelliCorp.

[3] The same problem is obtained if equal numbers of intermediate subclasses are added along the two paths from Clyde to Elephants.

maintenance for those results in that they remove information they have derived when a change occurs in the form or content of the hierarchies on which those derivations are based. If a general-purpose TMS has been incorporated into a frame system, then the TMS can be used to maintain the inherited information, thereby significantly reducing the complexity of the inheritance mechanism. However, such a reduction can be obtained only if the derivations performed during inheritance are expressible in the logical formalism supported by the TMS.

The inheritance algorithm in the current KEE system (and in other similar systems) is unsuitable for providing such justifications because it depends on arbitrary LISP procedures to perform its deductions and allows those procedures to use information whose semantic interpretation is unclear such as the order in which inheritance links are stored. The OPUS inheritance algorithm we present here performs sound deductions describable to a TMS in the form of nonmonotonic justifications whose justifiers are propositions expressible in the frame language. OPUS, therefore, in combination with the KEEworlds system, performs context-relative inheritance.

After presenting the formal description of the frame language, we present the OPUS inheritance algorithm in two forms. In the first form, the algorithm provides justifications to a TMS, which then manages the inherited information. In the second form, the algorithm performs its own truth maintenance and therefore is useable in a system that does not include a TMS.

II. A Frame Language with Defaults and Exceptions

The formal description we present for the OPUS frame language is based on the formalism developed by Etherington for default and exception links in inheritance networks [Etherington, 1987]. We found it desirable to extend Etherington's formalism in several ways to accommodate the structure of a frame language, to include more powerful constructs for describing exceptions to defaults, and to provide for the overriding of defaults at superclasses by defaults at subclasses. Those extensions are noted in our presentation and the motivations for them discussed.

A. The Language To Which Defaults Were Added

We begin by providing a set-theoretic description of the OPUS frame language before defaults and exceptions were added.

1. Frames

A *frame* represents an entity in the domain of discourse. Formally, a frame corresponds to a logical constant. A frame includes a collection of *own slots* that describe binary relationships considered to hold between the entity represented by the frame and other entities in the domain. A frame's collection of own slots necessarily includes MemberOf, which represents the standard set (i.e., class) membership predicate from set theory.

2. Class Frames

A *class frame* is a frame that represents a collection (i.e., class) of entities in the domain of discourse. Such a class is itself considered to be an entity in the domain of discourse. Thus, a class frame has associated with it a collection of own slots describing the binary relationships that the class has with other entities. Those own slots include Subclass, SubclassOf, Member, and MemberOf, which represent the standard subset and set membership predicates from set theory. These slots provide the "links" over which inheritance is done. In addition, a class frame has associated with it a collection of *prototype slots* that describe binary relationships considered to hold between each member of the class represented by the frame and other entities in the domain.

3. Own Slots

An own slot has associated with it a collection of *values*, each of which represents an entity in the domain of discourse. Formally, an own slot named S has associated with it a binary predicate, which for convenience we will also call S. An own slot S in a frame F having value V corresponds to the assertion $S(F,V)$.

4. Prototype Slots

A prototype slot has associated with it a collection of *necessary values*, each of which represents an entity in the domain of discourse. Formally, a prototype slot S has associated with it a binary predicate Nec$S$. A prototype slot S in a class frame C having necessary value V corresponds to the assertion Nec$S(C,V)$. Predicate Nec$S$ is related to predicate S by the following definition:[4]

$$\text{Nec}S(C, V) \equiv \forall x\, [\text{MemberOf}(x,C) \supset S(x, V)]$$

The following theorem follows from this definition and the set theory definition of SubclassOf in terms of MemberOf:

$$\text{Nec}S(C, V) \wedge \text{SubclassOf}(x, C) \supset \text{Nec}S(x, V)$$

That is, necessary values of a prototype slot at a class frame representing a class C are also necessary values of the prototype slot at all class frames representing subsets of C. The OPUS inheritance algorithm performs the deductions implied by the definition of Nec$S$ and by the theorem by propagating necessary values of prototype slots to all subclasses and class members.

The OPUS frame language without defaults can be characterized as expressing statements of the form $S(x,y)$ and Nec$S(x,y)$ for arbitrary first order binary predicates S. The language does not recurse in that it does not represent predicates of the form NecNec$S$.

B. Adding Defaults and Exceptions

Our goal was to augment the frame language described above to enable class frames to include prototypical descriptions of class members. That is, we wanted to enable prototype slots to have *default values* that would be inherited to class members as assumed values for the corresponding own slots unless blocked by *exceptions*.

We began by attempting to directly implement the formalism for defaults with exceptions in inheritance networks described by Etherington [Etherington, 1987]. Etherington's formalism is stated entirely in terms of unary class membership predicates. That is, he treats each class C as a unary predicate, $C(x)$, that is true when x is a member of C. He defines a "Membership" link between an object a and a class C to mean a belongs to class C (i.e., $C(a)$). The OPUS MemberOf own slot corresponds to the membership link. He defines a "Strict IS-A" link between class $C1$ and class $C2$ to mean $C1$'s are always $C2$'s (i.e., $\forall x\, [C1(x) \supset C2(x)]$). The OPUS SubclassOf own slot corresponds to the strict IS-A link.

Own slots are treated in Etherington's formalism by considering each slot-value pair (S,V) to be a unary predicate, $SV(x)$, corresponding to the class of all objects having value V for own slot S (e.g., the class of objects having color grey). Given that formalism for own slots, a necessary value V of a prototype slot S in a class frame C is a strict IS-A link between C and SV.

Etherington represents default information in his inheritance networks by "Default IS-A" and "Exception" links. A default IS-A link from class $C1$ to class $C2$ means "Normally, $C1$'s are $C2$'s", and is expressed formally by the default logic inference rule:

$$\frac{C1(x) : C2(x)}{C2(x)}$$

[4] Here and in the rest of the pa[illegible] free variables are implicitly universally quantified.

The interpretation of this rule is: if $C1(x)$ (called the ***prerequisite***) is known, and $C2(x)$ (called the ***justification*** where it appears above the line) is consistent with what is known, then $C2(x)$ (called the ***consequent*** where it appears below the line) may be concluded.

An exception link has a class at its tail and a default IS-A link at its head. An exception link from class $C1$ to a default IS-A link from $C2$ to $C3$ means "$C1$'s are exceptions to $C2$'s being $C3$'s" (e.g., "Royal elephants are exceptions to elephants being grey"). Etherington provides no independent semantics for an exception link. Instead, he defines it formally as a modification to the default rule corresponding to the link being blocked. However, Doyle has suggested (as reported by Touretzky [Touretzky, 1986]) that if the justification of the defau[illegible]le corresponding to a default IS-A link contains an addition[illegible]ary predicate unique to that default, then an exception link b[illegible]king that default [illegible] be defined to correspond to an assertion of the negation of that predicate for each member of the class at the tail of the link. Following that suggestion, a default IS-A link from class $C1$ to class $C2$ would correspond to the default rule:

$$\frac{C1(x) : C2(x) \wedge \neg \text{ExceptionTo}C1C2(x)}{C2(x)}$$

and an exception link from $C3$ to the default IS-A link from $C1$ to $C2$ would correspond to the implication: $\forall x\, [C3(x) \supset \text{ExceptionTo}C1C2(x)]$.

To add Etherington's default IS-A and exception links to the OPUS frame language, we associate with each prototype slot in a class frame a set of ***default values*** and a set of ***prototype exceptions***, and we associate with each own slot in a frame a set of ***own exceptions***. Defaults consist of pairs of values and classes, where the class in the pair provides information to the inheritance mechanism indicating the class where the default originated. Prototype and own exceptions also consist of pairs of values and classes. An own exception (V,OC) for an own slot S in a frame F blocks inheritance to F of default value (V,OC) for S, and corresponds to the assertion "F is an exception to OC having S V" (e.g., "Clyde is an exception to elephants having color grey."). A prototype exception (V,OC) for a prototype slot S in a class frame C blocks inheritance to C and to all its members and subclasses of default value (V,OC) for S. Such a prototype exception corresponds to the assertion "Members of C are exceptions to OC having S V". For example, "Royal elephants are exceptions to elephants having color grey".

Formally, a default value (V,C) for a prototype slot S in the class frame C corresponds to the assertion Def$S(C,V)$, a default value V,OC for a prototype slot S in a class frame C representing a subclass of the class represented by class frame OC corresponds to the assertion SubDef$S(C,V,OC)$, an own exception (V,OC) for an own slot S in a frame F corresponds to the assertion OwnExc$S(F,V,OC)$, and a prototype exception (V,OC) for a prototype slot S in a class frame C corresponds to the assertion ProExc$S(C,V,OC)$. These predicates are related by the following definitions.

1. ProExc$S$

ProExc$S(C,V,OC)$ means there is an own exception at each member x of C blocking the inheritance of default value V from class OC to own slot S in x. For a given binary predicate S, ProExc$S$ is defined as follows:

$$\text{ProExc}S(C, V, OC) \equiv$$
$$\forall x\, [\text{MemberOf}(x, C) \supset \text{OwnExc}S(x, V, OC)]$$

As was the case for predicate Nec$S$, the definition of ProExc$S$ implies that prototype exceptions are inherited to subclasses. That is:

$$\text{ProExc}S(C, V, OC) \wedge \text{SubclassOf}(x, C)$$
$$\supset \text{ProExc}S(x, V, OC)$$

An assertion of the form ProExc$S(C,V,OC)$ corresponds in Etherington's formalism to an exception link from C to a default IS-A link from OC to SV. OwnExc$S$ statements are inferred from ProExc$S$ statements and serve, following Doyle's suggestion, to block default rules at appropriate class members.

2. Def$S$

Def$S(C,V)$ means that for each member x of C, if it is consistent to assume both that V is a value of own slot S in x and that no own exception at x blocks the inheritance of V for S from C, then it can be inferred that V is a value of own slot S in x. For a given binary predicate S, Def$S$ is defined as follows:

$$\text{Def}S(C, V) \equiv$$

$$\frac{\text{MemberOf}(x, C) : S(x,V) \wedge \neg \text{OwnExc}S(x, V, C)}{S(x, V)}$$

Def$S(C,V)$ corresponds in Etherington's formalism to a default IS-A link from C to SV.

3. SubDef$S$

The SubDef$S$ predicate is an extension to Etherington's formalism to provide for the inheritance of defaults to prototype slots in subclasses. That is, the frame language is designed so that the prototype slots at any given class frame C have all the necessary and default values to be inherited by members of C that have been asserted at C or at any of C's superclasses. For example, the class frame AfricanElephants inherits from class frame Elephants the default value grey for the color prototype slot. Etherington has nothing in his formalism corresponding to that functionality.

For a given binary predicate S, SubDef$S$ statements are inferred from Def$S$ statements by the following axiom and default rule:

$$\text{Def}S(C, V) \supset \text{SubDef}S(C, V, C)$$

$$\frac{\text{SubDef}S(C,V,OC) \wedge \text{SubclassOf}(C,OC) : \neg \text{ProExc}S(C,V,OC)}{\text{SubDef}S(C,V,OC)}$$

Defaults asserted at a class as Def$S$ statements are used to infer SubDef$S$ statements at the class and are inherited to all subclasses as SubDef$S$ statements.

C. Quantified Exceptions

Etherington's link types and the statement forms we have introduced thus far for OPUS allow exceptions to be stated for specific values from specific origin classes. In practice, however, there is a need to assert collections of exception links. For example, one typically wants to state for a given slot in a given class frame (say the color slot in RoyalElephants) that *any* default value from *any* superclass is to be blocked and replaced by a given default value. Such assertions would be second order statements in Etherington's formalism. We can express them in the OPUS formalism as first order quantified statements as follows:

$$\forall v\, \text{OwnExc}S(O, v, OC)$$
$$\forall oc\, \text{OwnExc}S(O, V, oc)$$
$$\forall v,oc\, \text{OwnExc}S(O, v, oc)$$

$$\forall v\, \text{ProExc}S(C, v, OC)$$
$$\forall oc\, [\text{SubclassOf}(C, oc) \supset \text{ProExc}S(C, V, oc)]$$
$$\forall v,oc\, [\text{SubclassOf}(C, oc) \supset \text{ProExc}S(C, v, oc)]$$

The quantification of the origin class that is supported for prototype exceptions is only to ***superclasses*** of the class to whose members the exception applies. The restriction to superclasses is meant to implement the intuition that defaults at subclasses

override defaults at superclasses. For example, a default color for royal elephants overrides a default color for elephants. Thus, we do not want a quantified prototype exception to block defaults from sibling classes and subclasses, but only from superclasses. (Although, note that the unquantified form of ProExc$S$ blocks defaults from any given class, including sibling classes and subclasses. The ability to block defaults from siblings may be useful in that it allows one to express a precedence ordering of defaults between classes even though their subclass-superclass relationship is unknown.)

As observed by Touretzky [Touretzky, 1984], the natural partial ordering of defaults in inheritance systems defined by the hierarchical structure of the inheritance graph resolves many ambiguities in an intuitive way. Touretzky introduces an "inferential distance" measure that expresses the desired natural ordering of defaults and uses that measure to filter out extensions that violate the ordering. In OPUS, that effect is obtained by the explicit quantification of exceptions over superclasses. In Touretzky's formalism, an exception always blocks a specific default value from all superclasses. Thus, unlike in OPUS, he cannot block all values from superclasses nor can he block values from a given superclass.

In summary, for any first order binary predicate S, the OPUS frame language represents statements of the following form (with their Etherington link equivalents where applicable):

$S(O, V)$	O>–Member–>SV
Nec$S(C, V)$	C>–IS.A–>SV
Def$S(C, V)$	C>–Def.IS.A–>SV
SubDef$S(C, V, OC)$	
OwnExc$S(O, V, OC)$	
$\forall v$ OwnExc$S(O, v, OC)$	
$\forall oc$ OwnExc$S(O, V, oc)$	
$\forall v, oc$ OwnExc$S(O, v, oc)$	
ProExc$S(C,V,OC)$	C>–Exc–>(OC>–Def.IS.A–>SV)
$\forall v$ ProExc$S(C, v, OC)$	
$\forall v$ [SubclassOf$(C, oc) \supset$ ProExc$S(C, V, oc)$]	
$\forall v, oc$ [SubclassOf$(C, oc) \supset$ ProExc$S(C, v, oc)$]	

The system does not recurse in that it does not represent NecNec$S$, DefNec$S$, etc.

Consider how this formalism would be used to express the situation shown in Figure 1. DefColor statements would be used at Elephants and RoyalElephants to express the two defaults, and a quantified prototype exception statement would be used at RoyalElephants to block the inheritance of default colors from all superclasses, as follows:

DefColor(Elephants,Grey)
DefColor(RoyalElephants,White)
$\forall v, oc$ [SubclassOf(RoyalElephants, oc)
$\supset$ ProExcColor(RoyalElephants, v, oc)]

III. A "Push" Inheritance Algorithm for Defaults and Exceptions

The OPUS frame language has been implemented by modifying the frame language in the KEE system. The inheritance mechanism implements the deductions defined by the definitions, axioms, and theorems given above by "pushing" necessary member slot values when they are asserted to subclasses and class members, and pushing default member slot values when they are asserted to subclasses and class members unless blocked by exceptions.

In this section we describe the algorithm in two forms, one assuming the availability of a TMS to maintain the derived results and the other not. In both cases we describe the information associated with each slot in the implementation and the operations performed by the algorithm.

A. What's In A Slot?

Each own slot in a frame has associated with it sets of values and own exceptions. Own exceptions are ordered pairs of the form (<value spec>, <origin class spec>), where <value spec> is either a value or the reserved symbol *, and <origin class spec> is either a class or the reserved symbol *. The * symbol matches any origin class or value and thereby corresponds to quantified own exceptions.

Each prototype slot in a class frame has associated with it sets of necessary values, default values, and prototype exceptions. Default values are ordered pairs of the form (<value>, <origin class>) and prototype exceptions are ordered pairs of the form (<value spec>, <origin class spec>). The * symbol in prototype exceptions matches any value or any origin class that is a superclass and thereby corresponds to the desired forms of quantified prototype exceptions.

B. Inheritance with a TMS

In order to perform inheritance using a TMS, each value or exception that is considered for a slot has an assertion (TMS node) associated with it. The assertion's formula (TMS datum) is as described in Section 2 for the different types of values and exceptions. A value or exception is added to a slot by giving its corresponding assertion a suitable justification, either a primitive justification or a justification recording some deduction external to the inheritance system. A given slot has a particular value or exception just in case the TMS assigns a positive belief status to its corresponding assertion. Demons are associated with each slot that are triggered by the TMS when an assertion concerning the slot is believed for the first time. A demon for a particular value or exception type is responsible for determining which inheritance justifications involving the newly believed assertion should be added to the TMS.

Necessary values of prototype slots are inherited to class members as values of own slots via justifications of the following form:

$$\text{Nec}S(C, V) \wedge \text{MemberOf}(Memb, C) \rightarrow S(Memb, V)$$

Necessary values of prototype slots are inherited to subclasses via justifications of the following form:

$$\text{Nec}S(C, V) \wedge \text{SubclassOf}(Csub, C) \rightarrow \text{Nec}S(Csub, V)$$

Prototype exceptions are inherited from classes to class members via justifications of the following form:

$$\text{ProExc}S(C, V, OC) \wedge \text{MemberOf}(Memb, C) \rightarrow \text{OwnExc}S(Memb, V, OC)$$

Prototype exceptions are inherited from classes to subclasses via justifications of the following form:

$$\text{ProExc}S(C, V, OC) \wedge \text{SubclassOf}(Csub, C) \rightarrow \text{ProExc}S(Csub, V, OC)$$

Default values of prototype slots are inherited to class members as values of own slots via nonmonotonic justifications of the following form:

$$\text{SubDef}S(C, V, OC) \wedge \text{MemberOf}(Memb, C) \wedge \text{OUT}[\text{OwnExc}S(Memb, V, OC)] \rightarrow S(Memb, V)$$

Note that there is no OUT justifier for $\sim S(Memb,V)$ in these justifications as the formal definition of default values requires.

Such a justifier is not needed since statements of the form $\sim S(Memb,V)$ cannot be expressed in the frame language and are therefore necessarily out.

Default values of prototype slots are inherited to subclasses via nonmonotonic justifications of the following form:

SubDef$S(C, V, OC)$ ∧ SubclassOf($Csub$, C) ∧
OUT[ProExc$S(Csub, V, OC)$]
→ SubDef$S(Csub, V, OC)$

As before, these justifications do not need to have an OUT justifier for ∼SubDef$S(Csub,V,OC)$ because statements of the form ∼SubDef$S(Csub,V,OC)$ cannot be expressed in the frame language and are therefore necessarily out.

Quantified own exceptions are used to generate instantiated own exceptions as needed to block the inheritance of default values that match the quantified form. The instantiated exceptions are produced via justifications of the following forms:

OwnExc$S(F, *, OC)$ → OwnExc$S(F, V, OC)$
OwnExc$S(F, V, *)$ → OwnExc$S(F, V, OC)$
OwnExc$S(F, *, *)$ → OwnExc$S(F, V, OC)$

Quantified prototype exceptions are not inherited. Instead, they are used to generate instantiated prototype exceptions as needed to block the inheritance of default values that match the quantified form. The instantiated exceptions are produced via justifications of the following forms:

ProExc$S(C, *, OC)$ ∧ SubclassOf(C, $Csuper$) ∧
SubDef$S(Csuper, V, OC)$ → ProExc$S(C, V, OC)$

ProExc$S(C, V, *)$ ∧ SubclassOf(C, $Csuper$) ∧
SubDef$S(Csuper, V, OC)$ → ProExc$S(C, V, OC)$

ProExc$S(C, *, *)$ ∧ SubclassOf(C, $Csuper$) ∧
SubDef$S(Csuper, V, OC)$ → ProExc$S(C, V, OC)$

1. Example

Consider the statements that would be asserted and derived by this inheritance mechanism for the example from Figure 1. The inheritance of color Grey from Elephants to RoyalElephants would be done via the following justification:

SubDefColor(Elephants,Grey,Elephants) ∧
SubclassOf(RoyalElephants,Elephants) ∧
OUT[ProExcColor(RoyalElephants,Grey,Elephants)]
→ SubDefColor(RoyalElephants,Grey,Elephants)

The inheritance of color Grey from Elephants to Clyde would be done via the following justification:

SubDefColor(Elephants,Grey,Elephants) ∧
MemberOf(Clyde,Elephants) ∧
OUT[OwnExcColor(Clyde,Grey,Elephants)]
→ Color(Clyde,Grey)

The generation of the instantiated prototype exception for Grey at RoyalElephants would be done via the following justification:

ProExcColor(RoyalElephants,*,*) ∧
SubclassOf(RoyalElephants,Elephants) ∧
SubDefColor(Elephants,Grey,Elephants)
→ ProExcColor(RoyalElephants,Grey,Elephants)

The instantiated prototype exception for Grey at RoyalElephants prevents inheritance of Grey as a default to RoyalElephants. Thus, no justification is generated for inheriting Grey from RoyalElephants to Clyde. Inheritance of the instantiated prototype exception for Grey at RoyalElephants to Clyde would be done via the following justification:

ProExcColor(RoyalElephants,Grey,Elephants) ∧
MemberOf(Clyde,RoyalElephants)
→ OwnExcColor(Clyde,Grey,Elephants)

That inherited exception would block the inheritance of Grey to Clyde.

The inheritance of color White from RoyalElephants to Clyde would be done via the following justification:

SubDefColor(RoyalElephants,White,RoyalElephants) ∧
MemberOf(Clyde,RoyalElephants) ∧
OUT[OwnExcColor(Clyde,White,RoyalElephants)]
→ Color(Clyde,White)

Since there is no exception at Clyde blocking the inheritance of White from RoyalElephants, White will become the color of Clyde.

C. Inheritance Without a TMS

The above inheritance scheme relies on a TMS to remove inherited values when the assertions on which the inheritance was based are removed. For example, if the default color for elephants is removed, then the TMS will also remove Clyde's color if it was in the model only because of the default. Inheritance without the services of a TMS is considerably more complex since the inheritance machinery must, in effect, provide a truth maintenance capability for inherited values.

In order to provide for the removal of inherited values, the OPUS inheritance machinery requires each slot to have both a *local* and a *resultant* set of values and exceptions. The local sets are used only by the inheritance algorithm and contain those values or exceptions that are either asserted or are determined by some means other than inheritance. Resultant sets contain all the values and exceptions, including the local ones and those derived by inheritance. When a value or exception is to be added to (or removed from) a slot, it is added to (or removed from) the appropriate local set and the inheritance machinery recomputes the affected resultant sets for that slot. When the values of the MemberOf (or SubclassOf) own slot of a frame are modified, the inheritance machinery recomputes the resultant sets of each own slot (or prototype slot) of the frame. When a resultant set of a prototype slot is modified, affected resultant sets of all its descendants in the inheritance graph are recomputed. In the paragraphs below, we describe how each type of resultant set is computed. References in the descriptions to values and exceptions are to the resultant sets unless explicitly indicated otherwise.

The set of resultant necessary values for a prototype slot S in a class frame C is the union of the local set of necessary values for S in C and, for each $Csuper$ that is a value of the own slot SubclassOf in C, the set of necessary values for prototype slot S in $Csuper$.

The set of resultant default values for a prototype slot S in a class frame C consists of the local default values for S in C and, for each $Csuper$ that is a value of the own slot SubclassOf in C, the default values for prototype slot S in $Csuper$ that do not match an exception for S in C.

The set of resultant values for an own slot S at a frame F consists of the local values for S at F and, for each C that is a value of the own slot MemberOf in F, the necessary values for prototype slot S in C and the default values for prototype slot S in C that do not match an own exception for S in F.

The set of resultant exceptions for an own slot S in a frame F is the union of the local set of exceptions for S in F and, for each C that is a value of the own slot MemberOf in F, the set of exceptions for prototype slot S in C.

The set of resultant exceptions for a prototype slot S in a class frame C consists of the local instantiated exceptions for S in C, for each $Csuper$ that is a value of the own slot SubclassOf in C, the exceptions for prototype slot S in $Csuper$, and each $(V, Csuper2)$ that matches a local quantified exception for S in C and is a default value for some $Csuper1$ that is a value of the own slot SubclassOf in C.

Note that quantified exceptions remain in the local set and are not inherited. Quantified exceptions produce instantiated exceptions as needed to block defaults that would otherwise be

inherited.

1. Example

Figure 2 shows the local and resultant values and exceptions produced by the inheritance algorithm for our elephants example. The default (Grey,Elephants) at Elephants and the quantified exception (*,*) at RoyalElephants would cause an instantiated exception (Grey,Elephants) to be generated at RoyalElephants. That instantiated exception would be inherited to Clyde. The exception at Clyde would block inheritance of the (Grey,Elephants) default from Elephants. The default (White,RoyalElephants) at RoyalElephants would be inherited to Clyde as Clyde's color.

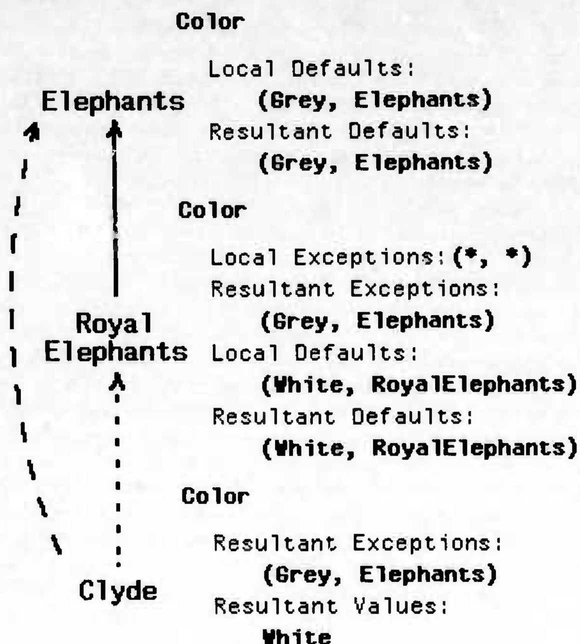


Figure 2: Inheritance without a TMS

IV. Conclusion

We have presented a formal description of a frame language that makes a clear distinction between necessary and default values of prototype slots. The formalization is based on previous work by Etherington, but extends his formalism to more closely match the structure of frame languages and to allow more convenient overriding of defaults at superclasses by defaults at subclasses.

We have presented two distinct methods for implementing the inferences warranted by the formal description of the frame language. The first makes use of nonmonotonic justifications in a TMS to record inferences corresponding to default inheritance. This method is suitable for situations in which a TMS is needed in order to maintain conclusions derived from non-inheritance inferences or to implement context-relative inheritance. The second method, in effect, implements a more efficient, special purpose truth maintenance algorithm in order to maintain the validity of inherited values. It is appropriate for situations in which a general purpose TMS is not needed.

A topic of current investigation is how to combine the two methods into a single system in which the special-purpose algorithm is used whenever possible. In many applications, general knowledge about the relationships among classes of objects in the domain and default values of prototype slots is entered directly by the domain expert and does not vary during the course of problem solving. The membership of individuals in classes and the values of own slots are more likely to be inferred during problem solving and to vary with hypothetical context. These considerations suggest that the special purpose algorithm can be used for maintaining inherited values in the upper regions of a taxonomy, with the TMS method being used as appropriate in the lower, more problem-dependent regions.

References

[Brachman, 1985] Brachman, R.J.
"I Lied about the Trees" Or, Defaults and Definitions in Knowledge Representation.
AI Magazine 6(3):80-93, 1985.

[Brachman *et al.*, 1983] Brachman, R.J., Fikes, R.E., and H.J. Levesque.
KRYPTON: A Functional Approach to Knowledge Representation.
Computer 16(10):67-74, 1983.

[Brachman and Schmolze, 1985] Brachman, R.J., and J.G. Schmolze.
An Overview of the KL-ONE Knowledge Representation System.
Cognitive Science 9(2):171-216, 1985.

[Doyle, 1979] Doyle, J.
A Truth Maintenance System.
Artificial Intelligence 12(3), 1979.

[Etherington, 1987] Etherington, D.W.
Formalizing Nonmonotonic Reasoning Systems.
Artificial Intelligence 31:41-85, 1987.

[Fahlman, 1979] Fahlman, S.E.
NETL: A System for Representing and Using Real-World Knowledge.
MIT Press, Cambridge, Massachusetts, 1979.

[Fikes and Kehler, 1985] Fikes, R. and T. Kehler.
The Role of Frame-Based Representation in Reasoning.
Communications of the ACM 28(9):904-920, 1985.

[Morris and Nado, 1986] Morris P.H., and R.A. Nado.
Representing Actions with an Assumption-Based Truth Maintenance System.
In *Proceedings AAAI-86.* Philadelphia, 1986.

[Roberts and Goldstein, 1977] Roberts, R.B. and I.P. Goldstein.
The FRL Manual.
MIT AI Memo 409, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, September, 1977.

[Touretzky, 1984] Touretzky, D.W.
Implicit Ordering of Defaults in Inheritance Systems.
In *Proceedings AAAI-84*, pages 322-325. Austin, Texas, 1984.

[Touretzky, 1986] Touretzky, D.W.
The Mathematics of Inheritance Systems.
Morgan Kaufmann, Los Altos, California, 1986.

Module-Oriented Programming in ABE: Modules and Abstract Datatypes[1]

Jay S. Lark

Teknowledge, Inc.,
Palo Alto, CA 94303

Abstract

Intelligent Systems Engineering is characterized by the need to support large-scale applications, the reuse of software modules and capabilities, intelligibility of both a system's definition and its operations, and the integration of a system with its external hardware and software environment. ABE is a new-generation software architecture that supports the process of building Intelligent Systems.

We describe our Intelligent Systems Engineering methodology, and how features of ABE support it. Focusing on two important aspects of our methodology, we show how to define primitive modules and abstract datatypes. In particular, we examine the importation of foreign code and data structures as modules and abstract datatypes.

1. Introduction

In April 1986, Teknowledge, Inc. demonstrated a preliminary version of ABE, a new generation software architecture for building intelligent systems [Erman 86]. In the year following, we have worked both on developing and refining the software itself, and on clarifying the problems that ABE addresses. This report addresses both of these areas.

Section 2 describes our Intelligent Systems Engineering methodology, describing in detail the important properties of intelligent systems, requirements for an Intelligent Systems Development Environment and ABE's methodology for the design of intelligent system. We then focus on two important aspects of building intelligent systems in ABE: Section 3 describes how to build primitive modules, particularly ones based on imported code, while Section 4 describes ABE's abstract datatype mechanism. The rest of this section presents a brief review of the Module-Oriented Programming methodology that ABE implements, and finally describes the application we use as the source of the examples for this report.

1.1. Review of Module-Oriented Programming

ABE is a multi-level software architecture for building intelligent systems. It implements a programming methodology known as *Module-Oriented Programming* (MOP). In its simplest form, MOP specifies that systems are composed of a number of *modules* which communicate with each other. Modules can be either *primitive*, with no internal structure visible to ABE, or *composite*, where their internal structure consists of a number of other ABE modules. By convention, modules communicate with each other by passing structured data in the form of abstract datatypes (ADTs).

Primitive modules are built in one of a number of supported *languages*. Currently, ABE supports many different languages such as Common LISP, CORAL[2], KEE[3], Knowledge Craft[4], and MRS.

Frameworks are special-purpose languages that provide the means to compose modules. Each framework implements a particular programming metaphor, such as blackboards or dataflow, by providing an interpreter which controls the execution of and communication among the modules composed within that framework.

---

[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, CORAL, Module Oriented-Programming, and MOP are trademarks of Teknowledge, Inc.

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.

[2]CORAL is an object-oriented programming language built on top of Common LISP as part of the ABE project

[3]KEE is a trademark of IntelliCorp.

[4]Knowledge Craft is a trademark of Carnegie Group Inc.

MOP requires all modules to have the same external form. Modules are described by their input/output behavior, not by the particular language or framework used to construct them. This property allows a module builder to implement the internals of a module using the most appropriate language or framework for that module, without having to worry about how it will be used. In particular, a composite module builder can change the framework used for the composite module but continue to use the same component modules.

MOP defines several layers on top of the basic modules and frameworks. *Skeletal systems* are configurations of modules that define a problem-solving structure, such as a generic replanning system. The addition of a domain-specific *customization* to a skeletal system results in a full-fledged *application*. For a more complete discussion of MOP see [Erman 86].

1.2. AADS -- An Example Application

The remainder of this report uses examples from a demonstration system we built called AADS, or the ABE Anti-Submarine-Warfare Demonstration System [Hollander 86]. We developed the system to demonstrate ABE applicability in the Navy Battle Management arena, particularly for the CASES (Capabilities Assessment) system [SPAWARSYSCOM 85]. While AADS addresses only a small subset of the CASES problem in a very superficial way, it does illustrate some important features of ABE.

The AADS system models an idealized anti-submarine-warfare (ASW) campaign. In this model, an ASW campaign consists of four stages: underwater mines, patrol aircraft, independent submarines, and aircraft carrier battle groups. Each stage operates more or less independently, and the total effectiveness, in terms of percentage of enemy submarines destroyed, is the product of each stage's individual effectiveness.

We have built a model of the ASW campaign, which takes as inputs force levels for the various stages (*e.g.*, number of mines, duration of aircraft patrol, *etc.*), and produces as outputs a measure of the overall effectiveness and a relative cost measure. Separate from the ASW model, we have implemented a simple hill-climbing search routine (the Satisficer) that, given an initial configuration of force levels and a utility function, searches for a new set of force levels that optimizes that utility function. The utility function consists of a number of semi-independent preference measures, such as expressing preferences for or against any specific stage or setting a maximum cost target. Finally, we have built a number of special-purpose graphical and tabular interfaces to the system, which allow the user to interact in a spreadsheet-like manner with the system.

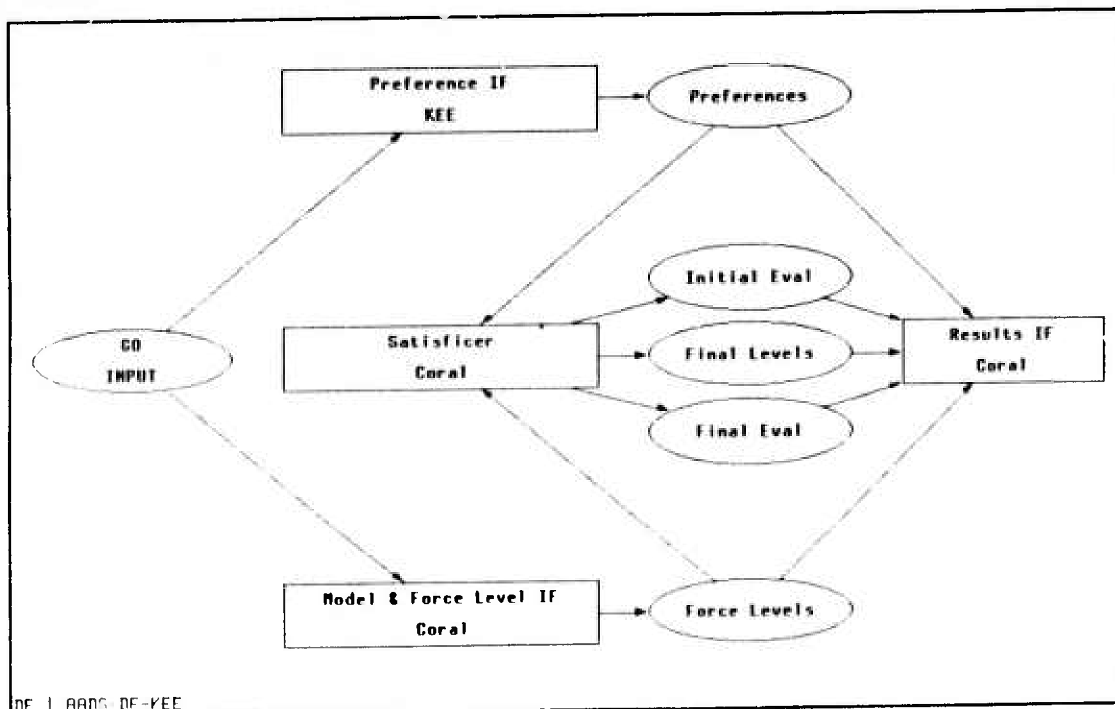


Figure 1-1: Top-level view of ABE/ASW Demonstration System (AADS)

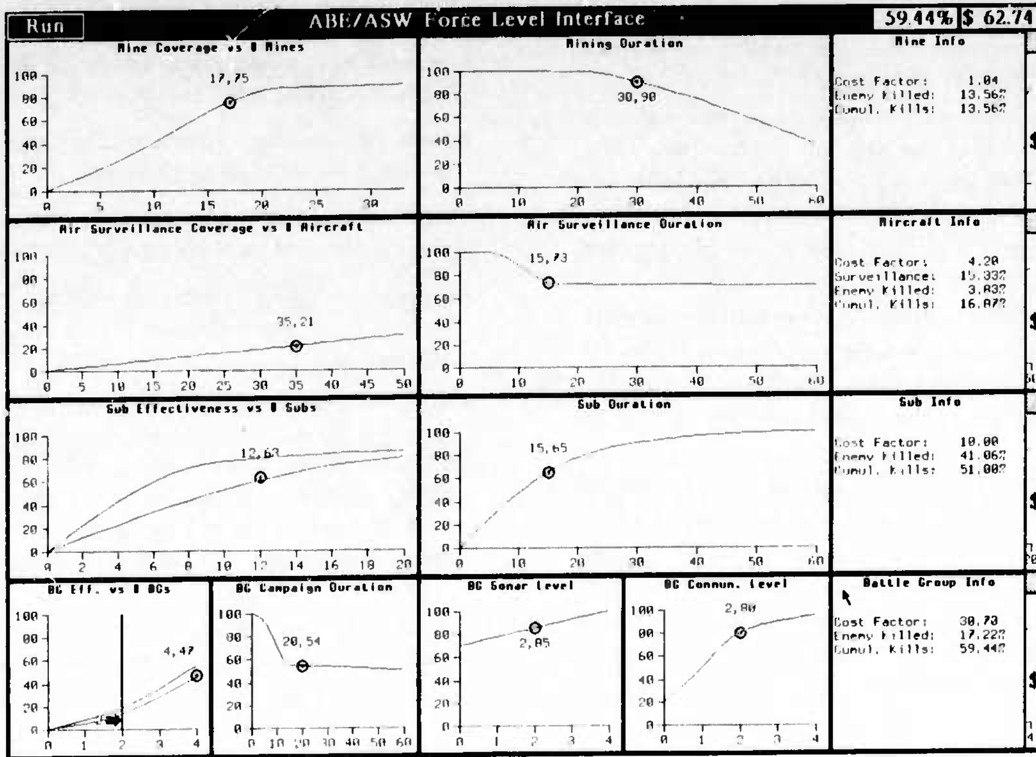


Figure 1-2: AADS ASW Model interface

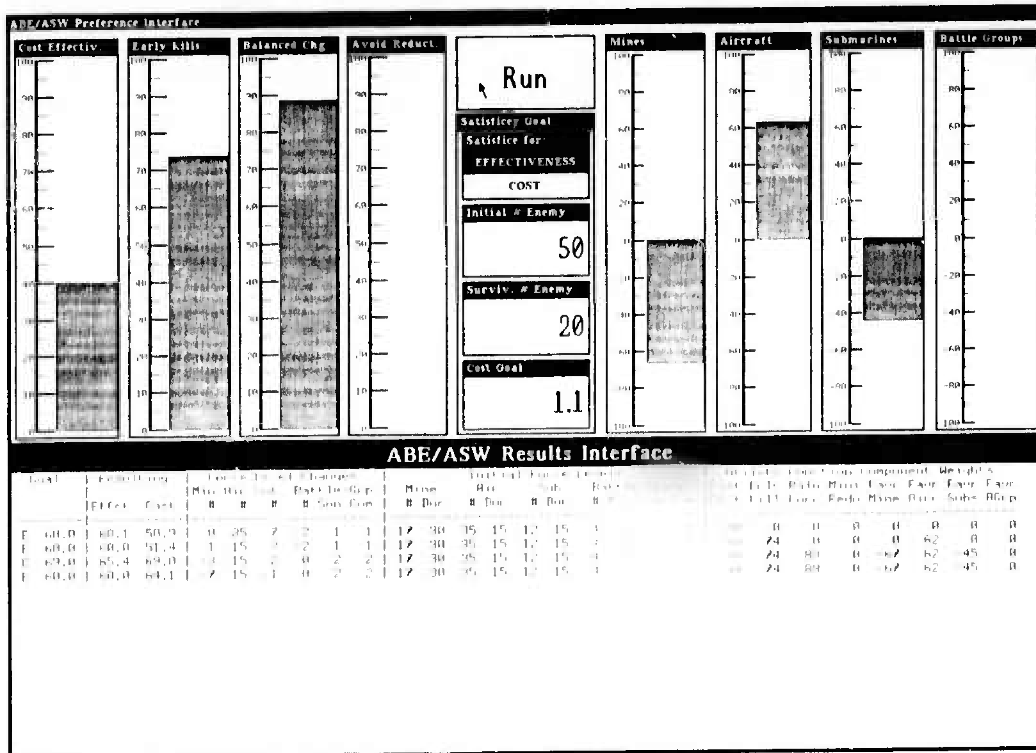


Figure 1-3: AADS Preference and Results interfaces

Figure 1-1 shows a view of the main AADS modules, composed using ABE's DF dataflow framework. The rectangular boxes represent modules, while the ellipses represent places that can hold ADTs Figure 1-2 shows the interface to the AADS ASW model, which consists of a number of graphs that the user can manipulate, along with various derived attributes such as cost and effectiveness. and Figure 1-3 shows the force preferences interface and the results table. The force preference interface consists of a number of gauges that control the various aspects of the utility function. The results table displays one row for each run of the Satisficer, showing the initial force levels, the utility function coefficients, and the recommended changes to the initial force levels.

AADS is an initial version of a decision-support tool for planning ASW campaigns. It allows a user to select an initial force preference configuration and determine its cost and effectiveness. It also allows a user to search for a "best" configuration, where the user can define his own version of "best". We have constructed AADS as a testbed and demonstration for ABE. In particular, we have demonstrated the ability to design a system using multiple, independent frameworks for combining modules, to use both hierarchical and non-hierarchical communication among the modules, and to interface to modules implemented in external languages such as KEE and Knowledge Craft.

2. Intelligent Systems Engineering

This section describes our view of the Intelligent Systems Engineering process. We first define intelligent systems and intelligent systems development environments, and then describe our system development methodology, concluding with a discussion of ABE features that support that methodology.

2.1. Characteristics of Intelligent Systems

Intelligent systems comprise both knowledge-based and conventional software components, and function in an integrated fashion with their surrounding environment. In contrast to conventional expert or knowledge systems, intelligent systems do not have a single "knowledge" component with which the rest of the system interacts. Instead, intelligent systems contain many modular capabilities, each of which may contain knowledge-based components.

An *intelligent system development environment* (ISDE) is a software environment that supports the construction of intelligent systems. ISDEs support the process of building intelligent systems by providing

- an architecture and methodology for describing and building intelligent systems;
- tools for describing system requirements and designs from multiple perspectives;
- interactive programming environment(s), including interpreters, debuggers, and browsers;
- support for the evolution of systems, especially including performance tuning and compilation; and
- tools for managing the complexity of the software development process, including tools for abstracting module and system descriptions, automatic cataloging and retrieval of modules, and source and configuration control across multiple machines and developers.

The following paragraphs describe the important characteristics of intelligent systems and present some detailed requirements for an ISDE. We also indicate some of the features that ABE provides that satisfy these requirements.

2.1.1. Scale

Intelligent systems address problems of a very large scale. Large-scale can refer to many different measures of a system:

- the size of the knowledge or databases the system uses;
- the number and variety of different functions the system must perform;
- the number of different components that comprise the system; and
- the complexity of the interactions among components.

ISDEs require tools and methodologies for dealing with the increased complexities of these massive systems. ABE supports hierarchical structuring of systems with associated graphical inspection tools which allow one to manage system complexity. In Phase 2 we will provide additional facilities to address the problem of scale, including integrating a commercial relational DBMS with ABE and developing techniques for arbitrarily compiling away unnecessary structure to support high-performance applications. For instance, we will support the compilation of hierarchically-structured systems described above.

2.1.2. Reusability

The large-scale systems described above cannot in general be built from scratch. System developers will need to reuse existing code, algorithms, and architectures in order to capture poorly-understood techniques, increase the efficiency of the system development process, and produce well-tested and reliable systems. ISDEs must provide a standard architecture that both supports the reuse of existing code and provide for newly-developed code to be reused in later applications. This same architecture can support the evolutionary development of an intelligent system, which can be viewed as the reuse of the subcomponents through different phases of the system development lifecycle.

By supporting the definition of standard interfaces, ABE allows a module builder to create modules that can be cataloged and reused in new applications. Over time, the catalog will grow to contain modules of proven utility, reliability, and performance.

2.1.3. Intelligibility

By intelligibility we refer to two distinct capabilities.

- In classical knowledge engineering terms, a system should have the capability to make its actions and reasoning understandable to the current task and user.
- From a software engineering viewpoint, a system should have a clear and understandable system definition, to support maintenance and evolution.

ABE provides the structure to incorporate the best ideas in system explanation, such as those coming out of the Strategic Computing Initiative Knowledge-Based Systems program [Chandrasekaran 86]. We also have concentrated on making the system definitions open and intelligible through the use of special-purpose graphics, browsers, and inspectors.

2.1.4. Integration

Finally, intelligent systems should support many different kinds of integration: integrating knowledge-based and conventional components, integrating components written in different languages, and integrating systems across a distributed heterogeneous computing environment. ISDEs must provide support for specifying standard interfaces to software components.

We have already demonstrated the integration of conventional components with knowledge-based components, and the integration of components written in different languages. In Phase 2 we will build a distributed processing infrastructure on top of the MACH distributed operating system [Rashid 86], and upgrade many ABE frameworks to work in a distributed environment.

2.2. System Design Methodologies

ABE takes a hybrid view of the Intelligent Systems Engineering process. We see system-building happening in both a top-down and bottom-up fashion simultaneously, with explicit support for evolving a system design as system requirements or problem understanding change. The top-down component consists of analyzing a system's function(s), and allocating those functions to various functional modules. The design at this stage also requires some notion of control and communication among these modules. ABE frameworks embody particular control and communication design decisions, and the system designer selects a framework appropriate to the task and modules at hand. For instance, if the system designer knows that the functional modules will want to pass control and communicate data in a simple dataflow manner, the designer would select the DF (dataflow) framework to describe the modules' interactions.

Given a decomposition of the system into a number of independent modules, the system designer can apply the same decomposition process recursively to the component modules. Control and communication design decisions made at higher levels have no effects on decisions at lower levels. This property allows the system designer to select the most appropriate ABE framework to implement any module. This recursive decomposition continues until the designer reaches primitive modules -- either existing modules from the ABE catalog, or modules that will be built from scratch in a programming language such as LISP, C, or KEE.

We have intentionally not committed to a specific decomposition methodology, such as SADT. We believe that over time each framework will acquire its own design methodology or, conversely, frameworks will be built to support standard methodologies. For instance, we are investigating the use of the blackboard metaphor [Erman 80] as a system design technique, with the BBD framework providing an environment for immediately testing new designs.

In parallel with the top-down decomposition process, ABE also supports a bottom-up synthesis process.

Starting with existing modules in the ABE catalog, system builders can compose new modules with higher-level functionalities. These new modules can themselves be placed in the catalog, for reuse in still higher-order modules.

ABE supports a prototyping approach to module synthesis. System designers can rapidly mock-up systems in an executable design framework, which provides for immediate feedback to the designer. ABE provides the following features which support this prototyping activity:

- Interactive interpreters, editors, and debuggers for each of the system design frameworks.
- Interactive catalog browsers, for searching a library of existing modules to find one appropriate to a specific application.
- Support for module stubs and delayed binding of modules and datatypes, which allows a designer to focus on one level at a time.
- Ability to change frameworks as the system designer comes to understand the module interactions better, without having to change the modules that comprise the system.
- Ability to save partial modules in the catalog where they can be retrieved later for additional refinement, or used as components in other systems.

The bottom-up approach combined with the top-down approach described above together permit system designers and builders to converge rapidly on a working prototype system. ABE provides or will provide support for the evolution of that prototype to a full operational system:

- Many of the features listed above for prototyping, such as changing frameworks.
- Provisions for adding non-hierarchical communication between modules.
- Advanced compilation techniques to collapse embedded systems and remove levels of interpretation.

3. Modules

This section briefly describes how a module builder implements primitive and composite modules in ABE. We first look at primitive modules, focusing on the specification of the I/O behavior. We then describe how to import foreign capabilities as modules, and list some of the problems to expect in importing foreign code and methods to minimize their impact.

3.1. Primitive Modules

ABE modules can be either primitive or composite. Primitive modules have no ABE-defined internal structure. They form the lowest level building blocks of ABE systems. The behavior of a primitive module is supplied by a piece of code written in one of the ABE-supported programming languages, such as CORAL, Common LISP, or KEE. When the programmer supplies that code with the rest of the module definition we refer to that module as a *black-box module*. When the bulk of that code is supplied by an existing piece of foreign code, we refer to the module as an *importer module*. The distinction exists only as an annotation to users to indicate that a module imports some external code; ABE does not distinguish these two kinds of primitive modules.

A primitive module specification has four main parts: a definition of the I/O behavior of the module, a definition of the function the module computes, definitions of other operations, and documentation and other annotations. The I/O specification consists of descriptions of a number of ports. Modules receive and transmit data through ports. Ports can be designated as input, output, or bidirectional, and can handle data synchronously or asynchronously with respect to the execution of the module. In addition, the module builder can attach type information to a port, indicating the allowable datatypes that can flow through that port.

Figure 3-1 shows the definition of the AADS force preference interface module. The `:IN` and `:OUT` arguments define the input and output arguments of the module, respectively. Note that the output argument `preferences` has an explicit type declaration.

The function definition of a primitive module, also known as its body, consists of a declaration of a set of local state variables and a procedure definition. ABE supports both dynamic local variables and persistent local variables, *i.e.*, variables that retain their state between invocations. The procedure definition determines the behavior of the module. It can be written in any ABE-supported language, *e.g.*, Common LISP. It can freely refer to any local variables, and can also access and store values in ports.

Referring back to Figure 3-1, the `:IVARS` argument defines a single persistent local variable `Frame`, which

points to the preference interface window. The :EXECUTE argument specifies the procedure to invoke when this module is called. The call to CREATE-ADT creates an abstract datatype, using slot values from the user-defined (k::preferences k::kee-asw) unit to serve as initial values for the ADT's slots. Finally, the RETURN-OUTPUT function returns this new ADT as the value of the preferences output. See Section 4 for more details on ADTs.

Modules can have more than one primary function. For example, a module may perform widely different operations depending on which input ports have data when the module is invoked. This case occurs often when a single module instance occurs in more than one framework simultaneously. We are investigating various alternatives for describing the different functions a module may perform.

A module must support other operations besides its primary function(s). ABE includes provisions for initializing the persistent state of a module, and for resetting part of that state. We call the initialization operation *customization.* Our intuitive definition of module customization is the specialization of a generic module, especially for a specific application domain. The module builder must supply a procedure for customizing a module for each specific domain the module will operate in. The customization operations may consist of preloading a database, defining domain-specific terms and operators, configuring special-purpose user interfaces, or other arbitrary operations.

ABE also defines two reset operations. A "soft" reset clears any execution-specific data in a module, to prepare it for operating on a new set of input data. A "hard" reset performs all the operations of a soft reset, and also clears any customizations in effect. If appropriate, a module builder may supply procedures to implement these operations.

Figure 3-2 shows the definition of the hard reset code for the preference interface module. This code will create a new interface (KEE desktop and windows) when invoked by calling the user-defined function LOAD-KEE-INTERFACE.

```
(defmodule ASW-PREFERENCE-INTERFACE-KEE
  (:IN go)
  (:OUT (preferences (:type asw-preferences)))

  (:IVARS
    (Frame
      :settable (:default-init nil)
      (:documentation "The KEE desktop that holds the ActiveImages")))
  (:EXECUTE
    (declare (ignore go))
    "Return the preferences specified by the user's menu settings."
    (RETURN-OUTPUT
      :preferences
      (CREATE-ADT 'asw-preferences-kee
        :kee-unit (k:unit '(k::preferences k::kee-asw)))))

  (:DESCRIPTION "The ABE/ASW Preference Interface - KEE version.")
  (:DOCUMENTATION
    "The preference interface implemented by a set of KEE
     ActiveImages. The user can freely interact and modify the values
     of the preference gauges. When called, this module looks at the
     values in the gauges and returns an abstract datatype that contains
     all those values."))
```

Figure 3-1: Definition of the AADS Preference Interface module

```
(defresponse (asw-preference-interface-kee :RESET)
  "Create a new KEE desktop and actuators."
  (setq frame (load-kee-interface))
  (send frame :bury))
```

Figure 3-2: Reset Code for the AADS Preference Interface module

Module builders can also specify annotations and documentation for a primitive module. Current annotations include:

- A one-line documentation string, which gives a brief description of the function of the module.
- A multi-line documentation string, which describes the operation of the module in detail.
- A graphical name and associated formatting information, which determine how ABE formats the name of the module on the computer display.
- A graphical icon, which provides a quick visual indication as to the identity, function, or operation of the module.

The `:DESCRIPTION` and `:DOCUMENTATION` arguments in Figure 3-1 show the short and long documentation for the force preference module.

3.2. Importer Modules

As described above, an importer module is a type of primitive module in which a piece of foreign code provides some functionality that, together with user-written code, defines the behavior of the module. We assume that the module builder is unlikely or unable to make modifications to the foreign code, *e.g.*, he may not have access to the source code. However, we do assume the module builder has a set of documented entry points into the code.

The basic steps needed to import foreign code into ABE are:

1. Identify the functions required of the foreign code, and the entry points that provide that functionality.
2. Determine the input and output data requirements for those entry points, *i.e.*, what arguments does the foreign code expect, and what values does it return. This may require the definition of new ADTs to interface to internal data structures in the foreign code.
3. Define an importer module to act as a transducer for the entry points by implementing foreign code calling conventions, converting data, and specializing the module to run in the ABE environment.

The module builder must also consider the following properties of the foreign code and the effect they will have on any ABE modules that include the imported code:

- reentrancy of the foreign code;
- user interface to the foreign code; and
- unintended interactions with other modules, or with the rest of the native computing environment.

The result of the importation process is a black-box module which can then be used like any other ABE module. The rest of this section describes three importation process in more detail.

3.2.1. Identifying Entry Points

As stated above, each module generally performs a single function or a set of related functions which vary according to the module's input data. The module builder must first identify the functions expected of the foreign code. Each function should specify an operation to be performed on a set of input data, resulting in a set of output data. In some cases, the function may not consume any inputs or may not produce any outputs. The specification should also state whether the data will be synchronous or asynchronous with respect to the invocation of and return from the foreign code.

The module builder must next determine the entry points that implement each desired function. Ideally, there exists a one-to-one mapping from functions to entry points. In cases in which this condition does not hold, the module builder can apply a number of techniques. The first calls for the module builder to look for a number of entry points that each provide a piece of the desired functionality, and then combine them into a single importer module. Note that the aggregation of the entries can provide extra functionality beyond that desired; the importer module can ignore the extraneous functionality.

If no combination of existing entries supplies the desired functionality, the module builder can try to create a new entry point. This generally requires access to the sources for the foreign code and an understanding of the internal structure and behavior of the foreign code.

A third alternative applies if an entry point supplies a critical subfunction. In this case, the module builder can

build the missing functionality into the importer module, or can define a composite module that includes the foreign code as one module, along with other ABE modules to supply the missing functionality. If this step fails, the module builder must reexamine the intended use of the foreign module.

3.2.2. Input/Output Requirements

The module builder must now identify the data requirements for the new module. This involves determining the content of input and output data, as well as the timing requirements for that data.

ABE modules communicate with each other through ADTs. ABE provides mechanisms for defining both the logical and physical structure of ADTs. Modules uniformly access and modify ADTs by sending them messages. See Section 4 for details.

In order to import foreign code, the module builder must identify or define ADTs that support the I/O requirements of the importer module. Usually, external considerations dictate the logical structure of these ADTs, *e.g.*, the importer module has to interface with other modules with pre-established I/O requirements.

Another point to consider in defining ADTs is their physical structure. The module builder can specify that a data structure in the foreign code provides the physical definition of an ADT. This requires knowledge of and access to the foreign data structures, but can simplify the job of the importer module in doing data conversions, increasing the program's efficiency at the same time. Section 4 describes this process in more detail.

Figure 3-3 shows the definition of the AADS force preferences abstract datatype. The `:IMPLEMENTATION` argument specifies the name of an ABE-supplied ADT storage method; this ADT will interface with KEE and access values inside KEE Units automatically. This interface frees the force preference module shown in Figure 3-1 from worrying about data formats inside KEE.

In addition to the structure of the data, the module builder must determine the timing characteristics of the data. Generally, a module receives all of its input data when invoked and returns all of its output data when it returns. We refer to this method of passing data as *synchronous*. ABE also supports *asynchronous* data exchange, where an already executing module can send messages to other modules to send or receive data. The AADS Satisficer module interacts in this manner with the ASW Model module, using the TX transaction framework to specify their interactions. Figure 3-4 shows this TX-defined composite module, with the ASW Model module acting as a server for the AADS Satisficer module.

```
(defadt ASW-PREFERENCES-KEE
  "Define an ADT that can access the values
   of the preference settings."
  (:IMPLEMENTATION :kee-copy)
  (:SLOTS
    (k::Cost-Or-Effectiveness
      (:to-read :driving-goal))
    (k::Initial-Enemy-Subs
      (:to-read :number-subs))
    (k::Acceptable-Surviving-Subs
      (:to-read :surviving-subs))
    (k::Cost-Goal
      (:to-read :raw-cost-goal))
    (k::Cost-Effectiveness)
    (k::Early-Kills)
    (k::Balanced-Changes)
    (k::Minimal-Changes)
    (k::Mines)
    (k::Aircraft)
    (k::Submarines)
    (k::Battle-Groups)))
```

Figure 3-3: Definition of Preferences abstract datatype

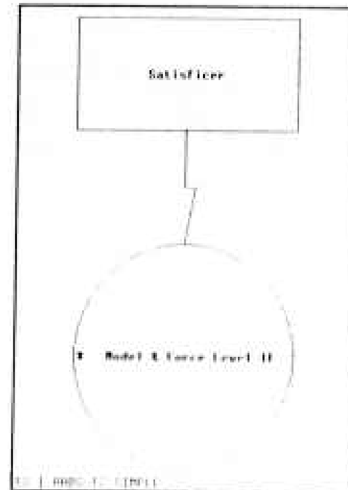


Figure 3-4: Transaction connection between AADS Satisficer and Model modules

3.3. Defining the Importer Module

The module builder can now define the importer module. As described earlier, an importer module is just a special case of a primitive module. The module body of an importer aggregates and sequences foreign code entry points, implements any necessary data transformations, and passes data into and out of the foreign code.

The module builder must also pay attention to several other subtle issues which can complicate the importation process. These generally involve unforseen interactions between the imported code and the rest of the ABE environment.

3.3.1. Reentrancy

The ABE model specifies that a number of instances of a given module can exist simultaneously. Standard ABE frameworks produce reentrant composite modules, so multiple instances do not present any problem. However, ABE cannot guarantee that imported foreign code will be reentrant.

Given a piece of non-reentrant foreign code, the module builder has three alternatives: modify the foreign code to make it reentrant, patch around the foreign code to make it appear reentrant from the outside, or ensure that no more than one instance of the module is active at one time.

The first alternative requires the module builder have access to sources and understand the foreign code enough to modify it correctly. The second method requires the module builder to identify the global objects used by the foreign code and create a copy of them for each instance of the importer module, storing them in the importer module's ivars. The third alternative requires cooperation among the modules that will interact with the importer module, as ABE currently provides no such support.

3.3.2. User Interface

ABE provides a standard system developer user interface for displaying the operation of composite modules. It does not provide any standard facilities for displaying the operation of primitive modules. When the foreign code needs to communicate with the user, it can use its own user interface.

Problems can arise when the foreign code's user interface makes assumptions about the processing environment in which it runs. For instance, if the foreign code assumes it has free access to write to the screen or read from the keyboard, the native window system can easily become confused.

At other times, the module builder will wish to suppress the foreign code's user interface entirely. Rebinding output streams that the foreign code uses to a null stream provides a quick method for ignoring all output from the foreign code.

We plan to adopt a standard for user interfaces in the near future. This standard will include prescriptions for user interfaces for imported code.

3.3.3. Virtual Memory Considerations

The current implementation of ABE runs on a Symbolics 3600 series computer. All software, including the kernel ABE system, all ABE modules, and all foreign code, must run in the same virtual memory. The Common LISP package system provides a means for multiple programs to share the same virtual memory without having name conflicts. However, the package system does not prevent conflicts of package names. These often occur for package nicknames, which can be one- or two-character strings. If the module builder wishes to load foreign code that has a package name conflict with an existing package, the existing package must be renamed to avoid conflict with the new package. The code in the renamed package will still continue to function properly, but editing it can cause problems.

Care must also be taken to watch for other conflicts among globally shared resources. Examples include readtable definitions and read macros, `Select` key commands, command processor commands, editor key bindings, and other environment customizations.

4. Abstract Datatypes

The MOP model calls for modules to communicate with each other by exchanging structured data. This data takes the form of *abstract datatypes* (ADTs). Modules take ADTs as input and return ADTs as outputs. In addition, ABE uses ADTs to act as interfaces to foreign languages and modules.

ABE provides the programmer flexibility in defining the physical storage of ADTs. The ABE ADT facility provides mechanisms for defining new storage methods for ADTs, particularly for accessing data structures defined by foreign code or languages. By default, ABE implements

ADTs as CORAL instances. An ABE user should not require other implementation methods unless he has to integrate with foreign code.

4.1. Logical Definition

An ADT is an object which logically comprises a number of named *slots*. Each slot contains a single value or a number of values. Values are LISP objects, including other ADTs. An ADT may specify a type for its slot values. An ADT generally supports read and write operations on its slots; the programmer can selectively declare slots read-only.

ABE also supports a special type of slot called a *link*. Links are like slots, except that ABE constrains their value to be another ADT instance or list of instances. Furthermore, each instance pointed to must have a link pointing back to the original ADT. We refer to that second link as a *back pointer*. Links automatically maintain the interconnections between complex structured objects.

ADTs are persistent, mutable objects. Modules access and modify the values of an ADT's slots by sending it messages. In response to these messages, an ADT can change state, send other messages to other related ADTs, or return a value based on its slot values.

4.2. Physical Definition

ABE provides the programmer flexibility in defining the physical storage of ADTs. The ADT facility provides mechanisms for defining new storage methods for ADTs, particularly for accessing data structures defined by foreign code or languages, *e.g.*, KEE Units. It also allows the programmer to specify access schemes for reading data from the foreign data structure.

By default, ABE allocates storage for an ADT's slots within the ADT instance. In this case, we say that the ADT instance *caches* its slot values. However, a programmer can specify alternate storage for slot values, such as a hashtable. In this case, the ADT *forwards* slot access requests to the appropriate data structure. The ADT instance then contains a pointer to the data structure that holds the slot values. Finally, a *hybrid* ADT maintains both local and remote storage.

The ADT facility provides multiple access schemes to support the integration of foreign languages and modules, *i.e.*, systems implemented outside of the ABE environment. The facility lets a programmer create standard interfaces to data structures in the foreign module, which other ABE modules can then use. Forwarding ADTs provide a "window" into foreign code; if slot values change from inside that code (*i.e.*, not through ADT messages), subsequent ADT accesses will return the changed values. Forwarding ADTs also provide means for creating new foreign data structures, containing a given set of initial slot values.

Conversely, a caching ADT provides a "snapshot" into a foreign data structure. The ADT copies all slot values out of the foreign structure into the ADT instance when the instance is created or when a slot is accessed for the first time; subsequent changes to the underlying data structure do not affect the ADT instance. The ASW-PREFERENCES-KEE ADT defined in Figure 3-3 is a caching ADT that reads initial slot values out of a KEE Unit when an ADT instance is created.

The hybrid ADT access mechanism provides a number of different capabilities. It can support selective caching or forwarding for particular slots, or implement "read-through/copy-on-write" access to the foreign data structure. Certain applications may require one or more types of slot access methods.

Figure 4-1 illustrates the structure of an ADT, which contains a pointer to an external data structure and/or local storage. The figure shows the use of KEE Units and NIKL[5] concepts as storage mechanisms for ADTs.

4.2.1. Defining New Storage Methods

In order to create a new ADT storage method, the programmer needs to inform the ADT facility how to access an external data structure. The ADT facility supports a simple model of data structures interfaces, consisting of a number of standard functional capabilities such as adding a new value to a given slot. The programmer must supply a function for each capability. With these functions, the ADT facility compiles a new storage method automatically.

Each ADT storage methods must provide an implementation for each of these capabilities:

- Add a new value to a set of values.
- Delete a given value from a set of values.
- Replace all values with a single new value.

[5] NIKL is a knowledge representation language developed by USC/ISI.

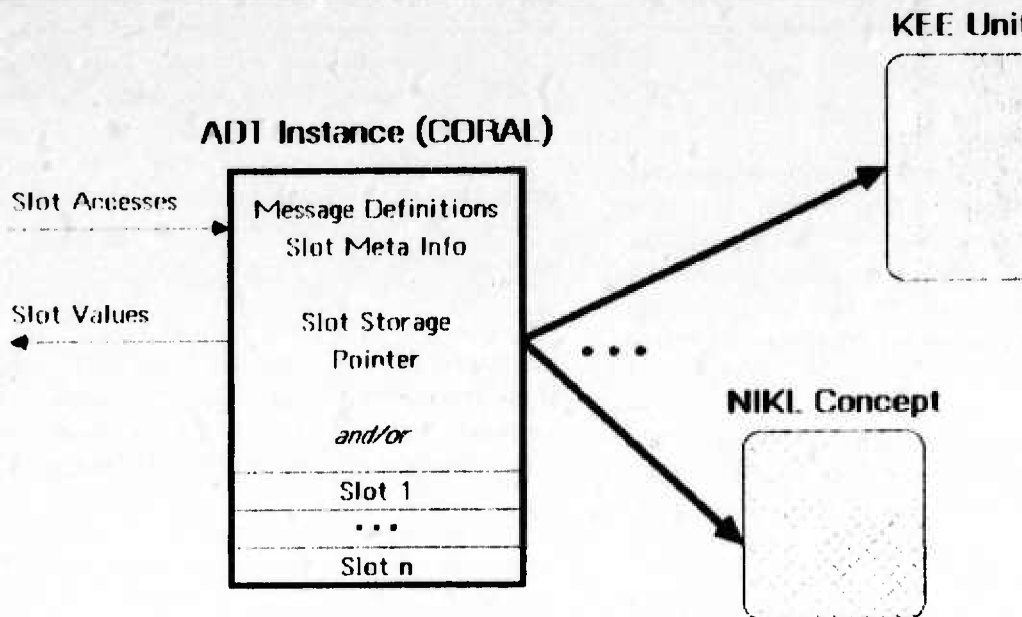


Figure 4-1: Abstract Datatype (ADT) structure

- Replace all values with a set of new values.
- Return a single slot value.
- Return all slot values as a list.

The ADT facility uses these supplied capabilities to build the responses to access and update an ADT instance's slots and links. In addition, an ADT's physical definition must specify the type of remote slot accesses, *e.g.*, copy-on-write, *etc.*

Figure 4-2 shows the definition of one of the interface functions for the KEE ADT interface. This function implements the "add a value to a slot" capability by calling the corresponding KEE function ADD.VALUE. We define the capability as a macro for performance considerations.

```
(defmacro ADD-KEE-ADT (adt slot new)
  "Add a new value to a KEE ADT slot."
  `(kee::add.value (Kee-Unit adt) ,slot ,new))
```

Figure 4-2: ADT Function to add a slot value for KEE Units

5. Summary

Intelligent systems are characterized by their need to solve large-scale problems, to support the reuse of software components, to have intelligible definitions and behavior, and to integrate with their surrounding environment. ABE is a new generation software system that supports the construction of intelligent systems. It provides a general-purpose software architecture for building intelligent systems based on its Module-Oriented Programming methodology.

Two of the basic concepts of Module-Oriented Programming are modules and abstract datatypes. Modules provide the basic building blocks used to construct intelligent systems. Primitive modules allow a module builder to define a new module using an ABE-supported language such as Common LISP or KEE. Importer modules are a special case of primitive modules that allow a module builder to import a piece of foreign code and make it look like an ABE module.

Abstract datatypes permit a module or system builder to describe his data at a high level without worrying about the implementation. Conversely, it also allows a user to create interfaces to foreign languages and code.

Acknowledgments

I would like to thank the following people and organizations for their contributions to the ABE project. The rest of the project team currently consists of Lee Erman and Terry Barnes, with substantial contributions from Rick Hayes-Roth, Neil Jacobstein, and Scott Fouse. Cliff Hollander and Terry Barnes implemented the AADS application, based on the ASW model provided by Dick Goodbody of the Naval Ocean Systems Command. Ray Bates of the University of Southern California/ Information Sciences Institute implemented the NIKL ADT interface. IntelliCorp provided the KEE software used in the AADS application.

I would also like to thank the following people for reviewing early drafts of this paper and providing helpful comments and criticisms: Terry Barnes, Lee Erman, Jim Davidson, and the rest of the Research and New Product Development group at Teknowledge.

References

[Chandrasekaran 86] Chandrasekaran, B. and J. Josephson. Explanation, Problem Solving, and New Generation Tools: A Progress Report. In *Proceedings: Expert Systems Workshop*, pages 101-126. SAIC, 1986. Sponsored by DARPA.

[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 86] Erman, L. D., J. S. Lark, F. Hayes-Roth. Engineering Intelligent Systems: Progress Report on ABE. In *Proceedings: Expert Systems Workshop*, pages 89-100. SAIC, 1986. Sponsored by DARPA.

[Hollander 86] Hollander, C. and T. Barnes. *AADS: ABE/ASW Demonstration System.* Technical Report TTR-ISE-86-103, Teknowledge, Inc., 1986.

[IntelliCorp 85] *KEE Software Development System User's Manual* IntelliCorp, Mountain View, CA, 1985.

[Kaczmarek 86] Kaczmarek, T. S., R. Bates, G. Robins. Recent Developments in NIKL. In *Proceedings: Expert Systems Workshop*, pages 191-197. SAIC, 1986. Sponsored by DARPA.

[Knowledge Craft 85] *Knowledge Craft Manual Guide* Carnegie Group Inc., Pittsburgh, PA, 1985.

[Rashid 86] Rashid, R. F. Threads of a New System. *UNIX Review* :37-49, August, 1986.

[Russell 85] Russell, S. *The Compleat Guide to MRS.* Technical Report KSL-85-12, Stanford Knowledge Systems Laboratory, Computer Science Dept., Stanford University, 1985.

[SPAWARSYSCOM 85] Space and Naval Warfare Systems Command. CASES (Capabilities Assessment) Functional Description.

[Teknowledge 87a] *ABE Programmer Reference Manual* Teknowledge, Inc., 1987. Report number TTR-ISE-87-102.

[Teknowledge 87b] *CORAL Reference Manual* Teknowledge, Inc., 1987. Report number TTR-ISE-87-101.

Copyright © 1987 Teknowledge, Inc.

Annual Report of the Experimental Knowledge Systems Laboratory

Edited by Paul R. Cohen
Department of Computer and Information Science
Lederle Graduate Research Center
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

This paper discusses DARPA-sponsored research in the Experimental Knowledge Systems Laboratory at the University of Massachusetts. The work focuses on the control of reasoning under uncertainty. This paper includes reports on the MU architecture, the role of task-level architectures for knowledge acquisition, a task-level approach to acquiring local combining functions, and a language for describing the kind of short-term contingency plans that seem appropriate for reasoning under uncertainty. The paper also describes evaluations of the GRANT system — an approach to information retrieval based on partial matching.

I. Introduction

This is the annual report for 1986-1987 on our DARPA-sponsored research. Our work over the last year has been concerned with the control of problem solving under uncertainty. Sources of uncertainty — noisy data, incomplete knowledge, and imprecision about the effects of actions — render standard planning and problem-solving approaches useless. These approaches depend on complete knowledge of the states of the world and complete prior knowledge of the effects of actions, neither of which can be guaranteed in uncertain environments. But problem solvers must act, nonetheless. Our research addresses the issue of how problem-solvers' control strategies — their mechanisms for focus of attention, control of inference, and control of actions — can affect the efficiency with which they solve problems, where efficiency measures roughly the tradeoff between costs and certainty.

We have been building an architecture called MU for problem-solvng under uncertainty. MU is a generalization of our earlier work on managing uncertainty in medicine, described in our previous annual report. Section 2 of this paper describes MU in some detail. MU is meant to facilitate the transfer of strategic and tactical expertise, thus much of our effort has gone into building tools for this purpose. Section 3 describes the intellectual basis for this work — it is closely related to Chandrasekaran's ideas about generic tasks (1986). The position described in Section 3 sets the tone for our research (particularly for a PhD thesis on automatic acquisition of control strategies — to be discussed in the next annual report). The impact of this perspective is seen in Section 4, where we discuss approaches to acquiring knowledge about combinations of evidence. Here, we contrast the global combining functions typically used in expert systems with the local functions developed for MU. Section 4 discusses the strengths and weaknesses of each approach and settles on a hybrid approach.

MU makes it easy to acquire and make explicit the *control features* on which control decisions depend. Similarly, it is easy to specify the functions that keep these features current, and to query their values during problem-solving. But as yet, these abilities have not been fully exploited by the strategic component of our problem solvers. This is because we have yet to develop an acceptable representation for strategies and tactics. We have adopted meta-rules (Davis and Buchanan, 1984) to express preferences between actions. This leads to an iterative view of problem-solving, where the problem-solver stops after every action to check its effects before selecting the next action. Very recently, we have developed a representation for short contingency plans, which should allow the problem-solver to plan a sequence of actions, each contingent on the outcomes of previous ones. This is discussed in Section 5.

Although much of our work is concerned with MU, we also performed an exhaustive series of tests on the GRANT system (discussed in the previous annual report). The results of these evaluations are discussed in Section 6.

This report is culled from five papers and reports:

Paul Cohen, Michael Greenberg, and Jefferson DeLisio. 1987. MU: A Development Environment for Prospective Reasoning Systems. *AAAI-87*, July, 1987.

Thomas Gruber and Paul Cohen. 1987. Knowledge Engineering Tools at the Architecture Level. *International Joint Conference on Artificial Intelligence (IJCAI-87)*, August, 1987.

Paul Cohen, Glenn Shafer, and Prakash Shenoy. 1987. Modifiable Combining Functions. *EKSL Report 87-05*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

Jefferson DeLisio, 1987. A Notation for Representing Strategies. *EKSL Report 87-08*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

Paul Cohen and Rick Kjeldsen. 1987. Information Retrieval by Constrained Spreading Activation in Semantic Networks. *Journal of Information Processing and Management*, Forthcoming.

Other publications and reports for this year are listed below. A brief annotation for each describes its contents:

Paul Cohen, David Day, Jeff Delisio, Michael Greenberg, Rick Kjeldsen, Daniel Suthers and Paul Berman. 1986. Managing of Uncertainty in Medicine. *IEEE Conference on Computers and Communications*. February, 1987. Also, to be published in *Int. Journal of Approximate Reasoning*,

This describes the MUM system as it was implemented in 1986. We have since reimplemented it in MU.

Paul R. Cohen. 1987. Steps Towards Programs that Manage Uncertainty. *EKSL Report 87-06*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

This includes a detailed analysis of diagnostic strategy, discusses the nature of problem solving under uncertainty, and describes the transition from MUM to MU.

Thomas Gruber and Paul Cohen. 1986. Knowledge Engineering Tools at the Architecture Level. *AAAI Workshop on High Level Tools*, October 7, 8.

This is a longer version of the IJCAI paper, cited above.

Thomas Gruber and Paul R. Cohen. 1986. Principles of Design for Acquisition. 1987. *3rd IEEE Conference on Artificial Intelligence Applications*. Orlando, Florida. February, 1987.

Thomas Gruber and Paul Cohen. 1986. Design for Acquisition: Principles of Knowledge System Design to Facilitate Knowlege Knowledge Acquisition. *AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, November 3-7, 1986. A revised version of this paper will appear in *The International Journal of Man-Machine Studies*, Spring 1987.

These papers describe three principles of design for task-level architectures, and illustrate them with examples from MU and MUM. The principles occassionally conflict. This is especially apparent in the context of acquiring strategic knowledge from experts, and it has led to a PhD thesis on automating this process.

Thomas Gruber. Acquiring Strategic Knowledge from Experts. 1987. *EKSL Report 87-07*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

This paper discusses why strategic knowledge is difficult to acquire and includes a detailed example of how an intelligent assistant can facilitate its acquisition.

Adele Howe and Paul R. Cohen. 1986. A Typology for Constructing Decisions *Proceedings of IEEE Conference on Computers and Communications*, February, 1987.

Adele Howe and Paul R. Cohen. 1987. Dynamic Construction of Decisions. *EKSL Report 87-09*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

Two papers on the constructive decision making algorithm — the second is a short, summary paper. The algorithm is a qualitative approach to decision making that stresses the construction of decisions in real time. It was designed specifically for "apples and oranges" problems, in which the decision-maker has good reasons for selecting mutually exclusive alternatives.

Adele Howe. 1987. A Constructive Approach to Decision Making. Master's Project. *EKSL Report 87-02*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

This is the definitive paper on the constructive decision making approach. It describes the approach in detail, how it is implemented, and gives examples of how it controls the construction of simple decisions.

John Harhen and Paul Cohen. 1987. Using Multiple Perspectives to Manage Uncertainty. *EKSL Report 87-04*, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

This paper describes extensions to the theory of endorsements for long-range business planning. The idea is that several *methods* can be used to answer any question about a company's long-range needs — each of which has an endorsement or pedigree that can be used to judge the credibility of its result.

Max Henrion and Daniel R. Cooley. An experimental comparison of knowledge engineering for expert systems and for decision analysis. Accepted for AAAI-87.

Max Henrion is a decision theorist from Carnegie-Mellon University and Dan Cooley is a plant pathologist from the University of Massachusetts. They teamed up with people from EKSL to do a pilot study of the relative merits of MU and decision analysis.

Paul R. Cohen, Philip Stanhope, and Rick Kjeldsen. 1986. Classification by Semantic Matching. *Proceedings of IEEE Conference on Computers and Communications* February,1987.

Rick Kjeldsen and Paul Cohen. The Evolution and Performance of the GRANT System. *3rd IEEE Conference on Artificial Intelligence Applications*. Orlando, Florida. February, 1987. Also selected to appear in a special issue of *IEEE Expert*, 1987.

These papers describe tests and modifications to the GRANT system.

II. MU: A Development Environment for Prospective Reasoning Systems

MU is a development environment for knowledge systems that reason with incomplete knowledge. It has evolved from a program called MUM that planned diagnostic sequences of questions, tests, and treatments for chest and abdominal pain (Cohen, *et al.*, 1986). This task is called prospective diagnosis, because it emphasizes the selection of actions based on their potential outcomes and the current state of the patient. Prospective diagnosis is uncertain because the precise outcomes of actions cannot be predicted, in part because knowledge of the state of the patient is incomplete. Yet we have found that physicians have rich strategic knowledge with which they plan diagnoses in spite of their their uncertainty. MU does not provide a knowledge engineer with any particular strategies, but rather provides an environment in which it is easy to acquire, represent, and experiment with a wide variety of strategies for prospective diagnosis and other *prospective reasoning* tasks.

Three goals underlie our research and motivate the MU system. First, MU is intended to provide knowledge-engineering tools to help acquire expert problem-solving strategies. MU allows us to define explicit *control features*, which are the terms an expert uses to discuss strategies. Control features in medical diagnosis include degrees of belief in disease hypotheses, monetary costs of evidence, the consequences of incorrect conclusions, and "intangibles" such as anxiety and discomfort. Some, like degrees of belief, have values that change dynamically during problem solving. MU helps the knowledge engineer define the functions that compute these dynamic values and keeps the values accessible during problem solving. For example, with MU we can easily define a control feature called *criticality* in terms of two others, say *dangerousness* and *degree of belief*, and acquire a function for dynamically assessing the criticality of a hypothesis as its degree of belief changes.

Second, we want to show that strategies enable a prospective reasoning system to produce solutions that are *efficient* in the sense of minimizing the costs of attaining given levels of certainty MU has no "built in" problem solving strategies, but we have been able to acquire and implement efficient, expert strategies in MU because we can define explicit control features that represent the various costs of actions, as well as the levels of certainty in the evidence produced by actions.

Third, we want to implement in MU a *task-level architecture* for prospective reasoning (Gruber and Cohen, 1987), an environment for building systems that plan efficient sequences of actions, despite uncertainty about their outcomes. After working in the domains of medicine and plant pathology, we now think that many control features pertain to diagnostic tasks in general. Moreover, diagnosticians in many fields seem to use similar strategies to solve problems efficiently. This view is influenced by the recent trend in AI toward defining *generic tasks* (Chandrasekaran, 1986) such as classification (Clancey, 1985) and the architectures that support their implementation. MU shares the orientation toward explicit control efforts such as BB* (Hayes-Roth, 1985; Hayes-Roth, *et al.*, 1986) and Heracles (Clancey, 1986), but emphasizes control features that are appropriate for prospective reasoning. In sum, MU is a tool for representing and providing access to the knowledge that underlies efficient prospective reasoning. This report begins with an analysis of prospective reasoning, then describes the MU environment first as a program, emphasizing its structure and function, then from the perspective of the knowledge engineer who uses it. As an illustration, we describe how MUM was reimplemented in MU. We conclude with a summary of current work.

A. Prospective Reasoning

Prospective reasoning is reasoning about the question "What shall I do next," given that

1. knowledge about the current state of the world is incomplete,
2. the outcomes of actions are uncertain,
3. there are tradeoffs between the costs of actions with respect to the problem solver's goals and the utility of the evidence they provide,
4. states of knowledge that result from actions can influence the utility of other actions.

An example from medical diagnosis illustrates these characteristics:

> A middle-aged man reports episodes of chest pain that could be either angina or esophageal spasm; the physician orders an EKG, but it provides no evidence about either hypothesis; then he prescribes a trial prescription of vasodilators; the patient has no further episodes of pain, so the physician keeps him on long-acting vasodilators and eventually suggests a modified stress test to gauge the patient's exercise tolerance.

The first and second characteristics of prospective reasoning are clearly seen in this case: Knowledge about the state of the patient is incomplete throughout diagnosis, and the outcomes of actions (the EKG, trial therapy, stress test) are uncertain until they are performed and are sometimes ambiguous afterwards. Less obvious is the third characteristic, the tradeoffs inherent in each action. Statistically, an EKG is not likely to provide useful evidence, but if it does, the evi-

dence will be completely diagnostic. The EKG is given because its minimal costs (e.g., time, money, risk, and anxiety) are offset by the possibility of obtaining diagnostic evidence[1]. Similarly, trial therapy satisfies many goals; it protects the patient, costs little, has few side-effects and, if successful, is good evidence for the angina hypothesis.

The fourth characteristic of prospective reasoning is that states of knowledge that result from actions can affect the utility other actions. This is because the costs and benefits of actions are judged in the context of what is already known about the patient. For example, trial therapy is worthwhile if the EKG does not produce diagnostic evidence, but is redundant otherwise. The outcome of an EKG thus affects the utility of trial therapy. This implies a dependency between the actions, and suggests a strategy: do the EKG first because, if it is positive, then trial therapy will be unnecessary.

Dependencies between actions help the prospective reasoner to *order* actions. We call this planning, though it is not planning in the usual AI sense of the word (Sacerdoti, 1979; Cohen and Feigenbaum, 1982). The differences are due to the first and second characteristics of prospective reasoning: the state of the world and the effects of actions are both uncertain. The prospective planner must "feel its way" by estimating the likely outcomes of one or more actions, executing them, then checking whether the actual state of the world is as expected. Plans in prospective reasoning tend to be short. In contrast, uncertainty is excised from most AI planners by assuming that the initial state of the world and the effects of all actions are completely known (e.g., the STRIPS assumption, Fikes, Hart, Nilsson, 1972). AI planners can proceed by "dead-reckoning," because it follows from these assumptions that *every* state of the world is completely known. All further discussions of planning in this report refer to the "feel your way" variety, not to "dead reckoning."

Prospective diagnosis requires a planner to select actions based on their costs and utility given the current state of knowledge about the patient. We have described prospective reasoning as planning because the evidence from one action may affect the utility of another. Alternatively, prospective reasoning can be viewed as a series of decisions about actions, each conditioned on the current state of knowledge about the patient. We considered decision analysis (Raiffa, 1970; Howard, 1966) as a mechanism for selecting actions in prospective reasoning, but rejected it for two reasons. First, collapsing control features such as monetary expense, time, and criticality into a single measure of utility negates our goals of explicit control and providing a task-level architecture for prospective reasoning (Cohen, 1985; Gruber and Cohen, 1987). Second, decision analysis requires too many numbers — a *complete*, combinatorial model of each decision. The expected utility of each potential action can only be calculated from the joint probability distribution of the possible outcomes of the previous actions. But although we do not implement prospective reasoning with decision analysis, MU is designed to provide qualitative versions of several decision-analytic concepts, including the utility of evidence and sensitivity analysis.

B. The MU Environment – An Overview

A coarse view of MU's structure reveals these components:

- a frame-based representation language,
- tools for building inference networks,
- an interface for defining control features and the functions that maintain their values,
- a language for asking questions about the state of a problem and how to change its state.
- a user interface for acquiring data during problem-solving,

With these tools, a knowledge engineer can build a knowledge system with a planner for prospective reasoning. MU does not "come with" any particular planners, but it provides tools for building planners and incorporating expert problem-solving strategies within them.

Among MU's tools is an editor for encoding domain inferences, such as *if EKG shows ischemic changes then angina is confirmed*, in an inference network. MU does not dictate what the nodes in the inference network should represent, except in the weak sense that nodes "lower" in the network — relative to the direction of inference — provide evidence for those "higher" up. However, the nodes in the network are usually differentiated; for example, in Figure 1 some nodes represent raw data, others represent combinations of data (called *clusters*), and a third class represents hypotheses. In the medical domain, data nodes represent individual questions, tests, or treatments. Clusters combine several data; for example, the *risk-factors-for-angina* cluster combines the patient's blood pressure, family history, past medical history, gender, and so on. Hypothesis nodes represent diseases such as *angina*.

Since MU does not provide a planner, the knowledge engineer is required to build one. The planner should answer two questions:

[1]This example oversimplifies the reasons for giving an EKG, but not the cost/benefit analysis that underlies the decision.

An Inference Net in MU

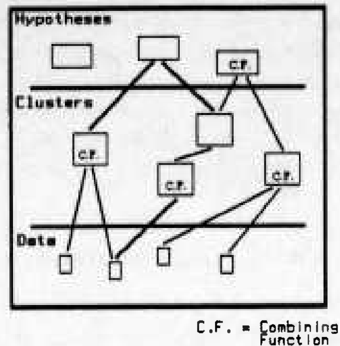


Figure 1: Organization of Knowledge Within Mu

MU System

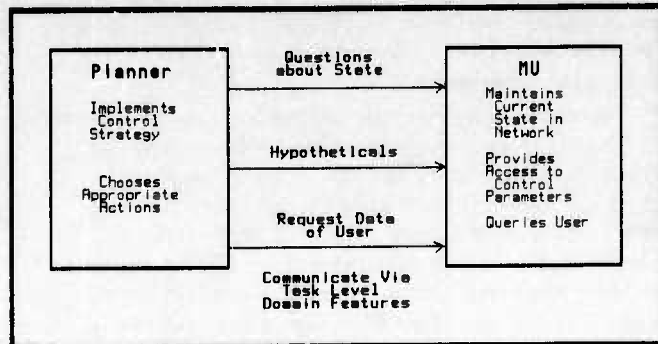


Figure 2: Mu System Schematic

- Which node(s) in the network should be in the focus set, and which of these should be the immediate focus of attention?
- Which actions are applicable, given the focus set, and which of these should be taken?

For example, in the medical domain the focus set might include all disease hypotheses that have some support, and the immediate focus of attention might be the most dangerous one. The potential actions might be the leaf nodes of the tree rooted at the focus of attention (Fig. 1), and the selected action might be the cheapest of the potential actions.

MU provides an interface to help the knowledge engineer define control features such as the degree of belief in hypotheses, the dangerousness of diseases, and the costs of diagnostic actions. It also provides a language with which a planner can query the values of features and ask about actions that would change those values. Planners can ask, for example, "What is the current level of belief in angina?" or "Tell me all the inexpensive ways to increase the level of belief in angina," or even the hypothetical question, "Would the level of belief in angina change if blood pressure was high?"

The relationship between these functions of MU and the functions of a planner are shown in Figure 2. Using MU, a knowledge engineer can: define a control feature such as criticality in terms of other features such as dangerousness and degree of belief; specify a combining function for calculating dynamically the value of criticality from these other features during problem solving; associate criticality and its combining function with a class of nodes, such as diseases, and have each member of the class inherit the definitions; and write a planner that encodes an expert strategy for dealing with critical or potentially-critical diseases. MU facilitates the development of planners, and makes their behavior explicit and efficient, but the *design* of planners, and the acquisition of strategies and the control features on which they depend, is the job of the knowledge engineer.

C. The MU Environment – Features and Combining Functions

Knowledge representation in MU centers around features. Features and their values are the information with which planning decisions are made. Each node in a MU inference network can have several features; for example, the node that represents trial therapy for angina includes features for monetary cost and risk to the patient. Features are defined in the normal course of knowledge engineering to support expert strategies for prospective reasoning. We have identified four classes of features, differentiated by their value types, how they are calculated, and the operations that MU can perform on them:

Static The value of a static feature is specified by the expert and does not change at run time. *Monetary cost* is a typical static feature, as the cost of an action does not change during a session.

Datum The value of a datum feature is acquired at run time by asking the user questions. Data are often the results of actions; for example *EKG shows ischemic changes* is a potential result of performing an EKG.

Dynamic The value of a dynamic feature is computed from the values of other feature values in the network. The value of each dynamic feature is calculated by a combining function, acquired through knowledge engineering. A dynamic fea-

ture of every hypothesis is its *degree of belief* — a function of the degrees of belief of its evidence.

Focus The value of a focus feature is a set of nodes whose features satisfy a user-defined predicate. Focus features are a subclass of dynamic features. In medicine, the *differential* focus feature can be defined as the list of all triggered hypotheses that are not confirmed or disconfirmed.

Feature values can belong to several data types, including integers, sets, normal (one of an unordered set of possible values), ordinal (one of an ordered set of possible values), boolean, and relational (e.g., isa).

Four operations are defined for features: one can *set* a feature value (e.g., assert that the monetary cost of a test is high) *get* a feature value (e.g., ask for the cost of a test), ask *how to change* a feature value, and ask *what are the effects* of changing a feature value. Planners need answers to these kinds of questions to help them select actions (see Section 5 for further examples.)

All combinations of feature type, value type, and operations are not possible. Figure 3 summarizes the legal combinations.

MU provides an interface for defining features. A full definition includes the feature type, value type, its range of values, and the domain of its combining functions. For instance, the dynamic feature *level of support* is defined to have seven values on an ordinal scale: disconfirmed, strongly-detracted, detracted, unknown, supported, strongly-supported and confirmed. Figure 4 shows the definition of level of support.

Instances of this feature (and others) are associated with individual hypotheses, each of which may have its own, local function for calculating level of support, and its own, dynamic value for the feature[2]. For example, Figure 5 shows part of the frame for the angina hypothesis, encompassing an instance of the level of support feature, and showing a fragment of the function for calculating its value for angina.

Level-Of-Support

Feature-type: Dynamic

Value-Type: Ordinal

Value-restriction:
(disconfirmed strongly-detracted detracted unknown supported strongly-supported confirmed)

Value: *the current level of support of the hypothesis*

Combination-function-slot: *local to each hypothesis*

Figure 4: Definition of Level-Of-Support

Combining functions calculate values for dynamic features such as level of belief, criticality, elapsed time, and so on. They serve two important functions: First, they keep the state of MU's inference network up-to-date; for example, when the result of an EKG becomes available, the combining function for the angina node updates the value of its level of support feature accordingly.

Angina

Feature-list: (level-of-support severity)

Current-level-of-support:
strongly-supported

Combination-function:
IF value of ekg is ischemic-changes
THEN angina is confirmed
ELSEIF episode-incited-by contains exertion
AND
risk-factors-for-angina are supported
THEN angina is strongly-supported . . .

Figure 5: Part of the Angina Frame With Local Combining Function

	Data Types				Questions			
Feature	Number	Set	Ordinal	Normal	Get	Set	How To	Effect Of
static	X	X	X	X	X			
datum	X	X	X	X	X	X		X
dynamic	X		X	X	X		X	
focus		X			X			

Figure 3: Capabilites By Feature Type

[2]Not all feature values are calculated locally, but, for reasons discussed in Cohen, Shafer, and Shenoy (1987) and Cohen, *et al.*, (1986) levels of belief are.

Second, and perhaps more important from the standpoint of a planner, combining functions provide a *prospective* view of the effects of actions; for example, the combining function for angina can be interpreted prospectively to say that EKG can potentially confirm angina. The same point holds for the combining functions for other features: MU can prospectively assess the potential effects of actions on all dynamic features. A planner can ask MU, "If EKG is negative, what changes?" and get back a list of all the features of all data, clusters, and hypotheses that are in some way affected by the value of EKG. The effects of actions are assessed in the context of MU's current state of knowledge (i.e., the state of its network). For example, if an EKG has been given and its results were negative, then MU knows that the answer to the previous question is that nothing changes.

The syntax of combining functions is relatively unimportant provided they are declarative, so MU's question-answering interface can read them, and experts can easily specify and modify them. Currently, combining functions look like rules, but we are experimenting with tabular and graphic forms (Cohen, Shafer, and Shenoy, 1987).

The two major classes of combining functions are *local* and *global*. A local function for a node such as angina refers only to the nodes in the inference network that are directly connected to angina. In contrast, global functions survey the state of MU's entire inference network. Functions for focus features take a global perspective because the value of a focus feature is the subset of nodes in the network whose features satisfy some predicate. For example, Figure 6 illustrates the combining function for the *differential* focus feature. Any node that represents a disease hypothesis, and is triggered, but is neither confirmed nor disconfirmed is a member of the differential.

Differential

feature-list: (focus-feature)

current-focus: (angina prinz-metal ulcer)

combining-function:

Set-of \$node\$ member-of disease Such-that
\$node\$ is triggered AND
level-of-support of \$node\$ is not confirmed AND
level-of-support of \$node\$ is not disconfirmed

Figure 6: Part of the Global Focus-Feature **Differential**

The knowledge engineer can define many focus features, each corresponding to a class of nodes that a planner may want to monitor. Besides the differential, a planner might maintain the set of critical hypotheses (e.g., all dangerous hypotheses that have moderate support or better), or the set of hypotheses that have relatively high prior probability, or the set of all supported clusters that potentially confirm a particular hypothesis. MU supports set intersection, union, and sorting on the sets of nodes maintained by focus features. A planner's current focus of attention is represented in terms of the results of these operations.

D. MU from the Knowledge Engineer's Perspective

MU is a development environment for prospective reasoning systems. We began our research on prospective reasoning when we were building a system MUM, for prospective diagnosis (Cohen, *et al.*, 1986), and realized that we lacked the knowledge engineering tools to acquire and modify diagnostic strategies. An example will illustrate the knowledge engineering issues in building MU:

MUM had several *strategic phases*, each of which specified how to assess a focus of attention and select an action. One phase, called *initial assessment*, directed MUM to focus on triggered hypotheses one by one and take inexpensive actions that potentially support each. This covered a wide range of situations, and maintained the efficiency of diagnoses by focusing on low-cost evidence, but it made little sense for very dangerous disease hypotheses. For these, diagnosticity — not cost — is the most important criterion for selecting actions. Once the expert explained this, we should have immediately added a new strategic phase, run the system, and iterated if its performance was incorrect. Unfortunately, control features such as criticality and diagnosticity did not have declarative representations in MUM, were implemented in lisp, and could not easily be composed from other control features. Operations such as sorting a list of critical hypotheses by their level of support were also implemented in lisp. Each strategic phase required a day or two to write and debug. From the standpoint of the expert, it was an unacceptable delay.

The MUM project showed us that MU should facilitate acquisition of control features, maintain their values efficiently, and support a broad range of questions about the state of the inference network. MU allows a planner to ask 6 classes of questions:

Questions about **state** are concerned with the current values of features. For example:

Q1: "What is the current level of support for angina?"

Q2: "Is an ulcer dangerous?"

Q3: "What is the cost of performing an angiogram?"

Another class of questions is asked to find out how to achieve a **goal**. Examples of questions about goals are:

Q4: "Given what I know now, which tests might confirm angina?"

Q5: "What are all of the tests that might have some bearing on heart disease?"

These questions help a planner identify relevant actions and select among them. Those that pertain to levels of belief are answered by refering to the appropriate combining functions and current levels of belief. For example, the answer to the question about angina is "EKG," if an EKG has not already been performed (Fig. 5).

Questions about the **effects** of actions allow a planner to understand the ramifications of an action. For example,

Q6: "Which disease hypotheses are affected by performing an EKG?

Q7: "What are the possible results of an angiogram?"

Q8: "Does age have an effect on the criticality of colon cancer?"

MU answers these questions by traversing the relations between actions and nodes "higher" in the inference network. For example, Q6 is answered by finding all the nodes for which EKG provides evidence. The planner may ask either for the *immediate* consequence of knowing EKG, or for the consequences to any desired depth of inference.

Focus questions help a planner establish focus of attention. For example:

Q9: "Give me all diseases that are triggered and dangerous."

Q10: "What are all of the critical diseases for which I have no information?"

Q11: "Are any hypotheses confirmed?"

Questions about **multiple effects** allow the planner to combine the previous question types into more complex queries such as "What tests can discriminate between angina and esophageal spasm?" In this case, the term *discriminate* is defined to mean "simultaneously increase the level of belief in one disease and lower it in an other."

Hypothetical questions allow the planner to identify dependencies among actions. For example, one can ask, "Suppose the response to trial therapy is positive. Now, could a stress test still have any bearing on my belief in angina?"

With the ability to define control features and answer such questions, we quickly reimplemented MUM's strategic phase planner. Most of the effort went into adding declarative definitions of control features and their combining functions to MUM's medical inference network.

E. Conclusion

MU supports the construction of systems that have the characteristics of prospective reasoning identified in Section 2: Prospective reasoning involves answering the question, "What shall I do next," given uncertainty about the state of the world, the effects of actions, tradeoffs between the costs and benefits of actions, and precondition relations between actions. The six classes of questions, discussed above, help planners to decide on courses of action despite uncertainty. Questions about **state** make uncertainty about hypotheses explicit. **Hypothetical** questions and questions about **effects** make uncertainty about the outcomes of actions explicit. Questions about **goals** and **multiple effects** help a planner identify the tradeoffs between actions. And **hypothetical** questions make dependencies between actions explicit.

III. Knowledge Engineering Tools at the Architecture Level

This report is about tools for knowledge engineering at the *architecture level*. A knowledge system architecture specializes common AI problem-solving techniques to a particular class of tasks. Architectures provide descriptions of particular kinds of problem solving (e.g., *diagnosis* or *configuration*) at a conceptual level that is above the implementation, thus making clear which aspects of a class of problems are intrinsic to the problem and which are artifacts of the implementation. Architectures are partial designs in which some decisions are made in advance to support particular task characteristics. For example, many medical diagnosis systems first interpret data bottom-up to find "triggered" disease hypotheses, then set top-down goals of acquiring evidence pro and con the triggered hypotheses. This "trigger/acquire evidence" cycle is an intrinsic part of any architecture for the class of medical diagnosis tasks, though it might be implemented in a wide variety of ways.

Architecture-level tools for knowledge engineers can improve the productivity of system development and knowledge acquisition because:

- By supporting the abstraction of representational and computational primitives at the architecture level, they permit the knowledge engineer and expert to cooperatively develop systems using a shared language of architecture constructs, rather than in terms of the underlying implementation.[3]

[3] Data abstraction and related methodologies such as object-oriented programming are well established software engineering techniques for reducing the complexity of large programs by hiding implementation details (Abelson and Sussman,1985). For knowledge

- They can incorporate knowledge about the architecture to facilitate system development and knowledge acquisition (e.g., by enforcing constraints on the types and values of elements in the knowledge base).

The idea of an architecture level underlies recent work on knowledge systems.[4] Chandrasekaran and his colleagues have identified a number of "generic tasks" such as hierarchical diagnosis and routine design, and have developed task-specific representation languages and control strategies for them (Chandrasakeran, 1986; Bylander and Mittal, 1986; Brown and Chandrasakeran, 1984). McDermott and colleagues have produced several knowledge systems using architectures that integrate knowledge acquisition tools with the problem solving methods (McDermott, 1983; Kahn, *et al.*, 1984; Eshelman and McDermott, 1986; Marcus, 1987; Khan, *et al.*, 1987). Clancey has described in detail the heuristic classification method embodied in the HERACLES architecture (Clancey, 1985, 1986). Newell (1982) anticipated much of this work in his AAAI President's Address on the *Knowledge Level*, where he distinguished the knowledge of an intelligent agent, which is used to model its behavior, from the knowledge representation that describes how the knowledge is encoded in a symbol system.

This report presents an analysis of the role of knowledge engineering tools at the architecture level. We describe three complementary views of what is meant by the architecture level, and illustrate them in the context of MU, an architecture for systems that manage uncertainty by deciding how to act. We show how the architecture-level analysis leads to a hierarchical organization of knowledge engineering tools to support software development and knowledge acquisition for MU systems. We conclude with some advantages of this approach to knowledge engineering.

A. Three views of the architecture level

Architectures can be viewed from three perspectives, each which suggests roles for architecture-level tools. First, the *functional* view presents an architecture as an application of general AI techniques to suit a particular style of problem solving. One might say that, functionally, the blackboard architecture is well-suited to problems with noisy data and multiple sources of evidence (Erman, *et al.*, 1980; Nii, 1986). There are architectures for simple classification (e.g., traversing decision trees), heuristic classification (e.g., HERACLES, Clancey, 1986; CSRL, Bylander and Mittal, 1986), constructing configurations (e.g., SALT, Marcus, 1987; COAST, Bennett, 1986), and design (e.g., CSRL, Brown and Chandrasekaran, 1984; DOMINIC, Howe *et al.*, 1986).

The second perspective is *structural*: an architecture is a partial design that includes specifications of knowledge representation formalisms, inference mechanisms, and control strategies. Many of these structural components are available from commercially available AI programming environments. Architectures, however, are not arbitrary combinations of these components, but "good" combinations designed by the knowledge engineer for particular tasks.

A third view of an architecture is that it defines a *virtual machine*. The architecture provides a language that describes the behavior of a system in terms natural for the knowledge engineer and expert. For example, most medical diagnosis systems provide some kind of support for *triggering* – making particular hypotheses "active" when certain events (typically input data) occur. To the expert, triggering might correspond to "bringing a diagnosis to mind." A programmer can produce the effect of triggering using implementation-level primitives (e.g., giving triggered diseases high certainty factors or agenda priorities). But terms such as "trigger" — not their implementation — are the medium of knowledge engineering. Such *task-level* terms promote explanation (Swartout, 1983) and knowledge acquisition[5] (Gruber and Cohen, 1986) Knowledge engineers, experts, and users can all understand triggering without thinking about how it is implemented. A virtual machine that executes "triggering" is easier to program.

The interactions of these views of the MU architecture are apparent in the design of knowledge engineering tools. Figure 7 shows a hierarchy of tools that supports development of systems in MU. The foundation is a commercially-available AI programming environment that includes implementation primitives such as rules and frames, and basic AI programming techniques such as pattern-matching rule interpreters and message-passing. The first layer in Figure 7 is a *structural* description of the implementation of MU. It is not a design for an architecture, because no *functional* description has been given or is implied by this collection of implementation primitives and AI programming techniques, which could be instantiated to provide a wide range of behaviors.

systems, the architecture is a particularly useful level of abstraction, and tools to support it reduce the inherent complexity of large knowledge-based programs by separating the representational and computational needs of the problem solving task from implementation decisions.

[5] Without task-level terms, the (non-programmer) expert is effectively barred from working directly with the knowledge base.

Tool Level	Objects in User's View	Software Support
Knowledge Acquisition Interface	*Application-specific Terms* diseases, intermediate diagnoses, questions, clinical tests, triggering symptoms for diseases, confirming test results, criticality of diseases, relative costs of tests, treatments, efficacy of treatment	*(Meta-)Knowledge-based Utilities* language-specific editors and form-filling terfaces, inferential consistency analy graphical display for objects and relation
Virtual Machine (shell)	*Task-level Constructs* hypotheses, intermediate conclusions, inferential relations, data descriptions, combining functions, control parameters, control rules, preference rankings among actions	*Task-specific Reasoning Mechanisms* value propagation functions, predicates the state of the inference net, rule-based pl ner, decision-making support
AI Toolbox (KEE)	*Implementation Primitives* frames and slots, rules, pattern matching language, Lisp objects and functions, windows and graphic objects	*AI Programming Techniques* knowledge base bookkeeping, rule in preter, knowledge base bookkeeping, inh tance mechanisms, assumption maintena demon invocation and message passing, dow system, network grapher

Figure 7: A hierarchy of knowledge engineering tools to support the MU architecture.

B. Tools for the MU Architecture

In this section we present an architecture for systems that reason under uncertainty, called MU, with the aim of illustrating how the three views of architectures influence the design of knowledge engineering tools. MU grew out of experience with MUM (Managing Uncertainty in Medicine), a system for planning a series of diagnostic questions, tests, and treatments for diseases manifesting chest and abdominal pain (Cohen, *et al.*, 1986) The primary aim of MUM is to decide how to act when data are insufficient for diagnosis and treatment. Like a physician, MUM reasons about tradeoffs between the costs of evidence, the marginal utility of potential data given what is already known, the effects of treatments and the evidence they provide, and so on. MU is an architecture for building systems like MUM that reason about uncertain situations in deciding how to act.

Viewed from a *functional* perspective, MU's task is *managing uncertainty* by taking appropriate actions. *Structurally,* MU has a large long-term memory of hypotheses and their supporting evidence and intermediate conclusions, a working memory of developing hypotheses, inference mechanisms for propagating the effects of evidence in working memory, and control strategies. Viewed as a *virtual machine,* MU supports knowledge engineering in terms that make sense for diagnostic tasks, such as *hypothesis* and *potential-evidence*. These terms are instantiated for specific domains by terms such as *disease*, or further instantiated as specific diseases such as *angina*.

The functional view of an architecture constrains how implementation-level primitives and techniques are *specialized* for a particular kind of problem-solving. The functional requirements of MU are that it should represent inferential relations between data, intermediate conclusions, and hypotheses. It should maintain measures of belief in all these objects, decide focus of attention (i.e., which objects to seek evidence for), and decide which evidence to seek. At the second level of Figure 7, the frames and slots of the first level are specialized as *hypotheses* and *inferential relations*, respectively. Rules are used to implement *combining functions* for evidence pro and con hypotheses. Some properties of hypotheses – a subset of their slot values – are used as *control parameters*, which help determine focus of attention. Similarly, value propagation functions are implemented via the demons and message passing, and so on. Thus, the functional view of the MU architecture tells the architecture designer how to specialize low-level implementation primitives and techniques to achieve a *virtual machine*, or shell, for a particular class of tasks.

An architecture is designed not for a specific task like diagnosing chest pain, but for a class of tasks such as diagnostic reasoning. Thus, the knowledge engineer and expert must *instantiate* architecture-level primitives for a particular application just as the architecture designer needed to specialize implementation-level primitives. Figure 8 is a structural view of MUM – a chest pain specialist – engineered in the MU architecture. Hypotheses are instantiated as *diseases* such as *classic angina*; intermediate conclusions are instantiated as *clusters* such as *exercise-induced-pain*; inferential relations such as *potential evidence* are instantiated by specific links between evidence and conclusion, such as *EKG results* and *classic angina*.

Once the knowledge engineer has decided to instantiate hypotheses as diseases, he or she can build a knowledge-acquisition interface to help elicit knowledge in the terms of the architecture. Meta-knowledge

about the terms is provided by the knowledge engineer while designing the shell, and is used by the knowledge acquisition interface to help the user build a syntactically valid and semantically consistent knowledge base. We currently have form-filling editors for all objects in Figure 8, a graphics interface for acquiring some continuous combining functions, and rudimentary consistency-checking abilities; other tools, especially for acquiring control knowledge, are in progress.

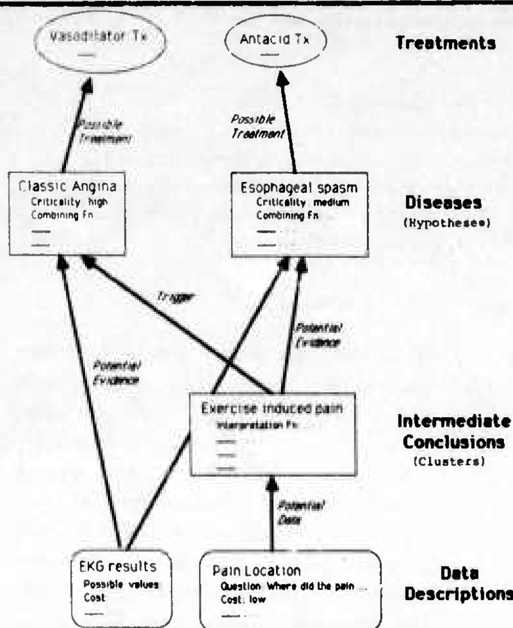


Figure 8: Fragment of the inference network for MUM

C. Conclusions

Architecture-level knowledge engineering tools have several advantages:

- One can capitalize on vertical integration of implementation-level tools at the architecture level. For example, a general-purpose frame editor and network grapher that is provided at the implementation level (such as the KREME interface (Abrett and Burstein, 1987)) can be customized as a knowledge acquisition interface for editing architecture-level constructs such as *hypotheses* and *diseases*. This is possible because the architecture-level objects are specializations of implementation-level objects.

- Software development is facilitated because architecture-level constructs — the primitive objects of the virtual machine — are represented declaratively. For example, once the *trigger* relation has been designed, one need not worry about several members of a software project trying to achieve its functionality by different implementations.[6] aims at making *every* implementation decision explicitly recorded in a language which allows a program writer to actually generate the code.

- Declarative architecture-level constructs also facilitate knowledge acquisition because metaknowledge (Davis and Buchanan, 1984) can be attached to objects to check for consistency, provide help, generate explanations, and so on. For example, a form-filling interface specialized for acquiring an instance of a disease can use a declarative description of the properties of diseases, such as the kinds of relations they have with data, to offer a menu of documented choices (Gruber and Cohen, 1986).

- Building a virtual machine at the architecture level and then a knowledge acquisition interface on top of the virtual machine defines the roles of the knowledge engineer and expert. The knowledge engineer *designs* an architecture by specializing general-purpose implementation-level tools to operationalize the constructs suited for the problem solving task, whereas the expert *instantiates* architecture-level constructs for the application domain. Virtual machine tools (shell support) assist the knowledge engineer in customizing an architecture for a particular application, and knowledge acquisition tools help the expert build, refine, and debug the knowledge base.

IV. Discussion

The hierarchy of tools discussed here reflects a power/generality tradeoff. Constructs at the implementation level are general (e.g., production systems can be configured for many kinds of problem solving) but from the standpoint of knowledge engineering they are weak. To say an object is a *disease hypothesis* is to imply much more knowledge about it than to say it is a *frame*, even though the implementation of the disease hypothesis may be no more than a frame. This added knowledge constrains the internal structure of the disease frame (e.g., values and types of slots, or the kinds of messages it can handle, etc.), constrains its relationships with other frames, and so on. Since these constraints facilitate knowledge engineering, architecture-level objects like disease frames are at the "power" end of the power/generality spectrum. Implementation-level objects, lacking constraints, are more general but correspondingly less powerful from the standpoint of knowledge engineering.

Thus, when one builds an expert system for a task, the utility of an architecture level analysis depends entirely on how much one *knows* about the task. The

[6]The EES project (Neches *et al.*, 1985)

knowledge system architecture embodies knowledge about a class of problem solving tasks – it is a virtual machine for that class – and as such facilitates system development and knowledge acquisition for problem solvers of that class. The power/generality tradeoff tells us that we can ameliorate the knowledge acquisition bottleneck for restricted classes of tasks by designing architectures and building integrated "power tools" at the architecture level.

V. Acquiring Combining Knowledge — Modifiable Combining Functions

This report presents a synthesis of two general approaches to combining evidence. When designing knowledge systems, knowledge engineers typically select one approach over the other, but each has strengths and weaknesses in terms of the ease with which knowledge can be acquired, represented, interpreted, modified, and explained. The synthesis we propose, called *modifiable combining functions* has many of the advantages of both approaches and overcomes some of their disadvantages. The basic idea of modifiable combining functions is to acquire degrees of belief for a subset of all possible combinations of evidence, then infer degrees of belief for other combinations in the set. If, in the course of knowledge engineering, a particular degree of belief is challenged, then it (and others) can be modified by an appropriate method.

A combination of evidence is a list of propositions, each with an associated degree of belief. For example, an expert system for diagnosing plant diseases has propositions like this:

((soil texture = heavy, .7)
(soil oxygen = low, .9))

That is, soil texture is believed to degree .7 to be heavy and soil oxygen is strongly believed (.9) to be low. Combinations of evidence are often found in the premises of inference rules. These rules can take two forms, called *specified* and *derived*:

specified form:

IF ((soil texture = heavy, .7)
(soil oxygen = low, .9))
THEN (water damage = yes, .8)

derived form:

IF ((soil texture = heavy, **x**)
(soil oxygen = low, **y**))
THEN (water damage = yes, *f* (**x**, **y**, **k**))

These forms suggest two general approaches to combining evidence. The specified form requires that for each combination of degrees of belief in the premise, a degree of belief is specified for the conclusion. The derived form requires a function, *f*, that derives a degree of belief in the conclusion for *any* degrees of belief in the premise. The constant **k** in the derived form represents the degree of belief that would be assigned to the conclusion if the degree of belief in the premise was 1.0, that is, the degree of belief in the inference rule itself. This quantity is implicit in the specified form.

These forms combine evidence *within* inference rules, but they have counterparts for the cases in which two or more rules draw the same, corroborating conclusion. By analogy with the specified form, degrees of belief can be acquired for each combination of corroborating rules; or a general function, analogous to *f* in the derived form, can be acquired to calculate degrees of belief for all corroborations.

Both approaches have been used in AI systems. Considering medical expert systems alone (reflecting our own interest in this area), we find knowledge in the specified form in PIP (Pauker, *et al.*, 1976), IRIS (Trigoboff, 1978), MDX (Chandrasekeran, Mittal and Smith, 1982), and MUM (Cohen, *et al.*, 1986); while MYCIN (Shortliffe, 1976; Shortliffe and Buchanan, 1975), INTERNIST/CADUCEUS (Pople, 1977) and ABLE (Patil, Szolovits, and Schwartz, 1976) use knowledge in the derived form.

In outline, we describe representations for combining functions that are closely-related to the specified and derived forms. We will discuss the tradeoffs between these approaches that motivate the idea of modifiable combining functions. We will illustrate how modifiable combining functions are generated and modified in the context of an example. Parts of the theory of modifiable combining functions have been implemented in a medical expert system (Cohen, *et al.*, 1986), but much of this report should be taken as research in progress.

A. Forms of combining functions

1. Tabular combining functions

Tabular combining functions are often represented as tables that specify degrees of belief in conclusions for each combination of degrees of belief of evidence. Figure 1 shows a tabular function that combines two

bel(E_2) \ bel(E_1)	1.0	.75	.50	.25	0	-.25	-.50	-.75	-1.0
1.0	1.0	1.0	.75	.50	0				
.75	1.0	1.0	.50	.25	0				
.50	.50	.25	0	0	0				
.25	25	0	0	0	0				
0	0	0	0	0	0				
-.25									
-.50									
-.75	0	-50	-75	-75	-1	-1	-1	-1	-1
-1.0	0	-.50	-75	-75	-1	-1	-1	-1	-1

Figure 1

pieces of evidence, E_1 and E_2, for conclusion C. In this case, degrees of belief in evidence range from -1 to +1, denoting complete disbelief and belief, respectively. A degree of belief of zero denotes ignorance; for example (soil oxygen = low, 0) means that the value of soil oxygen is unknown, either because it is an unavailable datum or because the data from which it is inferred are ambiguous. Many of the cells are blank, meaning that the expert does not consider these combinations of evidence relevant – does not expect them to arise during problem solving. From the knowledge engineer's perspective, blank cells and zero cells represent different events. A blank means that that particular combination of evidence was never considered, but a zero means it was considered and found to be uninformative. From the perspective of an AI program's interpreter, blank and zero *may* both mean that the combination of evidence is uninformative; or a blank may be used to alert the user to incompleteness in the combining function.

In tabular combining functions, degrees of belief in evidence index degrees of belief in conclusions. The combining function in Figure 1 specifies that when the degrees of belief in E_1 and E_2 are .5 and .75, respectively, the degree of belief in C is .25. Since conclusions are often used as evidence for subsequent inferences, the cells in tabular combining functions contain values that can themselves be used to index degrees of belief in other tabular functions. Tabular functions increase exponentially in size: A function for N pieces of evidence requires an N-dimensional table, similar to the *signature tables* invented by Samuel (1959).

Some important knowledge about patterns or regularities in combinations of evidence is implicit in tabular combining functions. For example, the entire upper-right quadrant of Figure 1 is blank, suggesting that no combination of positive degrees of belief in E_1 and negative degrees of belief in E_2 is meaningful. Similarly, in the lower-left quadrant we see a *threshold* on the degree of belief in E_1: the values in the table are determined by E_2 for all values of E_1 less than or equal to -.75. These regularities are easily captured by a *rule-based variant* of tabular combining functions. The two examples we just mentioned can be represented this way:

Upper-right quadrant:

$$\begin{aligned} \text{IF bel}(E_1) &\geq 0 \text{ and} \\ \text{bel}(E_2) &\leq 0 \\ \text{Then bel}(C) &:= 0 \end{aligned}$$

Lower-left quadrant:

$$\begin{aligned} \text{IF bel}(E_1) &\leq -.75 \text{ and} \\ \text{bel}(E_2) &= .5 \text{ or} \\ \text{bel}(E_2) &= .25 \\ \text{Then bel}(c) &:= -.75 \end{aligned}$$

$$\begin{aligned} \text{IF bel}(E_1) &\leq -.75 \text{ and} \\ \text{bel}(E_2) &= .75 \\ \text{Then bel}(c) &:= -.5 \end{aligned}$$

Irrespective of whether the knowledge engineer acquires tables like Figure 1, or rules as above, he or she must take care to maintain important distinctions in the domain. For example, the rule for the upper-right quadrant could be extended to account for the blank cells in the lower-right quadrant, too, by changing its first clause to "IF bel(E_1) $\geq$ -.5." While this rule describes the table, it obscures what may be an important distinction between positive and negative values for E_1.

Tabular combining functions and their rule-based variant are ways to represent combinations of evidence given in the specified form, described above. A representation that relies on both specified and derived combinations is discussed next.

2. Interpolated combining functions

Three of the four corner cells of Figure 1 represent degrees of belief in the conclusion given categorical (certain) data about E_1 and E_2 (the upper-right cell

is blank because nothing is known about it.) They can be arranged in a *categorical table* as shown below. To distinguish categorical tables from the larger ones like Figure 1, we call the latter *full* tables.

bel(E1):

bel(E2):	1	-1
1	1	X
-1	0	-1

The upper-left cell contains the degree of belief in C given that E_1 and E_2 are both true; conversely, the lower-right cell is the degree of belief in C when both are false; the 0 in the lower-left cell represents ignorance in C given that E_1 is true and E_2 is false. To reiterate, these are the corner cells of the full table in Figure 1. All other, noncorner cells in Figure 1 represent interpolations between the the values in this categorical table, interpolations due to uncertainty in E_1 and E_2. For example, the cells around the center of Figure 1 tend toward the value 0, since the center cell represents the case in which the degrees of belief in E_1 and E_2 are both zero, that is, completely uninformative. Similarly, in the lower half of the table, we see degrees of belief in C ranging from 0 when bel(E_2) = +1, to -1 for lower degrees of belief in E_2.

The full table in Figure 1 was built by hand, but full tables can also be derived by interpolating functions. Figure 2 shows the derivation of a full table by a Bayesian interpolating function. The categorical corner cells are 1.0, .95, .25, and 0.0, respectively. All other cells contain intermediate values that reflect uncertainty about the evidence. For example, when the degrees of belief in *episode* and *risk factors* are both .75, the degree of belief in the conclusion is .79, a value intermediate between the four corner points but nearer to 1.0 — its nearest neighbor — in magnitude. This table and its derivation will be explained in Section 5.

To summarize, full tables can be built by hand, by specifying the value in each cell, or specifying rules that assert the values of subsets of the cells. Alternatively, they can be derived automatically by interpolating from categorical tables. Once the decision has been made to use interpolating functions, full tables are usually not generated and stored. Instead, the values of combinations of evidence are computed as needed. However, the following Section suggests that there are advantages to keeping both forms of combining functions.

B. Comparison

Our comparison will focus on the tabular and interpolating forms of combining functions. The strengths of one often correspond to weaknesses in the other. First, tabular combining functions do not infer anything that is not stated by the expert. Most of the cells in a table are blank, meaning that the expert does not consider them to represent meaningful combinations of evidence. In theory, every nonblank cell represents a meaningful combination and every blank cell represents a meaningless one. But in practice, the sheer size of tabular functions means that some meaningful combinations of evidence are simply overlooked during knowledge acquisition. In this sense, tabular functions are *brittle*: they cannot account for all meaningful situations that will arise during problem solving.

Interpolation is clearly a solution to the brittleness problem, since the value of any blank cell can be inferred from the corners of a categorical table, or perhaps from its "nearest neighbors." The disadvantages of interpolating functions are that, unlike tabular functions, they produce values for *all* combinations of evidence in their domain, meaningful or not. Moreover, no value derived by an interpolating function is guaranteed to reflect an expert's judgment. A subtler problem is that interpolation produces a *continuous* gradient of values between the corners of the full table. But expert's degrees of belief in conclusions are unlikely to change continuously with the degrees of belief in the evidence. Thresholds are common, as illustrated by the rule-based variant of tabular functions.

Tabular functions are *locally modifiable*, meaning that a knowledge engineer can change the values of individual cells in the table with the assurance that the performance of the system will remain unchanged except in the cases of these particular combinations of evidence. This allows a combining function to be "tuned" in the normal course of knowledge base refinement: when the system presents a conclusion that the expert thinks is wrong, and the source of the error is localized to a particular cell, then that cell can be changed. In contrast, changing an interpolating function necessarily effects the values assigned to *all* combinations of evidence in its domain. Modifying an interpolating function is essentially redesigning one's inference system (Gruber and Cohen, 1987).

C. Modifiable Combining Functions

Once the knowledge engineer considers using interpolating functions, why bother to acquire full tables by hand? Why not simply acquire categorical tables, as above, and design interpolating functions to, in effect, "fill in" the intermediate values? Clearly, the two approaches are equivalent if the interpolating functions generate the same values as the expert for any combination of evidence. But there is no way to test this, other than to acquire an entire table and then compare it with the results of an interpolation function. Consequently, the knowledge engineer can take one of

two positions with respect to potential differences between interpolated values and the expert's judgment:

- The knowledge engineer can design a function that has desirable properties and assume that, if the expert's judgment is different, it is because the expert's reasoning is inconsistent or otherwise flawed.
- The knowledge engineer can design a function that is assumed to reflect expert judgment, but modify it to conform to the expert when deviations become apparent.

The first position is associated with normative models, the second with performance models. In both cases, the knowledge engineer must carefully design interpolation functions given what he knows and can assume about the evidence in a domain. In the latter case, in addition, he must have some mechanism for modifying combining functions.

Modifiable combining functions are a synthesis of tabular and interpolating functions. They are tabular functions that have most of their values derived by interpolation, but that can be modified to conform to an expert's judgment. Knowledge engineers must first acquire a categorical table and any other cells in the full table that the expert can provide. Interpolating functions ideally should fill in cells that the expert and knowledge engineer neglected to specify, with values that are likely to match the expert's judgment, but not fill in cells they intended to leave blank. If these goals are not achieved, the tabular function can be modified by one of the three mechanisms discussed below.

D. An Example

This section illustrates modifiable combining functions for two pieces of evidence from a medical diagnosis problem. Most diagnosis begins with the physician taking a *history*: asking about the patient's chief complaint, age, past medical history, and so on. Our example concerns the diagnosis of *angina* and two pieces of evidence from the history: the patient's report of an *episode* of chest pain, and whether the patient has *risk factors* for angina. Clearly, other evidence plays a role in diagnosis, but we will focus on a single rule that infers that the patient's history is consistent with angina if he or she has a characteristic episode and risk factors:

episode & *risk factors* → *angina history*

Both pieces of evidence can be uncertain because each depends on several observations. For the purpose of this example, assume that degrees of belief in *episode* and *risk factors* are subjective probabilities ranging from 0.0 to 1.0. The interpretation of P(*episode*) = 0 is "the episode is not characteristic of angina." An intermediate degree of belief, say P(*episode*) = .5, means "some aspects of the episode are consistent with angina, but other aspects are missing." The following examples illustrate assessments of degrees of belief for particular observations:

- crushing chest pain, induced by exercise, lasting a few minutes, radiating to one or both arms, accompanied by sweating and shortness of breath: P(*episode*) = 1.0
- sharp, fleeting chest pain, induced by sudden movement, not radiating: P(*episode*) = 0.0
- diffuse chest pain, came on after eating, radiating, lasting about 30 seconds: P(*episode*) = 0.5
- 60 year-old male, overweight, smoker, with high blood pressure, and two brothers with coronary artery disease: P(*risk factors*) = 1.0
- 30 year-old female, nonsmoker, not overweight, normal blood pressure, no history of heart disease in the family: P(*risk factors*) = 0.0
- 45 year-old male, smoker, not overweight, marginally-high blood pressure, uncle had coronary at age 60: P(*risk factors*) = .5

Given that *episode* and *risk factors* can be uncertain, how should a knowledge engineer acquire knowledge about the combinations of this evidence that support (or detract from) the conclusion? Degrees of belief for all possible combinations could be acquired in the *specified* form, and arranged in a tabular combining function. Alternatively, the knowledge engineer might design a combining function, *f*, and derive the degrees of belief of combinations by interpolation.

Modifiable combining functions present an intermediate alternative: the knowledge engineer acquires some degrees of belief for a subset of the possible combinations, then designs a function to interpolate the values of the rest and arranges the results in a table, then modifies the table if necessary to accord with the expert's judgment. An obvious place to begin this process is with the categorical table, from which a full table can be interpolated. Imagine the following rules, qualified by degrees of belief, are acquired from the expert:

episode & *risk factors* → *angina history* , 1.0
episode & ~ *risk factors* → *angina history* , .95
~ *episode* & *risk factors* → *angina history* , .25
~ *episode* & ~ *risk factors* → *angina history* , 0.0

These can be arranged in the following categorical table:

		P(episode)	
		1	0
P(risk factors):	1	1.0	.25
	0	.95	0.0

The knowledge engineer needs to design a function from which P(*angina history*) can be derived for values of P(*episode*) and P(*risk factors*) other than 1 and 0. Such functions reflect the knowledge engineer's assumptions about the domain. We will illustrate a Bayesian function designed under the assumption that P(*episode*) and P(*risk factors*) are independent.

The Bayesian interpolation function is derived from the rule of total probability, which says

$$P(A) = \sum_{i=1 \rightarrow n} P(A|B_i)P(B_i)$$

where $B_1, \ldots, B_n$ is an exhaustive list of mutually exclusive possibilities. For our example, A is the conclusion *angina history* and $B_1, \ldots, B_n$ is

episode & *risk factors*
episode & ~ *risk factors*
~ *episode* & *risk factors*
~ *episode* & ~ *risk factors*

Then, P(*angina history*) can be derived for any degrees of belief in *episode* and *risk factors* as follows:

$$\begin{aligned} P(a) &= \\ &P(a \mid e \,\&\, r)\, P(e \,\&\, r) \\ &+ P(a \mid e \,\&\, \sim r)\, P(e \,\&\, \sim r) \\ &+ P(a \mid \sim e \,\&\, r)\, P(\sim e \,\&\, r) \\ &+ P(a \mid \sim e \,\&\, \sim r)\, P(\sim e \,\&\, \sim r) \end{aligned}$$

where *episode, risk factors* and *angina history* are abbreviated *e*,*r*, and *a*, respectively.

The values of the conditional terms in this expression have already been acquired from the expert and are recorded in the categorical table (e.g., $P(a|e \,\&\, r) = 1.0, P(a|e \,\&\, \sim r) = .95 \ldots$). The knowledge engineer now must decide whether to acquire the other terms in the expression $P(e \,\&\, r)$, $P(e \,\&\, \sim r)$ ... This effort can be avoided by assuming that *e* and *r* are independent, in which case $P(e \,\&\, r) = P(e)P(r)$, and

$$\begin{aligned} P(a) &= \\ &P(a \mid e \,\&\, r)\, P(e)P(r) \\ &+ P(a \mid e \,\&\, \sim r)\, P(e)P(\sim r) \\ &+ P(a \mid \sim e \,\&\, r)\, P(\sim e)P(r) \\ &+ P(a \mid \sim e \,\&\, \sim r)\, P(\sim e)P(\sim r) \end{aligned}$$

or,

$$\begin{aligned} P(a) &= \qquad (1) \\ &P(a \mid e \,\&\, r)\, P(e)P(r) \\ &+ P(a \mid e \,\&\, \sim r)\, P(e)[1 - P(r)] \\ &+ P(a \mid \sim e \,\&\, r)\, [1 - P(e)]P(r) \\ &+ P(a \mid \sim e \,\&\, \sim r)\, [1 - P(e)][1 - P(r)]\ . \end{aligned}$$

Figure 2 illustrates a full table containing the values of P(a) derived by this function from these categorical values

$$\begin{aligned} P(a \mid e \,\&\, r) &= 1.0 \\ P(a \mid e \,\&\, \sim r) &= .95 \\ P(a \mid \sim e \,\&\, r) &= .25 \\ P(a \mid \sim e \,\&\, \sim r) &= 0 \end{aligned}$$

and letting P(e) and P(r) range through the values 0, .125,.25, .375, .5, .625, .75, .825, and 1.0.

	P(episode)								
P(risk factor)	1.0	.875	.75	.625	.50	.375	.25	125	6
1.0	1.0	.91	81	72	63	53	44	34	.25
.875	99	90	80	70	.61	51	41	32	22
.75	99	89	79	69	59	49	39	29	19
.625	98	88	78	67	57	47	36	26	.16
.50	98	87	76	66	55	44	34	23	13
.375	97	86	75	64	53	42	31	20	09
.25	96	85	74	63	51	40	29	18	06
.125	96	84	73	61	49	38	26	.15	03
0	.95	83	71	59	48	36	24	12	0

Figure 2

The Bayesian function (1) is an example of what is sometimes called Jeffrey's rule (Shafer, 1981; Shafer and Tversky, 1985). In such a design the conditional probabilities $P(a|e\&r)$ ... reflect the expert's heuristic judgments based on previous cases of angina. In contrast, the unconditional probabilities $P(e\&r) = P(e)P(r)$ ... reflect knowledge about the individual patient who is currently being diagnosed. This is because, to calculate $P(e\&r)$, we assume that the probability of an angina episode is independent of whether one is at risk. This is true for an individual patient: for *this* patient the probability of an angina episode *is* independent of the probability that he is at risk. *This* patient either has risk factors in addition to his angina episode or he doesn't. Thus, the decision to design a Bayesian function for which $P(e\&r) = P(e)P(r)$ implies that the expert's knowledge of patients in general is dominated by his knowledge of the probabilities $P(e)$ and $P(r)$ for the individual patient.

Consider how this assumption might lead to a conflict with the expert's judgment. In general,

$$P(e \,\&\, r) = P(e|r)P(r)$$

or, if $P(e)$ and $P(r)$ are independent, then $P(e|r) = P(e)$ and

$$P(e \,\&\, r) = P(e)P(r)$$

But this seems wrong because, in general, the probability of an episode given risk factors is higher than the probability of the episode, or

$$P(e|r)P(r) > P(e)P(r)$$

Consequently, in some cases, $P(e \,\&\, r)$ will be too low, and so the value of $P(a)$ denoted by (2) will be too low, as well. For example, according to Figure 2, if $P(episode) = .5$ and $P(risk\ factors) = .75$, then $P(angina\ history) = .59$. But in the course of testing a system, the expert may challenge this result. He may say that if there is moderate evidence of an episode and strong evidence of risk factors then the probability of angina history should be much higher, say, 0.75.

What should the knowledge engineer do in this case? If he is relying exclusively on interpolating functions then he has 3 options:

1. insist that the expert's judgment is flawed
2. change the categorical table
3. change the interpolating function

The first is practical only if the knowledge engineer is confident that the assumptions that underlie his interpolating function are reasonable. The other two have global effects on all the numbers in the table, not just the few the expert criticized. Thus, in fixing the immediate problem the knowledge engineer could introduce new ones. Knowledge engineering often extends over a period of months, and the knowledge engineer relies on a kind of monotonicity — the idea that adding new knowledge to a system will not make it perform differently on the majority of previous cases. Changing the categorical table has ramifications only for the inference rule with which it is associated, but changing an interpolation function will change the degrees of belief of all the conclusions derived by that function — potentially every conclusion previously derived by a knowledge system.

If the knowledge engineer *does* decide to change the function, how should he go about it? We could change (1) by eliminating the independence assumption and acquiring the required conditional probabilities from the expert. Or, we might design a completely new Bayesian function that exploits the causal associations between the evidence and the conclusion (Pearl, 1986). Or, we could conclude that a belief-function design better characterizes the relationship between the evidence and the conclusion (Shenoy and Shafer, 1986)[7]. Many interpolation schemes are possible, but most of them are mathematically complicated, or computationally expensive, or require many more numbers than the expert can accurately provide. The Bayesian function above (1) is very simple and requires few numbers. Its major deficit is that, in a few cases, it produces numbers with which the expert disagrees.

P(risk factor) \ P(episode)	1.0	.875	.75	.625	.50	.375	.25	.125	0
1.0	1.0	91	81	72	63	.53	44	.34	.25
.875	99	90	80	.70	61	.51	.41	32	.22
.75	99	89	79	69	59	49	.39	29	19
.625	.98	.88	78	67	57	47	36	26	.16
.50	.98	87	76	66	55	.44	34	23	.13
.375	.97	.86	75	64	.53	42	.31	20	.09
.25	96	85	74	63	51	.40	29	.18	.06
.125	.96	84	.73	61	49	.38	26	.15	.03
0	.95	.83	71	59	48	36	.24	12	0

Figure 3

If the knowledge engineer does not rely exclusively on interpolating functions to calculate degrees of belief, then he has another option besides the three listed above: He can simply change the values that the expert says are wrong and store the new values in a tabular form that overides the derived values. The idea of modifiable combining functions is, in essence, to use simple interpolating functions to derive full tables from categorical tables, then, when the expert criticizes a derived degree of belief, to simply change it. This is shown in Figures 3 and 4. In Figure 3, the expert identifies a block of cells with values that are too low, for the reasons we discussed earlier. Figure 4 shows one possible modification.

In sum, modifiable combining functions offer three methods for representing expert judgments about combinations of evidence. First, individual cells, or blocks of cells in a derived tabular function can be changed. Second, the value in the categorical table can be changed. Third, and as a last resort, the interpolating function can redesigned.

[7] The authors are currently working on a formulation of modifiable combining functions based on belief functions. The formulation is preliminary, and space limitations preclude introducing it here.

P(episode)

P(risk factor)	1.0	.875	.75	.625	.50	.375	.25	.125	0
1.0	1.0	91	81	.75	.75	50	50	34	.25
.875	99	90	80	75	.75	50	50	32	22
.75	99	89	79	75	75	50	50	29	19
.625	98	88	78	75	75	50	50	26	16
.50	98	87	76	66	55	44	.34	23	13
375	97	86	75	64	53	42	31	20	09
.25	96	85	74	63	51	40	29	18	06
.125	96	84	73	61	49	38	26	15	03
0	.95	83	71	59	48	36	24	12	0

Figure 4

E. Conclusion

Modifiable combining functions share many of the advantages of tabular and interpolating functions while avoiding some of their disadvantages. The information burden of tabular functions is reduced because the full table is derived by interpolating from the values the expert *can* provide. (One natural basis for the interpolation is the categorical table, but others are possible.) The brittleness of tabular combining functions, especially multidimensional ones, is overcome. Simple interpolating functions can be used, requiring relatively few numbers from the expert. Then, any values in the derived full table can be overridden by the expert's judgment. Discontinuities can easily be expressed in the rule-based variant of tabular combining functions. When an interpolating function fills in cells that the expert thinks should be blank (meaningless), the function can be modified accordingly. All modifications to cells are local in the sense that they affect the system's performance for combinations of evidence represented by those cells only. But if global modifications are appropriate, if *all* the values in a modifiable combining function seem wrong to the expert, then the knowledge engineer can first consider modifying the categorical table (or any other set of points used for interpolation) and then consider modifying the interpolation function.

Currently, we are acquiring tabular combining functions for a medical expert system (Cohen *et al.*, 1987) and a plant pathology system. They are represented as rules, as discussed above. We have built interfaces for acquiring and modifying these rules, and we have almost completed a graphic interface for representing them in tabular form.

VI. A Notation for Representing Strategies

Our current approach to encoding strategies in MU planners has been to divide the decision space into phases whose applicability is decided by a test on the state of the network. Each phase then provides a means for deciding the current focus. Potential evidence for the hypotheses is obtained and each phase may further specify filtering criteria to form a subset of these actions to actually undertake. At this point though, we don't do anything very intelligent. We have either considered these remaining actions as a "plan" and just executed them, as in the latest version, or undertaken one and seen what changes occurred; cycling through the phases again to see which one is now most appropriate.

Looking at some of the strategies/tactics that we have obtained from the medical domain, it appears that a richer representation at this action level would allow us to implement more sophisticated plans. In discussing possible orderings of actions we found many of the operations involved forming and manipulating sets. Further, we recognized the existence of these sets by Feature predicates on the possible actions. It became helpful to have a notation to express these sets and their orderings and this is what we are currently working on.

Consider that we have already selected a focus set of hypotheses and have obtained actions relevant to them and done some type of filtering on those actions. We have a plan which may be applicable at this point which says - "if there are two actions we can take w.r.t. a hypothesis, and one has high cost and high diagnosticity, while the other has low cost and low diagnosticity - then perform the action with low cost first and if it has a positive effect do the higher cost one". We need to compare our actions on two different features -

Cost and Diagnosticity

We know MU will allow us to define these and other features so we can write a general expression to denote this situation:

$$\exists X, Y, H[F_1(X, \text{Value}, H) \;\&\; F_2(X, \text{Value}, H) \;\&\; F_1(Y, \text{Value}, H) \;\&\; F_2(Y, \text{Value}, H)]$$

Where F_1 and F_2 are features and X and Y are actions, with *Value* being the current value that action has for that particular feature, and H the hypothesis under consideration.

If actions A and B are the variables that we place into an instantiation of the above form, say -

Cost(A, High, $Hypo_1$) &
Diagnosticity(A, High, $Hypo_1$) &
Cost(B, Low, $Hypo_1$) &
Diagnosticity(B, Low, $Hypo_1$)

Then we want to say something like - {B : A, C}, Where ":" is a shuffle operator and means:

Do A IF Result(B) satisfies the condition C
and
$C(B) = |\ \Delta$ belief$(H)| > 1$ &
belief$(H) \geq$ Precond(A)[8]

That is do B then do A only if the result of B is a change in our belief of $Hypo_1$. This will require that we note $Hypo_1$'s belief level before taking action B and after, but it does not mean that we must exit this phase and perform other computation. What we are trying to do is order our applicable actions by the dependencies that may exist between them.

We are ordering by features (whose value we can easily access) and since our actoins are taken to modify other features (such as degree of belief) we can also express and verify the conditions in the dependencies. We might imagine a set of these dependencies:

{ {A B} {C D E} {F G} }

Where we could choose to carry out either -

- One of the sets
- All of the sets, in order or randomly
- Dependencies similar to those inside the sets

However, inside the sets themselves it would appear that the only meaningful operation would be some form of dependencies. For example, we have the ability right now in MU to order our final set of actions by feature(s). But if we are going to perform them all anyway what is point. And re-evaluating our phase state isn't always called for. What we want is to group our actions into meaningful steps which indeed do form a plan.

Graphically we have something like this -

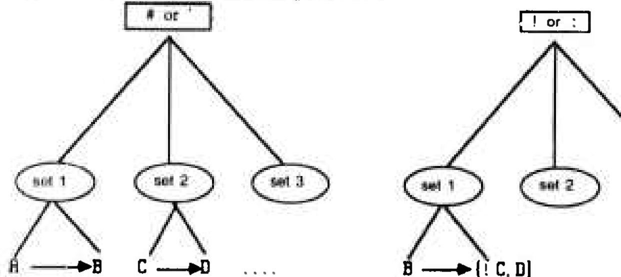


Figure 9: Ordering of action sets

In the strategy mentioned earlier, we might first want to see if there are any actions that have high diagnosticity and low cost and ask those first. If they do not effect our belief in the hypothesis then we will check to see if the example situation is present. We may also have the case where none of our actions differ on some feature. We may choose to order them by another. For instance if all our actions have the same costs we may decide to order them by potential diagnosticity. Then if one of them succeeds in achieving its maximum potential we can stop because we know none of the others can do better, (this does ignore the fact that the further actions might give us conflicting negative information).

With sets of possible actions, each set containing some dependencies internal to them, and possible orderings existing between the sets. We can denote these, borrowing a bit from work by P.Bates on event descriptions, by

\# - Shuffle, do all the sets, order is not important

' - Inorder, do all the sets, in the order specified

| - Do one, any one of the sets - presumably if you only want one in particular you know what it is and won't have a set

: - Dependency, do one set, if a condition is met/not met do another

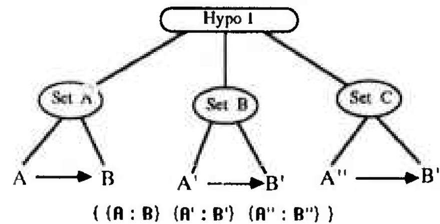


Figure 10: Dependencies of actions

The dependencies in a set are shown by the arrow, again we may not have dependencies between the actions we decide to undertake, but when this is the case we will not have this same grouping of sets. Our dependencies are of the form -

Do A, if the result of A is such and such then either:
Do B
Do either B or C
Do B then C

So a possible phase could have the following instructions once a focus has been found and some preliminary filtering of actions has been done.

1. If all actions are equal on some feature like cost, they could be ordered by another feature, diagnosticity, and performed with dependencies or in the usual case where we are trying to increase our belief state we could go on until our state was equal or greater than the potential state achievable by the remaining actions themselves.
2. If some actions are the best we could hope for - Cost(Z,low) and Diagnosticity(Z, high) then we will perform these actions. If this isn't the case or if after this we still have actions whose potential is greater than our currtent state of belief then we can go on to -
3. Identify the situations mentioned earlier for which we have a tactic for organizing a plan.

Suppose we have three sets of actions -

If these are formed of a pair of actions whose dependencies are determined as in the cost, diagnosticity example; we can express several plans -

{ :{**set A**} {**set B**} } - Do set A, If condition (in this case increase in belief) not met then do B
Gives AB or AA' or AA'B'

{ #{**set A**} {**set B**} } - Do both sets but order unimportant
Gives AA' or AA'B' or ABA' or ABA'B' or with primes first

{ '{**set A**} {**set B**} } - Do set A then set B
Gives the first case of example above

{ !{**set A**} {**set B**} } - Do set A or set B
Gives A, AB or A', A' B'

And of course....

{ : {**set A**} {!{**set B**} {**set C**}} } - Do set A and if no change do set B or set C

{ : {**set A**} {{**set B**} : {**set C**}}} - Do A if no change do B if no change do C

Most likely if we are concerned with sets for one hypo we are looking at -

{: {! **A A' A"**} {! **B B' B"**} } - Do one of the first set, if condition met do one of the second set
or
{: {: **A A' A"**} {! **B B' B"**}} - Do A, if condition not met do A'... then do one from second set

In fact you can come up with a lot of improbable combinations, however, you can also realize some rather practical ones you hadn't considered before. The motivation is to be able to express the situations for which strategies we have obtained (or made up) apply. We do this by writing general forms with variables allowed to specify -

- features - like cost, degree of belief
- their values
- the applicable actions - ekg, stress test
- the hypothesis under consideration to which the actions apply

Again, for example, consider forming a set of actions with two criteria -

$\forall$ X [F_1(X, Value, $Hypo_1$) &
F_2(X, Value, $Hypo_1$)]

and we use *time* for F_1 and *short* for its value and *cost* for F_2 and *medium or less* for its value. We form a set of actions which apply - {C D E }, and order them - {: C D E}. That is, do C if condition not met do D... Now we make another set by testing the remaining actions with the same pattern as above but with no cost considerations. This becomes {: F G}. And now we order the two sets {: {: C D E} {: F G}}. Our strategy says we are interested in actions which have a small time cost associated with them but we would still like to consider other costs initially. If these initial actions are insufficient then we are willing to incurr any of these other costs for the sake of short time.

There are some possibly large computation costs here. We have several hypos under consideration at one time and many actions can affect more than one hypo. In the medical domain this doesn't appear to be too bad a problem in reality. The numbers are manageable. We can deal with one hypo at a time when constructing our trees and we can identify the values of the features in question in a straightforward manner. Whether the small numbers are true in other domains (where similar grouping of actions occurr as tactics) remains to be seen.

We can however have the situation

```
  Hypo_1          Hypo_2
    |               |
  {set A}         {set B}
   :E F            :F G
```

Where we do action F for $Hypo_2$ after deciding not to do it for $Hypo_1$ because the result of E did not meet the specified condition - explanation is important.

A further consideration of this approach is to use a similar representation to obtain the sets of hypos which form the focus also.

VII. Evaluation of the GRANT System

GRANT's evolution from a small, prototype system (Cohen, *et al.*, 1985) to the present has given us the opportunity to compare performance as the system has been scaled up, and to consider the potentials and pitfalls of developing other GRANT-like systems. This section discusses a battery of tests on the current system.

The primary measures of GRANT's performance are *recall* and *fallout rate*. (A third statistic, *precision*, is 1.0 - fallout.) Recall is the percentage of all the agencies accepted by the expert that GRANT found, and fallout is the percentage of all the agencies found by GRANT that were judged good by GRANT but bad by the expert:

recall rate:

$$\frac{\text{num. of agencies judged good by GRANT, good by expert}}{\text{num. of agencies judged good by expert}}$$

fallout:

$$\frac{\text{num. of agencies judged good by GRANT, bad by expert}}{\text{num. of agencies judged good by GRANT}}$$

To calculate recall and fallout for a proposal, we need to generate a list of agencies from which the expert can select the ones that are likely to fund the proposal. One method would be to have the expert rank all 700 agencies in the network for each proposal, but this would be exhausting. Instead, GRANT is run in a minimally-constrained, spreading activation search that reports all agencies found within a given "distance" from each research topic in the proposal. This is called breadth-first (BF) search[9]. For each proposal, we first run a BF search then ask our expert to classify the agencies it finds as good or bad. Since the search is blind, many of the agencies are bad; that is, unlikely in the expert's judgment to fund the proposal. Then we run GRANT in an *endorsment constrained* mode called EC search, avoiding negatively-endorsed pathways and favoring positively-endorsed ones. It finds a subset of the agencies discovered by BF search. Ideally, it should find all and only the agencies ranked as good by the expert, but in practice it fails to find some of the good agencies (called *misses*) and finds some bad ones (called *false positives*). GRANT's miss rate tends to be very low, so we will be concerned primarily with the relationship between the fallout rate and recall rate.

[9]Completely unconstrained BF search finds all agencies in the network, each by dozens of different paths, and requires hours of CPU time on a TI Explorer Lisp Machine. The data presented here are for a modified version of BF search that avoids nodes with extremely high fan-out and prunes paths longer than 4 links.

The following tests were all performed on a set of 27 proposals, representing the interests of a diverse group of first-year faculty at the University of Massachusetts. The first test was designed to probe the utility of endorsement-constrained search. We compared EC and BF search with a third mode called *unconstrained keyword search* (UKW). It finds all agencies that share a common research interest with a proposal. It is implemented as a search for all agencies exactly 2 links distant from the proposal. For example, if a proposal and an agency share the common interest *dandelions*, then each will be linked to that node by, say, a SUBJECT link. The two-link

SUBJECT : *dandelions* : SUBJECT-OF

path connects the agency and the proposal via the common term dandelion; and, in general, any two-link path between an agency and a proposal indicates a shared term. UKW search is thus a simple *keyword* search, since it finds only those agencies that share terms with proposals. The relevant statistics for UKW, EC, and BF searches are shown in Table 1.

	UKW	EC	BF
fallout rate	64%	71%	94%
recall rate	44%	67%	100%
number of agencies found	164	406	2145
number of false positives	106	207	2013
number of hits	58	88	132
number correctly rejected	0	111	0

Table 1. Statistics from UKW, EC, and BF searches.

EC search has a higher recall than UKW and a lower fallout rate than BF. Its fallout rate is typically higher than UKW because it subsumes UKW: it finds all the agencies that UKW finds, then finds some more by exploiting semantic relations. Let us consider the utility of this additional search.

Of the agencies found by GRANT for the 27 test cases, the expert thought that 132 would be likely to fund their respective proposals. UKW found just 44% of these. To find the rest, it is necessary to exploit semantic relationships between the terms used in research proposals and agency descriptions. EC search found 67% of the agencies judged good by the expert. It found 242 more agencies than UKW search: 30 hits, 101 false positives, and 111 correctly rejected. So in the regions of the network that cannot be explored by keyword UKW search, EC search found 40% of the agencies it should, and incorrectly accepted 101 agencies, for a "marginal" fallout rate of 42% . In contrast, BF search found almost all the agencies judged good by the expert, but at a cost of a 94% fallout rate.

In practice, GRANT's mode of operation is EC search.

It is preferred to UKW search because it finds more agencies, and to BF search because it has higher precision. BF search finds about 80 agencies per proposal at a precision of 6% — only 1 agency in 20 is truly worth pursuing. EC search reports fewer agencies (15 per proposal), has a better level of precision (29%) than BF search, and has an acceptable, intermediate recall rate (67%).

Since EC search subsumes UKW search, it also inherits a significant fallout rate. The fallout rate for agencies found by keyword UKW search is 64%, but the marginal rate for those agencies found by additional semantic matching is just 42%. Clearly, path endorsements can increase precision. But their utility is obscured to some extent by the fact that EC search "starts off" with the 106 false positives found by UKW search. With this proviso stated, we now explore how to increase the recall and precision of EC search.

Our experiments are designed to address two general hypotheses.

- GRANT's performance is due to its path endorsements.
- GRANT's performance is affected by the *structure* of its network, including the lengths of pathways between proposals and agencies, and the degree of interconnection between nodes.

A third hypothesis is that GRANT's performance is affected by how its language of links is used to encode the interests of agencies. Since many people worked on GRANT's knowledge base, we were concerned that knowledge was encoded inconsistently. We calculated several statistics that measure consistency, but we did not find significant or even suggestive correlations of these measures with fallout rates. We cannot conclude that inconsistencies have no affect on GRANT's performance, because our measures of consistency may not be sufficiently sensitive. But we have found much stronger evidence for the other two hypotheses.

Structural Factors in Recall and Precision. We first calculated the recall and fallout rates as a function of the distance between proposals and agencies in EC search (Table 2). As noted, at distance = 2 EC has the same fallout rate as UKW search, which finds all agencies within two links of the proposal. Extending the search one more link increases the recall rate substantially (from 42% to 70%) and also raises the fallout rate somewhat. Interestingly, extending the search further has almost no affect on the recall rate but does increase the fallout rate. This suggests that endorsement-constrained search as implemented here offers most advantage when finding agencies based on a single semantic relationship between a term used in the proposal and a term used in the agency description. Increased fallout limits the utility of longer chains of relations.

length	fallout rate	recall rate
less than 3	64	42
less than 4	73	70
less than 5	78	69

Table 2. Recall and fallout rates for searches along pathways of different lengths.

The structural feature of GRANT's network that accounts for most variance in recall rate and fallout rate is the *branching factor* of nodes, that is, the number of links that connect nodes. In an experiment reported in (Kjeldsen and Cohen, 1987) we found that the fallout rate was correlated with the average branching factor of pathways to agencies. Average branching factor is the average of the number of links emanating from each node on a pathway. It is a measure of the "density" of the network in the vicinity of the pathway. We expected dense areas of the network to have low fallout rates relative to recall rates, since there are more nodes per agency in dense areas, and thus more basis for discriminating good agencies from bad ones. Table 3 shows the percentage of the false positives found along pathways with low, medium, and high branching factors.

EC Search	average branching factor		
	2 - 7	8 - 15	> 16
% hits	20.3	40.6	39.1
% false positives	8.4	36.9	54.6

UKW Search	average branching factor		
	2 - 7	8 - 15	> 16
% hits	30.7	55.1	14.1
% false positives	8.4	37.3	51.8

Table 3. Hits and false positives for EC and UKW search, distributed by average branching factor.

Contrary to our expectations, the majority of false positives were associated not with low branching factors but rather with high ones. For EC search, 54% of the false positives were found on paths with an average branching factor greater than 16. For UKW search, 51% of the false positives were associated with high branching factor; furthermore, only 14% of the hits were found in these areas. We looked at the test cases individually to try to explain this result. Many of the false positives were associated with nodes with high fan-out, such as "animal" and "location." We believe that such nodes are relatively general, that their fan-out is due to their many specializations. To say an agency is associated with one of these general nodes is to say very little about its interests, so agencies found via these nodes are more likely to be false positives.

These data seem to suggest that we could increase GRANT's precision by pruning agencies associated with general nodes. In fact, this is an artifact of the way we calculate precision. We could certainly reduce the number of false positives this way, but we would also reduce the number of agencies GRANT finds, and so would have little effect on the fallout *rate*. Moreover, since the denominator of the recall rate is constant — the number of agencies judged good by the expert — pruning agencies can only reduce the recall rate. Clearly, false positives are associated with higher branching factors. However, the key to improving precision is not to prune agencies, but to restructure the network so that it has fewer pathways with high branching factors, that is, fewer nodes that represent very general concepts. For example, the current network defines *dandelion* and *tomato plant* as instances of the *plant* node, though they are obviously different kinds of plants. The distinction could be made by defining dandelion as an instance of a *weed* and tomato plant as a *domestic plant,* but because these nodes are not in the network, the fan-out of *plant* is higher than it should be and *dandelion* and *tomato plant* are not adequately discriminated.

The statistics in Table 3 suggest that the "ideal" branching factor is less than 16. Another experiment was needed to pinpoint the ideal more precisely. Starting with the list of agencies found by the EC search and reported in Table 1, we ranked the agencies by their branching factors, and recalculated the recall rate and fallout rate for each successive level of the ranking. That is, we superimposed a ranking by branching factor on the list of agencies found by EC search and asked about the recall rate and fallout rate of all agencies that had, first, low branching factor, then those that had higher branching factor, and so on. (For reasons discussed below, we used the branching factor of the last node on a pathway instead of the average branching factor over all nodes on a pathway.) The results are shown in Table 4.

These data suggest that disproportionate numbers of false positives are associated with low and moderately high branching factors. At the lowest level (branching factor of 3 or less) there are few false positives (26) and hits (20) because few nodes have such low branching factors. At the next level we consider agencies found via nodes with branching factor of 7 or less. 47 are false positives, an increase of 81%, and 25 are hits, an increase of 25%. Thus, fallout rate increases faster than recall rate for nodes with relatively low branching factors. When nodes with higher branching factors (10 or less) are considered, fallout rate increases by 157% and recall rate by a comparable 140%. However, adding agencies that are found by nodes at the next level of branching factor (13 or less) increases fallout rate by 55% but increases recall rate by only 15%. The rates then increase proportionately for higher levels of branching factor.

The greatest increase in recall and fallout occurs when we add the agencies found via nodes with branching factors between 8 and 10. Moreover, the numbers of hits and fallouts increase by roughly the same amount in this area (about 150%). In contrast, false positives increase more rapidly than hits at low (3 - 7) and moderately high (11 - 14) branching factors. This suggests that the "ideal" branching factor is between 8 and 10, and supports the hypothesis that recall and fallout rate are correlated with the generality – as measured by branching factor – of nodes. As mentioned above, we used the branching factor of the last node on a pathway – the one "nearest" to the agency and "furthest" from the proposal – to produce the data in Table 4. We reasoned that very specific nodes, those with low branching factor, would rarely be part of an agency description, and so would not be associated with many hits. On the other hand, as we argued above, nodes with very high branching factors are too general to represent the interests of an agency unambiguously, and so would be associated with high fallout rates.

Agency is counted as "good" if the branching factor is:	fallout rate	recall rate	number of FPs	% change number of FPs	number of hits	% change number of hits
any number	73	63	219	2	82	1
16 or less	73	62	215	14	81	17
13 or less	73	53	188	55	69	15
10 or less	67	46	121	157	60	140
7 or less	66	19	47	81	25	25
3 or less	58	15	26		20	

Table 4. Fallout and recall rates from ranking agencies by branching factor.

The primary implication of these results is that knowledge engineers for GRANT-style systems should ensure that the definitions of new terms are as specific as possible. For example, the knowledge engineer should define a new plant in terms of the most specific possible subclass of plants, or perhaps create a new subclass, rather than linking the new plant to the general *plant* node. Currently, GRANT is programmed to avoid nodes with extremely high fan-out. An alternative would be to alert the knowledge engineer to them during the development of the knowledge base, to fix the problem before it arises. Then, any remaining nodes with high fan-out almost certainly denote concepts that are too general to be useful, and endorsements could be designed to avoid them, or to give them a low rank.

Endorsements as Factor in Recall and Precision. Our second hypothesis is that although the representation language for the network is probably sufficient to encode the meaning of research proposals and agency descriptions, these representations are not being exploited by endorsement-constrained search. Several findings support this hypothesis. In (Kjeldsen and Cohen, 1987) we reported that just three path endorsements accounted for 85% of the hits but the same three led to 42% of the false positives. The culprits were:

- SUBJECT : SUBJECT-OF
- SUBJECT : SUBJECT-OF : SUBJECT-OF
- OBJECT : SUBJECT-OF

Despite the fact that 48 distinct relations are used in the network to connect concepts, just 3 (SUBJECT, OBJECT, and SUBJECT-OF) were sufficient to find the majority of hits and a sizeable portion of false-positives. This is partly due to the relative frequency of these links in the network: they are very common and so support a disproportionate number of path traversals. However, our data suggest that the reliance on these links is not due entirely to their frequency, and that intelligent use of other links could increase recall rate.

We measured the frequency with which different links were used to represent agency descriptions. These data are shown in Table 5. As expected, SUBJECT, OBJECT, and FOCUS were most common, but WHO-FOR and LOCATION were not infrequent.

Link	Number of uses in agency definitions	Number of uses as last link of endorsements
subject	513	19
object	258	10
focus	238	17
who-for	124	2
location	80	0
dv	30	8
iv	20	5
rv	18	5

Table 5. Number of times each link is used to define agency interests, and number of times it is the final link in an endorsement.

Agency is counted as "good" if it is found by an endorsement classified as:	fallout rate	recall rate	number of FPs	% change in number of FPs	number of hits	% change number of hits
very-likely	55	18	28	425%	23	78%
likely or very-likely	73	42	147	41%	54	59%
maybe, likely, or very-likely	71	67	207	4%	86	0%
unlikely, maybe, likely, or very-likely	72	67	216		86	

Table 6. Fallout and recall rates from ranking agencies by final link.

However, these latter links were almost never traversed to find agencies: Table 6 shows the results of using the last link in a pathway (the one closest to the proposal) to rank the agencies found by EC search. If SUBJECT and OBJECT are the only links that GRANT is allowed to traverse, then it finds 74 hits and 179 false positives. It finds an additional 15 hits when it is also allowed to traverse FOCUS. But, remarkably, allowing it to traverse *any* link results in only 2 more hits: Most of GRANT's hits are found by following SUBJECT, OBJECT, and FOCUS links into an agency. Although WHO-FOR and LOCATION are used quite often to define the interests of agencies, they are not used to find the agencies. This is not surprising, since WHO-FOR and LOCATION are the final link in only 2 path endorsements. But it does suggest that using these and other links judiciously could increase GRANT's recall rate. In general, these results stress that path endorsements must reflect the conventions for representing concepts.

To get a more complete picture of the utility of GRANT's path endorsements we would perform "ablation studies" — removing path endorsements one at a time to see how they affect recall and precision. Unfortunately, an exhaustive analysis of all endorsements would require weeks of computer time. Instead, we grouped the path endorsements and assessed the effects on performance of removing these classes. Every path endorsement is assigned to one of five classes that reflects the subjective probability that an agency found by that endorsement would fund the proposal. The classes are trash, unlikely, maybe, likely, and very-likely. We used these classes to rank as "good" or "bad" the agencies found by EC search, then recalculated recall and fallout rates for each rank. The results are shown in Table 7.

Agency is counted as "good" if the last link in a pathway is:	fallout rate	recall rate	number of FPs	number of hits
SUBJECT or OBJECT	71	57	179	74
SUBJECT, OBJECT, or FOCUS	72	68	228	89
ANY LINK	73	70	251	91

Table 7. Fallout and recall rates from ranking agencies by class of path endorsements.

When only *very-likely* endorsements are allowed, the numbers of hits and false positives are low (23 and 28, respectively). Adding in agencies that are found via paths with *likely* endorsements increases the number of false positives by over 400% to 147. This seems an excessive price to pay for the 78% increase (from 23 to 54) in the number of hits. In contrast, adding in agencies with *maybe* endorsements increases the number of hits by 59% and increases false positives by a significantly lower amount, 41% . (The main reason for the increase in recall is that FOCUS links are used in a preponderance of *maybe* endorsements, and are infrequently used in *likely* or *very-likely*. We saw in Table 5 that the FOCUS link is used frequently in defining agencies, and in Table 6 that inclusion of the FOCUS link increases GRANT's recall rate.)

Clearly, GRANT's fallout rate could be improved by refining its *likely* endorsements. The improvement in performance due to adding *maybe* endorsements — specifically those dealing with FOCUS links — convinces us that it is possible to add endorsements that will increase recall and precision simultaneously. Table 5 suggests that these endorsements should exploit WHO-FOR and LOCATION links, which are used to define agencies but are rarely traversed to find them. We are currently designing new endorsements, though they will have to be tested on a new set of proposals to ensure that they are not simply "tuned" to the current test cases.

The main conclusion of our work is that constrained spreading activation finds agencies based on semantic relations, with reasonable recall and precision, that would not be found by simple keyword search. From a pragmatic standpoint, the Office of Research Affairs at the University of Massachusetts prefers GRANT for several reasons to the database program that it used previously. GRANT is more efficient. A session takes just a few minutes: the proposal is coded, GRANT runs a search, a list of 15 agencies (on average) is returned, and the user sorts them to find 2 or 3 that are ideal for the client. In contrast, a similar search takes about 2 hours with the old keyword database system, in part because the dozens of agencies returned by the old system must be carefully sorted (its precision is only about 5%). GRANT's performance is well-suited to the funding domain because researchers rarely send a proposal to many agencies, but several agencies will typically fund a piece of research. Thus, GRANT's relatively low precision (29%) is not bothersome because a search returns relatively few agencies — ample to find 2 or 3 for the client but few enough to sort quickly. And since a proposal can potentially be funded by several agencies, GRANT's recall rate (67%) is sufficient to find enough good candidates for the user.

GRANT was designed to have the advantages of human associate memory but to be more reliable. It is difficult to evaluate any system on such vague criteria, but the experiences of GRANT's users are suggestive. At first, they expected GRANT to accelerate their processing of "easy" cases. They found instead that easy cases were those that could be answered from memory, and that GRANT is most useful for difficult cases — those for which no agencies come to mind. Apparently, GRANT's associative memory finds plausible semantic connections between topics in proposals and agencies that human funding advisors either forgot or never knew.

We are considering other applications of constrained spreading activation. A straightforward extension of GRANT is to run the system "backwards," taking as input an agency's request for proposals (RFP) and searching for the appropriate faculty members to receive the RFP. The research interests of many of the faculty at the University of Massachusetts have been encoded for this purpose. Another goal is an intelligent index for a major reference book, since GRANT is adept at inferences of the form "if a researcher (or reader) is interested in topic **X** then he or she is likely to be interested in a related topic **Y**." Other potential applications are literature search and searching databases of news wire services.

Although constrained spreading activation is a simple algorithm, and seems widely applicable, the investment required to build GRANT-like systems is substantial. Five steps are involved. First, one must analyze the domain to design a language for representing the domain's concepts and their interrelationships. Concepts in GRANT's network are linked by 24 different relationships and their inverses. We had to interview an expert funding advisor at length to acquire this vocabulary of links. Second, a network must be constructed to represent and index the targets of search, be they agencies, bibliographic references, or people. Roughly 4 person-months of effort were required to build GRANT's 4500-node, 700-agency network. Third, path endorsements must be formulated. Fourth, the system must be tested and the path endorsements refined. Finally, for most interesting domains, one will be constantly updating information about the targets of search, adding new ones, modifying the descriptions of old ones, and so on.

References

[Abelson and Sussman, 1985]
Abelson, H. & Sussman, G. J. *Structure and Interpretation of Computer Programs.* Cambridge, Massachusetts: The MIT Press, 1985.

[Abrett and Burnstein, 1987] Abrett, G. & Burstein, M. The KREME knowledge editing environment. *International Journal of Man-machine Studies*, 1987.

[Bennett, 1986] Bennett, J. S. COAST: A task-specific tool for reasoning about configurations. Technical Report, Teknowledge Inc., Palo Alto, CA, 1986.

[Brown and Chandrasekaran, 1984] Brown, D. C., & Chandrasekaran, B. Expert systems for a class of mechanical design activity. *Proceedings of the IFIP WG5.2 Working Conference on Knowledge Engineering in Computer Aided Design,* Budapest, Hungary, 1984.

[Bylander and Mittal, 1986] Bylander, T. & Mittal, S. CSRL: A language for classificatory problem solving and uncertainty handling. *AI Magazine*, **7**(3), August, 1986, 66–77.

[Chandrasakeran, 1986]
Chandrasakeran, B. Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design. *IEEE Expert*, Fall:23–30, (1986).

[Chandrasakeran, Mittal and Smith, 1982]
Chandrasakeran, B., Mittal, S., & Smith, J. W. (1982). Reasoning with uncertain knowledge: the MDX approach. *Proceedings of the Congress of American Medical Informatics Association*, San Francisco, 335–339.

[Clancey, 1985] Clancey, W. J. Heuristic Classification. *Artificial Intelligence,* **27**, 1985, 289–350.

[Clancey, 1986] Clancey, W. From GUIDON to NEOMYCIN and HERACLES in twenty short lessons. *AI Magazine*, **7**(3), 1986, 40–60.

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

[Cohen *et al.*, 1987a] Cohen, P., Day, D., Delisio, J., Greenberg, M., Kjeldsen, R., Suthers, D., & Berman, P. Management of uncertainty in medicine. *Proceedings of the IEEE Conference on Computers and Communications*, Phoenix, Arizona, February, 1987.

[Cohen *et al.*, 1987b] Cohen, P., Day, D., Delisio, J., Greenberg, M., Kjeldsen, R., Suthers, D., & Berman, P. Management of uncertainty in medicine. *International Journal of Approximate Reasoning.* 1(1): Forthcoming.

[Cohen and Feignebaum, 1982] Cohen, P. R., and Feigenbaum, E. A. *The Handbook of Artificial Intelligence, Vol. 3.* Addison-Wesley. Reading, Massachusetts, 1982.

[Cohen, Shafer, and Shenoy, 1987] Cohen, P. R., Shafer, G., and Shenoy, P. Modifiable combining functions. *EKSL Report 87-05,* Department of Computer and Information Science, University of Massachusetts,

[Davis and Buchanan, 1984] Davis, R. & Buchanan, B. G. Me a-level knowledge. In B. G. Buchanan & E. H. Shortliffe (Eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Reading, MA: Addison-Wesley, 1984.

[Erman *et al*, 1980] Erman, L. D., Hayes-Roth, F., Lesser, V., & Reddy, D. R. The Hearsay-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Survey*, **12**, 213–253.

[Eshelman and McDermott, 1986] Eshelman, L. & McDermott, J. MOLE: A knowledge acquisition tool that uses its head. *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, August, 1986, 950–955.

[Fikes, Hart, and Nilsson, 1972] Fikes, R., Hart, P., and Nilsson, N. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4):251–288, 1972.

[Gruber and Cohen, 1987b] Gruber, T. R. & Cohen, P. R. Design for acquisition: principles of knowledge system design to facilitate knowledge acquisition. To appear in the *International Journal of Man Machine Studies*, 1987

[Gruber and Cohen, 1987a] Gruber, T. R. & Cohen, P. R. Principles of Design for Knowledge Acquisition, *Proceedings of the Third IEEE Artificial Intelligence Applications Conference*, Orlando, Florida, February 23–27, 1987.

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

[Hayes-Roth *et al.*,, 1986] Hayes-Roth, B., Garvey, A., Johnson, M.V., and Hewett, M. A layered environment for reasoning about action. *KSL Report No. 86-38.* Department of Computer Science. Stanford University, 1986.

[Howard, 1966] Howard, R.A. Decision Analysis: Applied Decision Theory. In D.B. Hertz and J. Melese, editors *Proceedings of the fourth International Conference on Operational Research*, pages 55–71, Wiley, New York. 1966.

[Howe *et al*, 1986] Howe, A. E., Dixon, J. R., Cohen, P. R., Simmons, M. Knowledge. DOMINIC: A domain-independent program for mechanical engineering design. *International Journal for Artificial Intelligence in Engineering*, **1**(1), July, 1986, 23–29.

[Kahn *et al*, 1987]
Kahn, G. S., Breaux, E. H., Joseph, R. L., & DeKlerk, P. An intelligent mixed-initiative workbench for knowledge acquisition. To appear in *International Journal of Man-machine Studies*, 1987.

[Kahn, *et al*, 1984] Kahn, G., Nowlan, S. & McDermott, J. A foundation for knowledge acquisition. *Proceedings of the IEEE Workshop on Principles of Knowledge-base Systems*, Denver, Colorado, December, 1984, 89–98.

[Kjeldsen and Cohen, 1987] Kjeldsen R.; Cohen, P.R. "The Evolution and Performance of the GRANT System." to appear *IEEE Expert*. Spring 1987.

[Marcus, 1987] Marcus, S. Taking backtracking with acquisition grain of SALT. To appear in *International Journal of Man-machine Studies*, 1987.

[McDermott, 1983] McDermott, J. Extracting knowledge from expert systems. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, August, 1983, 100–107.

[Neches *et al.*, 1985] Neches, R., Swartout, W. R., & Moore, J. Enhanced maintenance and explanation of expert systems through explicit models of their development. *Transactions on Software Engineering*, **SE-**(11), 1985, 1337–1351.

[Newell, 1982] Newell, A. The knowledge level. *Artificial Intelligence*, **18**, 1982, 87–127.

[Nii, 1986] Nii, H. P. Blackboard systems, Part Two. *AI Magazine*, **7**(3), 1986, 82–65.

[Patil, *et al.*, 1981]
Patil, R.S., Szolovits, P., and Schwartz, W. B. 1981. Causal understanding of patient illness in medical diagnosis. *Proceedings of 7th International Conference on Artificial Intelligenee*. 893-899.

[Pauker, *et al.*, 1976] Pauker, S., Gorry, G., Kassirer, J., and Schwartz, W. 1976. Toward the simulation of clinical cognition: Taking a present illness by computer. *American Journal of Medicine*. 60:981-995

[Pearl, 1986] P[illegible], J. 1986. Fusion, propagation, and structuring in Bayesian networks. *Artificial Intelligence*. **29**: 241-288.

[Pople, 1977] Pople, H. 1977. The formation of composite hypotheses in diagnostic problem solving — an exercise in synthetic reasoning, *IJCAIS*. 1030-1037

[Raiffa, 1970] Raiffa, H. *Decision Analysis: Introductory Lectures on Choices Under Uncertainty*. Addison-Wesley, Reading, Massachusetts, 1970.

[Sacerdoti, 1979] Sacerdoti, E. Problem solving tactics. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 1077–1085, 1979.

[Samuel, 1959] Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*. 3:210:229.

[Shafer, 1981] Shafer, G. 1981. Jeffrey's rule of conditioning. *Philosophy of Science*. **48**(3): 337-362.

[Shafer and Tversky, 1985] Shafer, G. and Tversky, A. 1985 Languages and Designs for Probability Judgment, *Cognitive Science*. **9**:309-339.

[Shenoy and Shafer, 1986] Shenoy, P. and Shafer, G. 1986 Propagating belief functions with local computations. *IEEE Expert*. **1**(3):43-52.

[Swartout, 1983] Swartout, W. XPLAIN: A system for creating and explaining expert consulting systems. *Artificial Intelligence*, **21**(3), 1983, 285-325.

[Trigoboff, 1978] Trigoboff, M. 1978. IRIS: A framework for the construction of clinical consultation systems. *Doctoral dissertation*, Computer Science Dept. Rutgers University.

On Making Expert Systems More Like Experts[1]

William R. Swartout
Stephen W. Smoliar

USC Information Sciences Institute
4676 Admiralty Way Suite 1001
Marina del Rey, CA 90292-6695

ABSTRACT: Expert systems still lack the skill of an expert when it comes to providing explanations of the results of expert reasoning. This is because while such systems may implement knowledge which is sufficient to mimic the *performance* of an expert, they do not necessarily model that expert's *understanding* of a problem domain. Such a model must include knowledge of that domain's terminology, knowledge of domain facts, and knowledge of problem solving methods. The Explainable Expert Systems project has been exploring a new paradigm for expert system development that is intended to capture such missing knowledge and make it available for explanation. This paper will discuss the principles behind this paradigm and consider two systems which have been subsequently developed.

1. Introduction

Being an expert, even in a single domain, requires more than one kind of expertise. In testing or validating an expert system, we focus on *performance* expertise, the knowledge required to get the right answer, or more precisely, to get the same answer as a human expert. But we require more of human experts than just getting the right answer. They must be able to explain and justify their answers, acquire new knowledge from a variety of sources, and learn from experience. Focussing on explanation, in this paper we will describe how examining the problem of producing explanations of expert system behavior has led us to a better model of the kinds of expertise that an expert system should possess. This includes knowledge of terminology, which is knowledge of domain concepts and how they are defined, knowledge of domain descriptive facts, which is knowledge that describes the structure of the domain and relationships among entities within it, and knowledge of problem solving methods.

For most expert system projects, the primary concern is to represent the knowledge needed for the expert system to produce solutions similar to ones produced by human experts. At one time, it was thought that if that knowledge was represented in a sufficiently declarative form it would be relatively easy to produce explanations of the system's behavior by paraphrasing the rules or traces of their execution into natural language. The fact that the expert system could mimic the expert's behavior was evidence that the expert's knowledge had been captured, so explanation merely required parroting back the knowledge base to the user in a palatable form.

This paraphrase-the-code approach could give reasonable explanations of *how* the system's problem solving methods worked or were applied to a particular problem. However, the problem was that the range of questions that people may reasonably ask of an expert goes beyond questions about how a solution was obtained. Answering these questions requires additional knowledge over and above the knowledge needed to produce the solution.

We will illustrate the limitations of the parapharse-the-code approach with an example from an early version of the Digitalis Therapy Advisor [Swartout 77]. Digitalis is a drug given to many cardiac patients. It is a difficult drug to give properly and considerable expertise is required for proper administration. One complicating feature is that certain physiological abnormalities in a patient may make him unusually sensitive to the drug. These sensitivities must be taken into account to reduce the risk of accidental overdose.

In Figure 1-1 the physician who is using the system wants to know why the system needs to know the patient's serum calcium level. The system responds by describing those activities it is trying to perform. While this explanation does suggest that serum calcium is some sort of a sensitivity, in all probability, it would not be considered acceptable if it were given by an expert physician. It would probably provoke certain follow-on questions which are, in fact, beyond the scope of this explanation facility.

To better understand the types of questions that should be offered, and the expertise that must be modelled to

[1]This paper has been submitted to the Journal of Expert Systems

ENTER NUMERICAL VALUE ==> why?
The user types "why?" indicating that he questions why he is being asked to supply the value of serum calcium.

WHAT IS THE LEVEL OF SERUM CALCIUM?

MY TOP GOAL IS TO BEGIN THERAPY. ONE STEP IN DOING THAT IS TO CHECK SENSITIVITIES. I AM NOW TRYING TO CHECK SENSITIVITY DUE TO CALCIUM.

Figure 1-1: A Limited Justification for a Question

answer them, we took protocols of potential end-users interacting with expert systems. The users were encouraged to ask questions during the interaction. No formal analysis of the protocols was undertaken, but based on them and our own experience as expert system builders, we identified approximately a dozen different classes of useful explanations (see [Swartout 86]). We will focus on three types of questions here:

1. Justifications
   - *Why is serum calcium an important factor in digitalis administration?*
2. Questions about terminology of the domain and its definition.
   - *What is a "sensitivity?"*
3. Questions about the intent behind a goal.
   - *What does it mean to perform a diagnosis?*
   - *What does it mean to check sensitivities?*

There are several reasons why it is important for an expert system to be able to answer questions such as these. First, a user is more likely to accept an expert system's recommendations if he can assure himself that the system's reasoning is based on a sound understanding of the underlying principles of the domain. Second, the answers to these questions can help a user understand how closely his understanding of the domain agrees with the system's. If there is a wide disparity, that can serve as a warning that the expert system may be being pushed beyond the bounds of its capabilities. Third, these explanations may help educate inexperienced users about the fundamentals of the expert system's domain.

The reason why these and similar questions, whose answers are so natural to an experts are so problematical for an expert system is because most explanations, like that in Figure 1-1, are based on knowledge that is only sufficient to mimic the *performance* of an expert, or traces of the results of the application of such knowledge. Thus, explanations can be provided of *how* a method works, or was applied in a particular setting. However, no account can be given for *why* such activities have occurred or *why* the system is trying to achieve them. Also, the terms that the system employs and the intent behind its goals cannot be defined because no explicit definition is provided for them.

What has happened to the information required to deal with such questions? The knowledge that is needed to answer them is known by a system builder at the time he creates an expert system and is used by him in the process of deriving the expert system's rule or methods. But because that knowledge is not needed for the expert system to *perform* properly, it does not appear in the rules or methods of the expert system itself, and hence is unavailable for explanation.

In the Explainable Expert Systems (EES) project, we have been exploring a new paradigm for expert system development that is intended to capture such missing knowledge and make it available for explanation. In our approach, system builders and domain experts collaborate to construct a high-level representation of knowledge in the domain that includes the normally missing knowledge that forms the basis for an expert system's rules or methods. Expert behavior is then derived automatically from this knowledge base. A trace of the derivation process is left behind. This trace connects the behavior of the system with the additional knowledge required to satisfy the needs of explanation.

In the remainder of this paper, we present two systems that follow this approach. In Sections 2 and 3 we discuss EES version I, which explicitly separates knowledge of how the domain works, knowledge of problem solving, and knowledge of terminology. From this high-level knowledge base, an automatic programmer derives performance-level rules or methods of the sort found in conventional expert systems (see Figure 1-2). The derivation process is recorded in a machine-readable development history, and that recorded trace is used to provide the normally missing knowledge needed for explanations that reflect not only the system's performance level knowledge but also the support knowledge underlying it.

In Section 4 we discuss EES version II. This system embodies a more explicit representation of the relation between problem solving knowledge and domain descriptive knowledge. Domain specific problem solving knowledge is derived directly from domain descriptive knowledge, so that problem solving methods may be explained in terms of the domain knowledge that underlies them. We have also added "weak methods," which are used to represent domain independent problem solving knowledge.

Specific issues concerned with the generation of explanations are discussed in Section 5. This task imposes additional contraints on the structure of our knowledge base, and we consider how these constraints may be satisfied.

2. A Declarative Representation for Expertise: The Kncwledge Base

In constructing an expert system using this approach, the first step is to understand what kinds of knowledge or expertise need to be represented and how that knowledge should be partitioned. Relying again on our study of question types, we found three important kinds of expertise to represent:

- *Terminological Knowledge* is knowledge of the concepts and relationships of a domain that experts use to communicate with one another. In expert systems, terminology forms a language that knowledge sources use to communicate with one another and it provides the building blocks from which representations for other kinds of knowledge are constructed.

- *Domain Descriptive Knowledge* describes how the domain works. It can be thought of as the "textbook rudiments" which are required before one can turn to solving problems. In a medical domain, this would be primarily physiological knowledge, describing causal relations among physiological states and symptoms associated with diseases, and the effects of various therapies. In another domain, such as diagnosing an electronic circuit, this would include knowledge of the circuit schematic and of the behavioral characteristics of the various components that made up the circuit.

- *Problem Solving Knowledge* is "how to" knowledge. It supplies knowledge about how tasks (called *goals* in our system) can be accomplished. This is where knowledge about how to perform a diagnosis or how to administer a drug belongs. In our representation, problem solving knowledge is organized into plans.

We shall now discuss each of these kinds of knowledge in terms of some specific examples.

2.1. Terminology

It may seem odd to think of terminology as a kind of expertise, but before one can begin to understand a domain, one must understand the terms that are used to describe it. During the first stages of building an expert system, when system builders are debriefing an expert about a domain, much of the time they spend together is concerned with understanding the terminology of the domain. Since terminology provides the building blocks out of which an expert system is constructed, it plays a pivotal role in the process of building an expert system.

Despite its importance, few expert systems have any representation for the definition of terminology. The terminology is known by the system builder, but it is not explicitly defined within the expert system itself. Instead, the terms used by the system implicitly acquire a

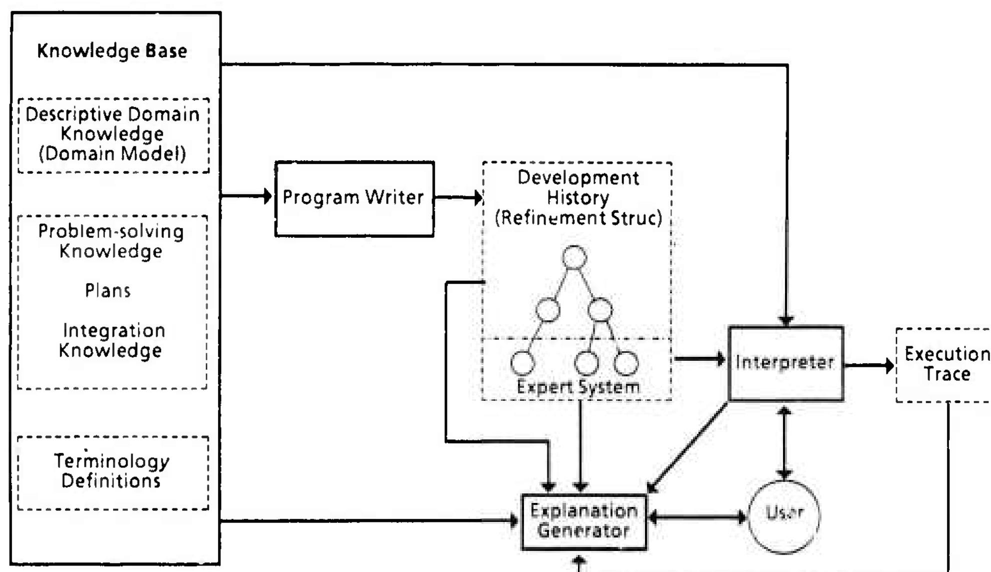


Figure 1-2: Architecture for EES version I

definition based on how other knowledge sources in the system react to them and the operational mechanisms for recognizing instances of those terms. This can lead to problems both in explanation and maintenance of an expert system.

To illustrate the problem of implicit terminology briefly, suppose we define a simple rule for recognizing fever:

If patient's temperature > 100
then conclude fever.

One might envision an explanation facility which, if faced with the need to define the concept of "fever," would search for knowledge of how fever was recognized. It could then use this rule to reply that fever was a condition in which the patient had a temperature greater than 100 degrees.

In fact, that would confuse an operational means for recognizing fever with a definition for it. To see that, suppose now that an expert system with this rule and explanation facility is actually deployed "in the field." Under such circumstances it may yield many false positive results because some people drink hot coffee before their temperature is taken. Such a "bug" may be easily fixed by modifying the rule:

If patient's temperature > 100
and patient has not recently drunk coffee
then conclude fever.

What will this do to the explanation facility? Since it depends on the modified rule, it will now include the consumption of coffee as part of its description of fever. Such a rule can also make system maintenance difficult. The two predicates in the rule serve very different roles. The first is concerned with establishing criteria for recognizing fever, while the second insures the validity of the temperature measurement. These roles are not differentiated in the rule itself and in more complex systems this can exacerbate maintenance problems. This example should demonstrate why, for the sake of explanation and maintenance, terminology should be regarded as a body of knowledge distinct from problem solving knowledge.

To provide an explicit representation for terminology, we have been using a knowledge representation system based on the ideas pioneered in KL/ONE [Brachman 78, Moser 83]. A design goal behind KL/ONE based formalisms is to provide an explicit representation for the definition of terminology. Our representation, like KL/ONE, is a semantic network-based formalism. Concepts (which correspond to terms) have attributes, corresponding to "slots" in frame-based representations. Restrictions may be placed on the possible fillers of the attribute slots for a particular concept, and these restrictions, together with the attributes of a concept contribute to the definition of the concept. Concepts are attributes arranged in a generalization hierarchy based on subsumption relations among them. As new terms are introduced, an automatic classification facility [Schmolze 83] determines their position in the classification hierarchy, based on their definitions and the definitions of existing terms (see [Neches, et al. 85] for an example of classification).

In the domain of digitalis therapy, some of the terms include physiological parameters that are important in this domain such as `increased serum calcium` and `decreased serum potassium`, both of which are specializations of `observable deviation`. Composite terms may be built from other terms, such as the goal `compensate digitalis dose for digitalis senstivities` which would be composed from the individual terms `compensate`, `digitalis`, `dose` and `sensitivity`.

As we will see below, terminology plays an important role in integrating domain descriptive knowledge and problem solving knowledge during the process of synthesizing an expert system. The term `drug sensitivity` will play that role in the example we present. We define it as:

drug sensitivity: `an observable deviation that causes something dangerous that is also caused by the drug`

2.2. Domain Descriptive Knowledge

As was mentioned above, domain descriptive knowledge describes how the domain works. It is typically the sort of knowledge that one finds in textbooks. Like definitions, such knowledge may be represented as declarative assertions. The domain descriptive knowledge for digitalis therapy included facts such as:

```
Increased digitalis causes increased
automaticity.

Decreased serum potassium causes increased
automaticity.

Increased serum calcium causes increased
automaticity.

Increased automaticity may cause
ventricular fibrillation.

Ventricular fibrillation is a dangerous
condition.

Decreased serum potassium is an observable
deviation.

Increased serum calcium is an observable
deviation.
```

In the heart, *automaticity* refers to the degree to which muscle cells are likely to fire spontaneously, resulting in abnormal heart rhythms. *Ventricular fibrillation* is one such abnormal rhythm that is very dangerous because it indicates a condition in which the heart has ceased to pump blood.

While causal knowledge was a central part of domain descriptive knowledge for this domain, it is important to emphasize that domain descriptive knowledge is not always causal. In well understood domains, such as electronic circuit analysis, the domain descriptive knowledge might be a mechanistic description of the circuit, while in poorly understood domains, the domain descriptive knowledge might merely consist of probabilistic associations between various states and state changes in the domain.

What is not part of domain descriptive knowledge is knowledge of how to achieve various results: how to diagnose a patient or administer a drug. That knowledge is part of problem solving knowledge.

2.3. Problem Solving Knowledge

Problem solving knowledge is expressed as plans that express how tasks can be accomplished. Plans have *capability descriptions* which describe what goals they can achieve. Each plan also has a *method* which is a sequence of substeps (which may themselves include subgoals) for accomplishing the goal. Capability descriptions are patterns and may include variables that are bound when the capability description is matched against a goal to be achieved. Plan capabilities and goals are represented using our representation of terminology described above. The generalization hierarchy of terms induces a generalization hierarchy of plans which we use for finding candidate plans for achieving goals (see [Neches, et al. 85] for details).

Another aspect of problem solving knowledge is *integration* knowledge. This is knowledge of how to combine similar results from multiple knowledge sources. We will describe integration knowledge in greater detail in the context of an example in Section 3.1.

In EES, we have tried to improve on the very specific, low-level representation of problem solving knowledge that exists in conventional expert system frameworks. As an example, one of the situations that MYCIN had to deal with was the possibility of being able to figure out the genus of the micro-organism that was infecting the patient but not being able to figure out its species. In that case, MYCIN just assumed that the species of the micro-organism was the most likely one for the particular genus, a reasonable default strategy. What was unreasonable was that that strategy was not expressed as a single rule, but instead as a whole collection of rules, each specific to one of the genera that MYCIN knew about. From the standpoint of explanation, this is bad because the system has no representation of the general strategy it is following; hence it cannot be explained. It is also bad from the standpoint of modifying the expert system. If we wanted to modify MYCIN's default strategy, there would be no single place to modify. Instead, we would have to carefully locate by hand each of the rules that instantiated that strategy and modify them, with all the attendant possibilities for making a mistake.

Plans in EES are usually expressed at a higher level of abstraction than the rules or methods of conventional expert systems. Modification or extension of the system's problem solving knowledge is performed at the level of plans, and EES is responsible for propagating the results of such modifications into the implementation of the expert system. This eases extension and modification. Such abstraction also opens up the possibility of re-using existing problem solving knowledge in new domains and of explaining problem solving methods at a general level (see [Swartout 83]).

As an example, consider a plan for compensating for drug sensitivities. It captures the common sense notion that if a patient has a sensitivity to a particular drug, then the dose of that drug should be reduced. This plan may be paraphrased as follows:

```
Capability-description:
   Compensate drug dose for a drug
     sensitivity

Method:  If the drug sensitivity exists
          in the patient,
         then reduce the drug dose because
          of the drug sensitivity
```

We shall now show how this plan, together with domain descriptive knowledge and terminology, can be used to implement specific rules for checking for specific sensitivities.

3. Combining Different Kinds of Expertise: The Program Writer

The program writer combines these different kinds of expertise together to produce a working implementation of an expert system. The program writer creates the expert system using goal refinement and reformulation. Starting with a high level goal (such as "administer digitalis") the writer searches through its hierarchy of plans for those plans whose capability descriptions subsume (that is, match) the goal. It selects one of the matching plans and instantiates its method. This results in the posting of subgoals in the method as further goals to be implemented, and the program writer searches for plans to implement those goals in turn. The writer

continues in this fashion until all goals have been implemented in terms of primitive constructs[2].

In the event that no plan is found for implementing a goal, the program writer attempts to reformulate, or transform, the goal into a goal or set of goals that *can* be implemented. We added this capability to provide several benefits. Maintenance and initial system construction would be easier because the program writer would be able to bridge larger gaps between plans and goals, and knowledge would be re-usable in a larger range of situations. We identified several different kinds of reformulations (see [Neches, et al. 85] for a detailed discussion). As an example, we shall consider ***reformulation into cases***, that is, reformulating a goal of an action to be performed over a set of objects into a set of goals where the action is performed on individual elements of the original set of objects. Such reformulation takes place frequently (but implicitly) in expert systems.

For example, in many diagnostic systems a problem that arises is to determine how likely it is that a patient has some disease based on its signs and symptoms. In conventional expert systems that goal is usually not explicitly represented in the system, because the system designer mentally reformulates it while constructing the system. What does appear is the result of the reformulation: a set of goals that inquire about each of the symptoms individually and a combining function that deals with the problem of how to collect the individual assessments of signs and symptoms into an appropriate overall assessment for the disease. EES allows us to represent the original goal, the reformulation, and the result of the reformulation explicitly. As we will show in the example below, goal reformulation plays an important role in integrating domain descriptive knowledge, problem solving knowledge, and terminology.

3.1. An Example

To make the operation of the program writer clearer, let us consider an example from the digitalis domain concerned with the problem of adjusting the patient's dose to account for digitalis sensitivities.[3]

At some point in the program writing process a goal would be posted to:

```
Compensate digitalis dose for
   digitalis sensitivities
```

This is a goal to compensate the dose for all the digitalis sensitivities that are known to the system (by being represented in the domain descriptive knowledge) or that can be deduced by the system. The program writer would search through the hierarchy of plans to find all the plans whose capability description subsumed the goal. In this case, none would be found. The plan in Section 2.3 might seem to be directly applicable to this goal because the pattern in its capability description looks very similar to the goal. In fact, we must do some goal reformulation before that plan can be applied. The problem is that the goal requires compensating for *all* the sensitivities, but the plan can only compensate for an *individual* sensitivity. Thus, we must reformulate the goal into a set of goals over individual sensitivities before program writing may proceed. The system does that by making use of its terminological and domain descriptive knowledge. Consulting the terminological knowledge for the definition of sensitivity (given in Section 2.1), it finds that a drug sensitivity is an observable deviation that causes something dangerous to happen that is also caused by the drug. Specializing that term to *digitalis sensitivity*, and using the domain facts (given in Section 2.2), the classifier finds two individual observable deviations that are digitalis sensitivities: `increased serum calcium` and `decreased serum potassium`. The writer then reformulates the orginal goal over digitalis sensitivities into two goals over individual digitalis sensitivities:

```
Compensate digitalis dose for
   increased serum calcium
Compensate digitalis dose for
   decreased serum potassium
```

The method of Section 2.3 can then be applied to each of these two goals. When that is done, the methods are instantiated to produce two code fragments:

```
If increased serum calcium exists in
   the patient
then reduce the digitalis dose
   because of increased serum
   calcium

If decreased serum potassium exists
   in the patient
then reduce the digitalis dose
   because of decreased serum
   potassium
```

---

[2]These include constructs for setting a variable, conditional constructs, and the like, corresponding to LISP constructs such as SETQ, COND and so forth.

[3]The example we present here was actually implemented using the XPLAIN framework [Swartout 83], which also produced the sample explanations that appear in Figure 3-1. XPLAIN was a precursor to the EES framework and did not provide an explicit representation for terminology. We feel that version I of EES provides a cleaner conceptualization of the program writing process, and we have implemented two demonstration-sized systems in EES version I. Unfortunately, certain limitations on the expressivity of the knowledge representation system we used in EES version I would have made it difficult to express some of the terminology used in this example, particularly *sensitivity*. We are using an extended representation in EES version II, which we describe in Section 4. We are presenting this example from the perspective of EES, rather than XPLAIN, because we feel it provides a more understandable account of the program writing process.

The problems of determining whether increased serum calcium or decreased serum potassium exist and of reducing the dose are then posted as new goals for the system to implement.

An additional problem that confronts the program writer is to reason about how to integrate the two code fragments. Since both fragments cause the dose to be reduced the issue is to determine what should be done if both sensitivities co-occur. This is an example of what we refer to as an *integration* problem, that is, how to integrate similar conclusions reached by multiple knowledge sources. In most expert systems, this kind of problem is handled by some implicit mechanism built into the system's interpreter (like the certainty factor mechanism in MYCIN). Because the mechanism is built into the interpreter, it is convenient to use, but also subject to abuse, since the assumptions that underlie it are never explicitly checked. Also, usually only one mechanism is provided, so system builders will attempt to apply that mechanism to as many situations as possible, even if its appropriateness is questionable. We argue that integration problems should be reasoned about explicitly by the program writer while the expert system is being created. Taking that approach allows the assumptions that underlie an integration technique to be checked. Also, several techniques may be represented, allowing the program writer to select the most appropriate one for the problem at hand. In this particular case, the system uses a piece of integration knowledge that tells it that the two program fragments can be chained together (that is, connect the outputs of one to the inputs of the next) if the causal relations that the fragments are based on are independent and additive (see [Swartout 83] for further details).

This entire process was recorded so that it could later be used in giving much richer explanations that reflected the causal underpinnings that the expert system was based on, as shown in Figure 3-1. The critical difference between that explanation and the one in Figure 1-1 are the second and third sentences of the first explanation which provide a causal reason for checking serum calcium. This explanation was produced by paraphasing the causal relations that matched the domain rationale of the plan used to generate this code for checking serum calcium

4. Capturing Intent and the Roots of Problem Solving Knowledge

While the EES version I framework allows us to capture the knowledge needed to explain the rationale that underlies an expert system, two issues remain. First, EES version I still does not provide the capability to represent the *intent* behind a goal (our third question type). For example, it is not possible to answer the question: "What does it mean to administer digitalis?" Problem solving knowledge of *how* to give digitalis can be retrieved, but it is not represented anywhere that the problem of digitalis administration is a problem of finding a dosage level of digitalis that produces

Please enter the value of serum calcium: why?

The system is anticipating digitalis toxicity. Increased serum calcium causes increased automaticity, which may cause a change to ventricular fibrillation. Increased digitalis also causes increased automaticity. Thus, if the system observes increased serum calcium, it reduces the dose of digitalis due to increased serum calcium.

Please enter the value of serum calcium: 9

Please enter the value of serum potassium: why?

(The system produces a shortened explanation, reflecting the fact that it has already explained several of the causal relationships in the previous explanation. Also, since the system remembers that it has already told the user about serum calcium, and because it knows that the same plan was used to generate the code for both serum potassium and serum calcium, it suggests the analogy between the two here.)

The system is anticipating digitalis toxicity. Decreased serum potassium also causes increased automaticity. Thus, (as with increased serum calcium) if the system observes decreased serum potassium, it reduces the dose of digitalis due to decreased serum potassium.

Please enter the value of serum potassium: 3.7

Figure 3-1: A Causal Explanation of Why Serum Calcium and Potassium are Checked

satisfactory therapeutic results subject to the constraint of avoiding (or minimizing) toxic effects.

As another example, consider an expert system we built using EES for diagnosing faults in space telemetry systems. This system had several methods for diagnosis, which we have hand-paraphrased in Figure 4-1. Such a display is really the only response that system could have given to the question: "What does it mean to diagnose a decomposible system?" With some effort a user might be able to examine Figure 4-1 and figure out *how* the system performs a diagnosis, but it would be considerably harder for him to figure out *what* a diagnosis amounts to. What

is needed is an explicit representation of the intent behind a goal that would allow us to answer: "To diagnose a decomposable system means to find a primitive subcomponent of the system that is faulty."

```
To diagnose a decomposable system,
  If there is a fault in the system,
   then locate the cause of the fault
     within the system

To diagnose a primitive system,
  If the system is faulty,
   then conclude it is the diagnosis

To locate the cause of a fault within a
  system which is loosely-coupled,
 Diagnose the subcomponents of the system

To locate the cause of a fault within a
  system which is tightly-coupled,
 Locate the cause of the fault along the
   signal-path beginning at the
    system-input and ending at the
    system-output.

To locate the cause of a fault beginning at
  system1 and ending at system2,
 If system1 is faulty
    then diagnose system1
   else locate the cause of the fault along
    the signal-path beginning at the system
    that system1 outputs to and ending at
    system2.
```

Figure 4-1: Methods as an Inadequate Explanation of the Goal of Diagnosis

Second, we want to understand (and be able to explain) the source of problem solving knowledge. While the implementation of EES described in the preceding sections allows a system builder to represent problem solving expertise at a more abstract level than is possible in most expert system frameworks, it is clear that even that expertise is compiled from some still more basic knowledge. The question is: what is that more basic representation, and how does that compilation take place? By understanding the "roots" of problem solving knowledge, we hope to be able to provide better explanations of how the problem solving knowledge works.

Ultimately, we would like to be able to explain the compromises and approximations that were involved in an implementation and the way in which conflicting prefences were resolved. We are still a long way from that goal, but we have developed a basic framework within which we can begin to address these issues.

The approach we have adopted to address these problems is to represent goal intent in terms of a small number of primitive actions and then to mechanically derive methods for achieving those goals by transforming definitions and axioms in the domain descriptive knowledge base. As shown in Figure 4-2 this modifies the EES architecture by adding a new transformational component that derives EES plans by transformation. As before, these plans will then used by the program writer to create an expert system.

4.1. Capturing Intent: Primitive Actions

In general, expert systems lack any specification of what their goals mean that would make it possible to

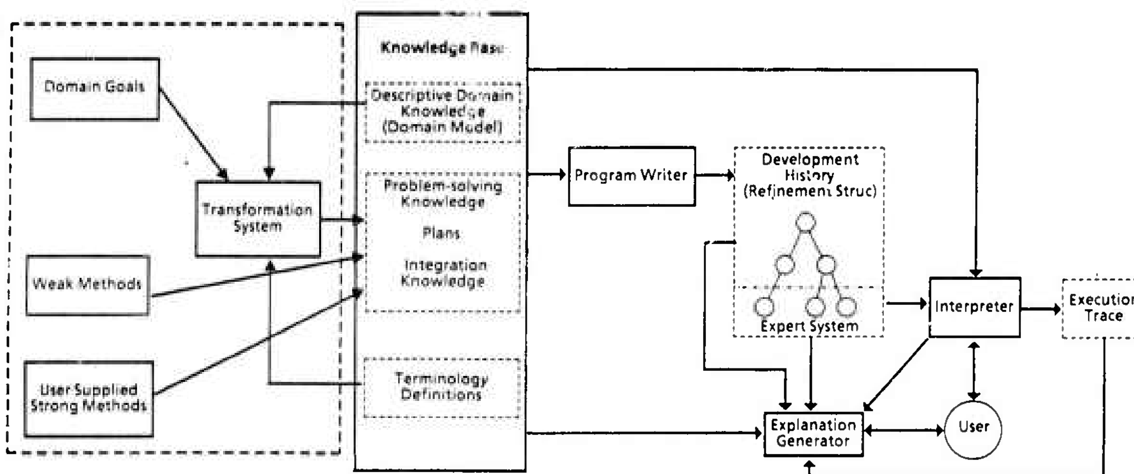


Figure 4-2: Architecture for EES version II

answer questions about intent. Thus, most expert system goals are defined like their terminology: implicitly. Goals acquire their meaning based solely on the methods that implement them. What we would like is a separate definition for the intent of goals. This would essentially serve as a specification for what achieving the goal entails. It would provide a basis for the meaning of goals that would allow the system to explain what they mean and provide an independent criterion that could be used in deciding whether or not the system understood goals in the same way as the user.

The problem of representing intent is really a problem of defining terminology, but specialized to defining verbs and verb clauses. To provide a basis for these definitions, we have attempted to uncover a core set of *primitive actions*. Higher-level goals are defined and explained in terms of these primitive actions. These primitive actions are objects that cannot be further explained, so in defining a set of primitive actions, it is important to select actions that almost every user can be assumed to understand readily.

This approach has its roots in Pappus' discussion of solving mathematical problems which was analyzed at great length by Polya [Polya 71]. Pappus based his work on the observation that all mathematical problems could be reduced to the primitive actions of *finding* and *proving* and that problem solving was a matter of planning subgoals (also expressed in terms of those primitive actions) through which those actions could be achieved.

We have been exploring this approach in the context of a system for diagnosing digital circuits. So far, we have identified and extensively analyzed two primitive actions in this domain:

1. ***determine-whether:*** establishes the truth of a given assertion

2. ***find:*** finds an object that matches a given description

Given these primitive actions, capturing the intent behind a domain level goal then involves linking that goal to its definition in terms of primitive actions. Thus, we would define the goal:

"diagnose decomposable digital system ***s***"

as the problem:

"*finding* a primitive system ***p*** such that ***p*** is a subcomponent of ***s*** and ***p*** is faulty"

We acknowledge that the primitive actions *find* and *determine-whether* may not accommodate other areas of problem solving as readily as mathematics or circuit diagnosis. Thus, we believe that our set of primitive actions will grow somewhat as we gain more experience with this approach. For example, a "canonical" representation of the goal of administering digitalis might be expressed as *finding* a digitalis dosage level which satisfies a set of assertional constraints; but what are we to make of goals involving the verb "compensate?" Such an action might be expressed in terms of *achieving* (or *avoiding*) a patient state; and, certainly, achieving and avoiding are actions that are often found in problem solving. However, we are still in the process of investigating how such actions may best be represented as primitive actions.

4.2. The Roots of Problem Solving Knowledge

While our study of primitive actions is still an area of active research, it has already yielded valuable consequences for our representation of problem solving knowledge. We shall discuss two of these consequences in greater detail:

1. The mechanical derivation of plans from the declarative representation of definitions and domain descriptive knowledge.

2. The introduction of domain-independent problem solving knowledge which may be integrated with domain-specific plans.

Mechanically derived plans

As has already been observed, domain descriptive knowledge can be likened to textbook knowledge. Often, however, the major problem with such knowledge is that it offers little help regarding how it should be used. For example, what is the use of knowing that decreased serum potassium causes increased automaticity?

In Section 3 that question was answered in terms of the behavior of an automatic programmer which generated expert system code. However, the knowledge was used to tie together specific units of problem solving knowledge which had no direct connection back to the definitions and domain descriptive knowledge. We wished to consider an alternative situation in which plans could be *derived* from declarative knowledge, rather than entered independently.

Our approach is to construct a set of transformations, specific to each of the primitive actions, that can be applied to facts in the domain descriptive knowledge to create problem solving knowledge in the form of plans. For a very simple example, if the knowledge base contains the assertion that "A exists if and only if B exists," then it is possible to derive a plan that determines whether B exists by checking for the presence of A. Since the implication is two-way, it is also possible to derive another plan for checking for the existence of A by checking for B[4].

[4]Of course, care must be taken in interpreting such plans to avoid circular reasoning chains.

Considerably more complex examples can be handled by our transformations. For example, in constructing the digital circuit diagnoser, the domain descriptive model included the circuit schematic, detailing the interconnections among devices in the circuit, and descriptions of the functional behavior of those devices. Thus, the domain descriptive knowledge included the facts such as:

```
multiplier M2 is connected to adder
   A1

the expected output of an adder is
   equal to the sum of its expected
inputs
```

These facts, and others describing the remaining topology of the circuit and the behavior of other devices, it was possible to mechanically derive a set of procedures for *finding* the expected signal value along any connector in the circuit, given a particular set of input values.

Encouraged by our results in the domain of diagnosis of digital circuits, we have begun a re-implementation of portions of the digitalis advisor using this framework. An interesting observation is emerging from our initial work in the digitalis domain, which is that different kinds of primitive actions seem to involve transformations over different kinds of domain descriptive knowledge. The transformations for deriving plans for performing *find* actions involve sets and instances, *determine-whether* involves implications and types, and *achieve* and *avoid* involve states, state transitions, and causality.

We feel that the tranformation mechanisms capture some very general kinds of problem solving knowledge which is transformed into more specific plans given the domain particulars as expressed in the domain descriptive knowledge. If the number of primitive actions remains relatively small, the fact that that general problem solving knowledge is only captured implicitly in the transformation system will not be too much of a problem because it will be feasible to build that knowledge into the explanation routines as well. On the other hand, if the number of primitive actions grows, such an approach will not be feasible, and it will be necessary to find more declarative means for defining the problems solving knowledge.

Weak methods

Because the objects affected by primitive actions are very simply defined, one may conceive of plans whose capability descriptions involve such actions in a domain-independent manner. For example, performing the determine-whether action on an assertion which is a conjunction does not involve any domain-specific knowledge. The method for such a plan would be based on subgoals which perform the determine-whether action on each of the conjuncts. Thus, one may develop a set of plans for the determine-whether action corresponding to the different syntactic possibilities of an assertion whose truth is being determined.

Such plans are generalizations of the domain-specific plans which have already been considered. Any goal which matches the capability description of a domain-specific plan may also match the capability description of one or more of these more general plans. We call the domain-independent plans *weak methods*; and they play two important roles:

1. They provide an operational semantics for primitive actions. Thus, the weak methods provide knowledge of how to determine-whether an assertion holds, regardless of that assertion's domain-specific content.

2. This semantic base allows the problem solver to apply "first principles" to any goal which cannot be accommodated by domain-specific knowledge. If no domain-specific plan has an appropriate capability description, one can always resort to the weak methods.

Our view of "weak methods" differs from the more common use of the term, such as may be found in [Laird 83]. This more familiar usage is based on a view of problem solving as application of operators within a problem space. From this approach, weak methods serve as decision procedures for the selection of the most appropriate operators. In our approach a problem is represented not by a problem space but by a primitive action which must be achieved, and the solution of a problem is the planning of subsidiary actions which will obtain this result. Weak methods, then, may be regarded as the most general plans for performing primitive actions, because they deal with the actions themselves, rather than with any problem-dependent objects affected by the actions. Nevertheless, our approach does share some common ground with the problem space view of weak methods. For example, the generate-and-test method of [Laird 83] is represented in our system as a weak method for dealing with the *find* action. Indeed, we conjecture that *every* weak method analyzed in [Laird 83] may be represented as the realization of some primitive action.

We recognize that due to limitations in our technology for automatically deriving plans, there may be desired expert behavior that we cannot achieve relying solely on mechanically derived plans and weak methods. For that reason, we have included the possibility of manually defining strong (domain-specific) methods and entering them directly into the plan hierarchy (see Figure 4-2). Like other plans, these are organized in the hierarchy based on their capability descriptions. The program writer retrieves and instantiates them just like other plans. The difference is that these plans are less

explainable. They cannot be related them back to underlying domain descriptive knowledge, as with the mechanically derived plans. We feel that one of the strengths of our approach is that it permits us to approach explanation in this incremental fashion. We can derive some plans mechanically, but our inability to derive other plans does not preclude constructing a system. Thus, it is possible to build a system even if it cannot be fully explained, but the explanations will improve with our increased understanding of the relation between domain descriptive and problem solving knowledge.

As of this writing, we have constructed sets of transformations for the *find* and *determine-whether* primitive actions and have used them to derive plans in the domain of digital circuit diagnosis. We have also defined approximately 25 weak methods for the *find* and *determine-whether* actions. We have constructed an interpreter that can execute these plans to produce problem solving behavior. We are currently in the process of integrating the transformational system with our program writer. We are also exploring the applicability of this approach in other domains, such as digitalis therapy. We feel that these explorations will lead to a better understanding of the kinds of primitive actions that are appropriate to model.

5. Further Requirements for Explanation

In the preceding sections, we have argued that if a system is to explain its reasoning, it is necessary to model additional kinds of expertise that are normally left out of a performance-oriented expert system. We have described the nature of those additional kinds of expertise and our approach to capturing that expertise in an expert system. In this section, we return to the issue of explanation, and describe some additional constraints that explanation imposes on the way that knowledge is structured and represented.

The Need for a Continuum of Abstraction

From the preceding sections it might appear that we feel that a more abstract or "deeper" representation for expertise is always best. In fact, it is important to have a variety of different levels of abstraction available for explanation and to select among them based on the experience and interests of the user. For example, we have argued that an explanation that is just based on performance-level expertise is probably not appropriate for many users because it leaves out the rationale that justifies it. But such an explanation may be very appropriate for an expert user who fully understands the rationale and is just interested in assuring himself that the system will take the correct actions in a particular set of circumstances.

Thus, it is not just a matter of reasoning with the compiled, performance-level knowledge for efficiency while falling back to the deeper knowledge for explanation. Instead, the explanation routines must be capable of selecting among knowledge expressed at different levels of compilation and producing explanations from that knowledge. That implies that the different levels of compilation and the correspondences among them must exist together. In comparing our approach of compiling an expert system from deeper level expertise with an approach in which the compilation step is skipped and the deep knowledge is directly interpreted, we often we often argue for our approach on the basis of increased efficiency but the explanation requirement for simultaneously existing multiple levels of compilation also argues for our approach.

Different Information for Different Users

Paris [Paris 87] has observed that explanations for novices and explanations for more experienced people differ fundamentally in the *kind* of information that is conveyed, not just in the level of detail, as had been previously thought [Wallis 82]. Paris studied descriptions of various devices in both junior and adult encyclopedias. She discovered that the entries in junior encyclopedias tended to emphasize the function of the device and the functional relations of its parts, while adult encyclopedias emphasized component/subcomponent relationships and the physical structure of the device. Presumably, functional information is left out of the adult entries because adults already know that information. While Paris has identified this phenomenon she has not yet provided a theory that will explain what kinds of knowledge will be included and what kinds will be left out.

We feel that the compilation process in EES may be the beginnings of such a theory, at least for expert systems. Explanations presented at different levels of compilation differ fundamentally in the kinds of knowledge that is presented. Explanations produced from uncompiled knowledge will include definitions of terminology, causal relations, and abstract descriptions of problem solving strategies. That is all information that will probably be familiar to an expert, and hence not necessary to explain to him, but that a novice is likely to want to know. Explanations produced from compiled knowledge will be most appropriate for experts because they will not contain the motivating knowledge that experts would already know.

6. Conclusion

We have argued that explanations produced solely from performance expertise will be inadequate, because such explanations leave out the rationale upon which the expert system's behavior is based. We described three kinds of expertise that must be modelled to provide adequate explanations: knowledge of terminology, domain descriptive knowledge, and abstract problem solving knowledge. In conventional expert systems, these

different kinds of knowledge are confounded together in a relatively low level representation, such as rules, if they are represented at all. Separating the different kinds of knowledge improves explanations because it allows the explanation routines to select just the right information to present to answer a user's questions, free of confounding factors. The separation also makes the system easier to maintain because it increases its modularity.

We presented a framework for expert system construction that employs an automatic programmer to integrate the different kinds of knowledge together to produce a working expert system. The program writer leaves behind a record of the design decisions that underlie the expert system. These "mental breadcrumbs" are used by explanation routines to explain the workings of an expert system in terms of basic domain knowledge.

We presented our current research, which is concerned with providing a representation for the intent of goals in terms of primitive actions, and with providing a better understanding of the relationship between domain descriptive knowledge and problem solving knowledge. Finally, we presented some additional requirements that explanation imposes, and showed how our approach supports them. Clearly, much remains to be done before expert systems will be able to explain themselves as lucidly as human experts. Nevertheless, our results so far encourage us in our future undertakings.

ACKNOWLEDGEMENTS

The research described in this paper was supported under DARPA Grant #MDA 903-81-C-0335 and National Institutes of Health Grant #1 P01 LM 03374-01 from the National Library of Medicine. The authors would like to thank Robert Balzer, Lewis Johnson, Jack Mostow, Robert Neches, Ramesh Patil, Peter Szolovits and David Wile for their comments, suggestions and discussions during the course of this research.

REFERENCES

[Brachman 78] Brachman, R., *A Structural Paradigm for Representing Knowledge*, Bolt, Beranek & Newman, Inc., Technical Report, 1978.

[Laird 83] Laird, J. and Newell, A., *A Universal Weak Method*, Carnegie-Mellon University Department of Computer Science, Pittsburgh, PA, Technical Report CMU-CS-83-141, June 1983.

[Moser 83] Moser, M.G., "An Overview of NIKL, the New Implementation of KL-ONE," in *Research in Natural Language Understanding*, Bolt, Beranek, & Newman, Inc., Cambridge, MA, 1983. BBN Technical Report 5421

[Neches, et al. 85] Neches, R., W. Swartout, J. Moore, "Enhanced Maintenance and Explanation of Expert Systems through Explicit Models of Their Development," *Transactions On Software Engineering*, November 1985. Revised version of article in Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, December. 1984

[Paris 87] Paris, C. L., "Combining discourse strategies to generate descriptions to users along a naive/expert spectrum," in *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, IJCAI, 1987. (to appear)

[Polya 71] Polya, G., *How To Solve It: A New Aspect of Mathematical Method,* Princeton University Press, Princeton, NJ, 1971. Second edition.

[Schmolze 83] Schmolze, J.G. & T.A. Lipkis, "Classification in the KL-ONE Knowledge Representation System," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, IJCAI, 1983.

[Swartout 77] Swartout, W.R., *A Digitalis Therapy Advisor with Explanations*, Massachusetts Institute Technology, Technical Report Laboratory for Computer Science TR-176, February 1977.

[Swartout 83] Swartout, W., "XPLAIN: A system for creating and explaining expert consulting systems," *Artificial Intelligence* 21, (3), September 1983, 285-325. Also available as ISI/RS-83-4

[Swartout 86] Swartout, W., "Knowledge Needed for Expert System Explanation," *Future Computing Systems*, 1986. (revised version of paper in NCC '85, accepted for publication)

[Wallis 82] Wallis, J.W. & E.H. Shortliffe, "Explanatory power for medical expert systems: Studies in the representation of casual relationships for clinical consultations," *Methods of Information in Medicine* 21, 1982, 127-136.

The Loom Knowledge Representation Language

Robert Mac Gregor
Raymond Bates

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

Abstract

The lengthening lifetimes of intelligent systems, and the desire to share or re-use knowledge bases, has created within the AI community the need for application-independent knowledge representation systems. The Loom system being developed at ISI represents the latest in a series of "classification-based" knowledge representation systems developed to meet this need.[1] In Loom, the traditional single-classifier architecture is replaced by one containing a collection of classifiers which exhibit increasingly powerful inference capabilities. This paper describes the knowledge representation language developed for the Loom system.

1. Introduction

Loom[2] represents a recent entry into the KL-ONE [Brachman and Schmolze 85] family of knowledge representation systems. Loom directly succeeds the NIKL system [Schmolze and Lipkis 83, Moser 83] developed jointly by ISI and BBN. During NIKL's lifetime, the NIKL user community produced a rather extensive list of extensions that they wished to see in future versions of NIKL [Kaczmarek 86]. Loom's designers determined that these needs could best be achieved by redesigning and reimplementing NIKL. The result is a more flexible architecture which preserves the strengths of the original NIKL while admitting some new and powerful forms of reasoning.

Loom's architecture strongly reflects the view that the variety of inferences provided by a comprehensive knowledge representation system can best be performed by a well-integrated collection of specialized reasoning components, rather than by a single, general-purpose reasoner. KL-ONE-style systems (e.g., KL-ONE, KL-TWO [Vilain 85], KRYPTON [Brachman, Fikes, and Levesque 83], and BACK [von Luck 87]) have traditionally divided their knowledge space into two partitions, called the "Terminological Box" and the "Assertional Box", and have utilized two distinct reasoners (terminological and assertional) to carry out their inferences. Loom's principle architectural contribution is to introduce two additional partitions (the "Universal Box" and the "Default Box"), each having its own associated reasoning component.

Complementing this increase in the number of domain-independent reasoners embedded in the system architecture is a growing library of domain-specific, "narrow-coverage" reasoners. Currently these include facilities for computing or reasoning about transitive relations, sets, intervals, and some elementary forms of numeric reasoning. These reasoners can be invoked independently, or called by the broad-coverage reasoners.

The trick in integrating this collection of reasoners is to develop a language for expressing knowledge which emphasizes the overall coherence and uniformity of the knowledge structures. Loom accomplishes this goal by building on the "concept-centered" view of knowledge employed in KL-ONE (and NIKL). Accordingly, all universal and default knowledge is attached to specific

---

[1] This research is supported by the Defense Advanced Research Projects Agency under Contract MDA903-81-C-0335. Views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion of DARPA, the U.S. Government, or any person or agency connected with them.

[2] Loom: "A frame ... for interlacing ... sets of threads or yarns to form a cloth." Webster's.

concepts. In a similar vein, sets, intervals, and relations (including transitive and composite relations) are all realized as specialized forms of concepts -- their definitions share a uniform syntax, and each of them has its own sublattice within the concept taxonomy.

This paper introduces the syntax and semantics of that portion of the Loom knowledge representation language which represents meta-level knowledge. We include discussions on some of the types of inference which can be performed by the Loom system. We begin by defining the four broad types of knowledge managed by the Loom system, and then discuss each of the "Boxes" devoted to representing meta-level knowledge. The appendices include the knowledge bases used to illustrate examples of Loom syntax. A longer version of this paper [Mac Gregor 87] contains a complete definition of the Loom system.

2. Boxes

In order to accurately define concepts and relations in Loom, it is necessary to have an understanding of how Loom treats various "kinds" of knowledge within the system. Loom partitions its knowledge space into four "Boxes", called the Terminological, Universal, Default, and Assertional Boxes. This section presents a brief characterization of each of these four kinds of knowledge. Later sections will present specifics on the expressive features available with each of the Loom boxes.

Definitions within the "Terminological Box" (TBox) serve to define the "terms" in our knowledge representation scheme ([Brachman, Fikes, and Levesque 83] contains a good discussion of what kind of knowledge is considered to be "terminological"). A TBox definition yields a set of necessary and sufficient conditions for recognizing an instance of some concept. Within Loom, the organization (classification) of concepts is based strictly on the terminological knowledge available to the system.

The "Universal" Box (UBox) widens the scope of things we can say about (generic) concepts to include certain forms of knowledge about the "real world". In the UBox we can attach necessary conditions to a concept definition. For example, we can state that "live-persons necessarily have heads", i.e.,

$$\forall x[\mathit{Live\text{--}Person}(x) \rightarrow \exists y\, \mathit{head}(x,y)].$$

In the UBox we can also state conditions which are sufficient, but not necessary to recognize an instance of a concept. For example, we can say that "all featherless bipeds are human", i.e.,

$$\forall x[\mathit{Featherless\text{--}Biped}(x) \rightarrow \mathit{Human}(x)].$$

A second, more powerful classifier is associated with the UBox. The UBox classifier makes its inferences (classifications) on the basis of combined TBox and UBox knowledge.

The "Default" Box is the proper location for representing "assumptions" or "default knowledge". For example, in it we can state such things as default values: "If nothing has been asserted about the color of some elephant x, make the assumption 'color(x Grey)'." We can also state some limited forms of closed-world assumptions:[3] "If some paper P has K authors, assume that it has *only* K authors."

The knowledge represented in the Default Box is used to make some very limited types of inferences during the process of realization. A full-blown use of default knowledge would seem to require the inclusion of a non-monotonic reasoning capability into Loom. This is beyond the scope of our current effort.

The Assertional Box (ABox) is the repository for assertions about individuals. For example, we might

---

[3] By default,the ABox assumes open-world semantics

place in the ABox the assertion that Clyde is a white elephant by making the assertions:

```
(assert (Elephant Clyde) (color Clyde white)).
```

The effect of these assertions is to create an instance in the ABox of the concept `Elephant` (unless `Clyde` already exists in the ABox) and to assign to the `color` role of the object `Clyde` the value `white`.

Loom has extended NIKL's terminological language CNIKL [Robins 86] to include expressions of universal and default knowledge. We believe that it is beneficial to associate each fragment of universal or default knowledge with a particular concept; thus, we have chosen to extend the syntax of the original `defconcept` (and `defrelation`) primitives, rather than to add new (top-level) constructs to the terminological language. The Engines and Cars knowledge base in Figure A-1 illustrates some Loom concept declarations. The original CNIKL definition of a concept serves as its definitional component. An "axioms" clause states universal knowledge about a concept, while a "defaults" clause states default knowledge.

Engineering Note:

Our introduction of a new type of reasoner (the UBox classifier) puts us in line with what we see as a long-range trend towards knowledge representation architectures which will employ increasing numbers of specialized reasoners. As the number of reasoners within a single system increases, it will become increasingly important that some organizing principle is available to integrate these various reasoners. Our decision to organize all universal and default knowledge within the context of particular concepts illustrates a belief that the "concept-oriented" (a.k.a. "frame-oriented") approach will prove to be a successful organizing principle for wider and wider classes of knowledge. Such an approach may be contrasted with that of the current generation of rule-based systems (including hybrid frame- and rule-based systems); in those systems, knowledge which we have classed as universal or default knowledge (other than "default values") tends to be dumped unceremoniously into a "rule base", i.e., such systems provide no formal scheme for structuring that knowledge.

3. Basic Terminology

Here we take time-out to formalize some of our terms.

By a *concept* we mean an "intentional description" of something. The most general instance of a concept is called "`Thing`". A *relation* is a concept which defines a set of k-tuples, with k being fixed for each individual relation. By convention, the the term "concept" is often used to refer to (the more specialized notion of) a *unary relation.* Thus, he `defconcept` form defines a unary relation.

A binary-relation for which the roles domain and range have been assigned will be called a *mapping.* By convention, the term "relation" may be used in place of the word "mapping", and the form `defrelation` is used to define a mapping. The most general instance of a mapping is called "`maps-to`".[4] The Loom implementation is intended to accommodate relations of order greater than two, but a complete syntax for defining higher-order relations has not yet been worked out. A relation which has been reified (equated with a unary concept of the same name) is termed a *relationship.*

The domain of a mapping is not considered to be a part of its (TBox) definition. The association of a mapping with a particular (domain) concept, other than the concept THING, induces a sub-relation we call a *role.*[5] A *role restriction* which associates a mapping M with a concept C defines a role R_{CM} such that R_{CM} is a subset of M, and has domain C. A *value restriction* is a role

[4] "maps-to" corresponds to the NIKL relation "MostGeneralRole".

[5] Roles are seen as virtual objects in Loom, i.e., there are no structures in the system which can be identified as roles.

restriction which restricts the range of R_{CM}, while a *number restriction* is a role restriction which places bounds on the number of role fillers of R_{CM} that can be associated with a single instance of C. A composition of mappings M_1, ..., M_k such that the domain of M_1 is restricted to a particular concept (other than THING) is called a *role chain*.

Loom distinguishes between "primitive" and "defined" concepts (and relations). A concepts is *primitive* if no complete definition can be given for it (see [Vilain 84, p. 549 or , Brachman and Schmolze 85]); otherwise it is *defined*. Concepts and relations are organized into a taxonomy based on a partial-ordering relation called "specializes". A concept C_1 *specializes* a concept C_2 if and only if membership in C_1 entails membership in C_2, i.e. iff

$$\models \forall x[C_1(x) \rightarrow C_2(x)].$$

An instance of a specializes relation between two concepts may be declared explicitly in a concept definition, or it may be deduced by the classifier.

A *value* is an object which corresponds to a logical constant in a knowledge base, and is typically left undefined in a knowledge base. The numbers 1, 3, and 8.2, and the sexes Male and Female are examples of values. A concept which is defined by enumerating its instances is called a *set*. Currently, all of the sets we define in the TBox are sets of values. Number and Sex are examples of sets. A (denumerable) set for which precedessor and successor relations exist is termed an *interval*, e.g., Integer and Days-of-the-Week are intervals.

To *classify* a concept means to link it into the specialization lattice so that (i) it is below all concepts which it specializes, and (ii) it is above all concepts which specialize it. The *most specific generalization* (MSG) of a concept is the set of those concepts which are/would become its direct ancestors (parents) if it were classified.

To *recognize* an ABox object/instance x means to compute the set of concepts $\{C_i\}$ such that for each C_i, x is an instance of C_i, and x is not an instance of any descendant of C_i. The set $\{C_i\}$ is referred to as the MSG of x. In an informal discussion we may use the term "classification" to refer to either the classification or the recognition process.

4. The TBox

In this section we present the syntax and semantics of TBox definitions for (unary) concepts, (mapping) relations, sets and intervals. Occasionally within this discussion we will pause to point out some of the deductions which the Loom classifier will (or will not) be able to make. These comments are intended to foster an appreciation for what kinds of inference one can expect from a classifier. Next comes a brief discussion outlining our reasons for prohibiting cyclically-defined concepts, and we conclude with a presentation of three additional restrictions which Loom imposes on TBox definitions.

4.1. Defconcept and Defrelation

A formal semantics for the term-forming operations `defconcept` and `defrelation` appears as Appendix B. The simple definitional constructs listed in the figure can be combined within a concept or relation definition to form compound definitions. The semantics for such a compound definition are defined as the logical conjunction of the individual lambda definitions.

For example, referring to the Engines and Cars KB in Figure A-1, suppose we declare a new concept

```
(defconcept (:specializes Engine)
      (:restriction cylinders (:min 4) (:max 6))
      (:restriction fuel (:vr Gasoline))).
```

This concept means "an engine fueled by gasoline which has between 4 and 6 cylinders." The TBox classifier will discover that this concept specializes the concept labeled `Internal-Combustion-Engine`.

The Familial Relations KB in Figure A-4 illustrates how **defrelation** constraints can be combined to form terms for the relations **parent**, **father**, **grandfather**, etc. The classifier will determine, among other things, that **grandfather** specializes **grandparent** and that **parent** and **grandparent** specialize ancestors. A few short-hand notations are provided in addition to the operators illustrated in Figure A-4. The following pairs of forms are equivalent:

The forms

```
(:restriction M (:number k))                and
(:restriction M (:min k) (:max k)),
```

the forms

```
(:restriction (:vrdiff M C) ...)        and
(:restriction
    (defrelation (:specializes M) (:range C)) ...),
```

the forms

```
(:restriction (:closure-of M) ...)      and
(:restriction (defrelation (:closure-of M)) ...).
```

Loom's *constraint* clause extends the CNIKL construct referred to as a "role-constraint" or "role-value-map" by (1) allowing for other operators than just set-equality and set-containment, and (2) allowing a value to take the place of a role-chain. The argument "CP" in the clause

```
(:constraint CP (...) (...))
```

must name a relation which falls in the sublattice rooted at the relation **Compute-Relation**. Figure A-2 illustrates some compute relations. The operators for computing set-equality, set-inequality, and set-containment are other examples of compute relations.

Again referring to the Engines KB, let us declare two new concepts:

```
(defconcept Big-Engine
    (:constraint greater-than (horse-power) 120))
(defconcept Very-Big-Engine
    (:constraint greater-than (horse-power) 200))
```

We plan to upgrade the Loom classifier so that it will be able to deduce that **Very-Big-Engine** specializes **Big-Engine**. The analysis will necessitate recognizing the truth of **(greater-than 200 120)**, and will involve reasoning about the transitivity of the **greater-than** relation. During a 1986 NIKL users workshop [Moore 86], Ron Brachman discussed the possibility of extending a NIKL-like system to include a couple of new "boxes" in addition to the traditional TBox and ABox. One of those boxes he termed a "Mathematics Box", which would be a specialized reasoner with the ability to derive mathematical inferences in conjunction with the TBox reasoner. The numerical reasoning facility just hinted at represents an embryonic step in the direction of developing a full-fledged mathematics box.

We will conclude this section will an example containing definitions for which Loom cannot deduce the implied subsumption relations. Referring to the Familial Relations KB again, consider the following definitions of a concept named "Only-Child":

```
(defconcept Only-Child-1
    (:constraint equals self (parent child)))
(defconcept Only-Child-2
    (:restriction siblings (:max 0))).
```

The current Loom classifier cannot deduce that the concepts **Only-Child-1** and **Only-Child-2** are equivalent. The NIKL classifier is similarly unable to deduce this equivalence relation (when applied to CNIKL analogues of the above definitions). Our current development philosophy is that we are committed to developing a system which makes inferences which are sound, but not necessarily complete. One of the philosophical goals of the Loom system is to investigate empirically where the boundaries should be on the expressive power of a TBox. Once those bounds have been more-or-less established, it may be appropriate to revive the goal of developing a reasoner which is as complete as we can make it.

4.2. Defset and Definterval

This section describes the operators **defset** and **definterval**, which can be employed to define sets and intervals, and also to define concepts corresponding to the values enumerated in those sets/intervals. Our ex-

amples will reference the Sets and Intervals KB in Figure A-3.

In many cases, there is a tight coupling between values in a set or interval which represent "qualities" (e.g., the sex `Male` or the color `Red`) and concepts such as `Male-Animal` or `Red-Thing` which are defined by having one of their attributes restricted to the corresponding value: Definitions for `Male-Animal` and `Red-Thing` might be

```
(defconcept Male-Animal (:specializes Animal)
       (:restriction sex (:vr Male)))
(defconcept Red-Thing
       (:specializes Monochrome-Thing)
       (:restriction color (:vr Red))).
```

Thus, we have

$$\forall x[(Animal(x) \wedge sex(x, Male)) \leftrightarrow Male{-}Animal(x)].$$
$$\forall x[(Monochrome{-}Thing(x) \wedge color(x, Male)) \leftrightarrow Red{-}Thing(x)].$$

The Loom syntax for sets and intervals includes an optional "partitions" clause which produces the set of definitions needed to characterize this behavior.

The declaration

```
(defset Sex (:values Male Female))
```

defines a set `Sex` and the values `Male` and `Female`. To introduce the concepts `Male-Animal` and `Female-Animal`, we can augment our definition with the clause `(partitions Animal)` (Figure A-3 illustrates the complete definition). This larger declaration implicitly declares the following expressions:

```
(defrelation Sex :primitive
  (:axioms (:domain Animal) (:range Sex)))
(defconcept Male-Animal (:specializes Animal)
  (:restriction Sex Male))
(defconcept Female-Animal (:specializes Animal)
  (:restriction Sex Female))
```

In addition, the declaration for the concept `Animal` is augmented by a clause which indicates that `Male-Animal` and `Female-Animal` form a disjoint covering of Animal.

We next turn our attention to the interval `Naval-Rank` defined in Figure A-3. The declaration of `Naval-Rank` implies the definition of a relation `Naval-Rank`, and also implies the declaration of the concepts `Seaman-Recruit, Seaman-Apprentice, ... , Admiral.` [6] The implied declaration for `Admiral` is

```
(defconcept Admiral (:specializes Naval-Person)
   (:restriction Naval-Rank Admiral)).
```
[7]

Because Naval-Rank is specified as an interval, rather that as a set, the relations "successor" and "predecessor" are defined for its instances. Their definition corresponds to the order of values in the "values" clause. For example, `(successor Commander Captain)` is true. The successor and predecessor relations may appear within the role chains of a constraint clause. A square-bracket notation can be employed to define a (contiguous) subset of an interval. This is illustrated in the definition of the set `Naval-Officer-Rank`, and in the definitions below that for `Natural-Number`, `Positive-Integer`, and `Non-Negative-Integer`. The semantics of subsumption for intervals is the same as that for sets. For example, the interval defined by

```
(definterval (:specializes Integer)
       (:values 3 7 5))
```

specializes the interval defined as

```
(definterval (:specializes Integer)
       (:values [2..9])).
```

4.3. How to Avoid Cycles

A concept (or relation) definition *depends on* another definition if it references the other concept by name within its definition. If these depends-on links form a cycle, then we say that the definitions involved are *cyclic*. The designers of the NIKL system expressly permitted cyclic definitions. However, the semantics associated with cyclic CNIKL definitions was never fully worked out, and the behavior of the NIKL classifier when it encountered cycles was far from satisfactory. Loom has taken an opposite position -- cyclic definitions are illegal in Loom.

---

[6] In the declaration of "Naval-Rank", the clause "(:suffix Nil)" prevented the suffix "-Naval-Person" from being appended to each new concept.

[7] Observe that the concepts "admiral as person" and "admiral as naval-rank" have the same name. Loom will automatically add suffixes "-1" and "-2" to distinguish between them.

A primary motivation for allowing cycles was to avoid placing a restriction on what concepts could appear within a value restriction clause. Consider the following definition of Human:

```
(defconcept Human :primitive (:specializes Mammal)
        (:restriction parents (:vr Human)))
```

The value restriction `(:vr Human)` allows the system to infer "If an individual is Human, then so are its parents, and their parents, and so on." Because that value restriction is self-referential (defining a cycle of length one), it is not permitted in Loom. However, Loom does allow an equivalent restriction to be expressed as an axiom in the UBox:

```
(defconcept Human :primitive (:specializes Mammal)
    (:axioms
        (:restriction parents (:vr Human))))
```

Thus, we retain in Loom the ability to make statements such as, "the parents of humans are also human"; we just don't allow them to be included as a part of the (terminological) definition of a concept.

5. The UBox

The knowledge which we place in the Universal box augments individual TBox definitions with what we call universal or contingent knowledge. The expressive power of the Loom language increases significantly when the definitional language is extended to include expressions of universal knowledge. This combined language admits a correspondingly larger class of inferences.

This section will first define the different types of knowledge which we class as "universal". Next, we introduce the notion of a "stable" classifier, which serves to sharpen the definitional boundary between terminological and universal knowledge. Finally, we will present the representational model and classification algorithm adopted by the Loom architecture to handle universal knowledge.

5.1. Types of Universal Knowledge

In anticipation of our later discussion on how Loom represents universal knowledge, we will group our universal knowledge into four categories. Referring to universal knowledge that is attached to a concept "P", the categories are:

1. Contingent restrictions and constraints -- these are restrictions or constraints which necessarily apply to an instance "x" if P(x) holds. These are often called "necessary conditions";

2. Implications -- these are statements of the form "P implies Q" (where Q is a concept which does not subsume P). Often called "sufficient conditions";

3. Equivalences -- these are statements of the form "P if and only if Q". Often called "necessary and sufficient conditions";

4. Other non-definitional knowledge about concepts and relations. Currently this knowledge consists of covering relations, disjointness relations, marking concepts as "individual", and domain and range constraints on mappings.

5.1.1. Contingent Restrictions and Constraints

The "axioms" clause of a concept or relation definition states universal knowledge which applies to that concept or relation. The Engines and Cars KB of Figure A-1 illustrates several such clauses. The next few examples will be drawn from that KB.

The clause

```
(:axioms (:res (:vrdiff has-component Engine)
               (:number 1)))
```

which appears in the definition of `Car` is an example of a "contingent restriction". The meaning of the clause is

$$\forall x[Car(x) \rightarrow \exists\ exactly\ one\ y\ (has\text{-}component(x, y) \wedge Engine(y))].$$

This is sometimes referred to as a "necessary condition" because it translates as "it is *necessarily* the case that a car has exactly one engine." In general, the meaning of a restriction (or constraint) appearing within an "axioms" clause of a `defconcept` form defining a concept C is, "this restriction (constraint) applies to all objects which are instances of C".

5.1.2. Implications and Equivalence Relations

The clause `(:axioms (:implies Car))` which appears within the `defconcept` form which defines `Battery-Powered-Vehicle` is an example of an *implication*. Its meaning is

$$\forall x[Battery\text{—}Powered\text{—}Vehicle(x) \rightarrow Car(x)].$$

This form of knowledge is sometimes called a "sufficient condition" because it can be translated as "to determine is x is an Car, it is *sufficient* to determine that x is a Battery-Powered-Vehicle."

It is important to distinguish the difference in semantics between an implication (an "implies" relation) and a "specializes" relation. While the logical form associated with each of them is identical, the semantics of the specializes relation is significantly stronger. The statement "B specializes A" says not only that (1) B *implies* A, but also that (2) B's (TBox) *definition* includes the definition of A, and (3) B *inherits* the (UBox) properties of A.

A two-way implication established between a pair of concepts defines an *equivalence* relation. More generally, any cycle of implications through a set of concepts establishes an equivalence relation between each pair of concepts in that set. Suppose a set of concepts $\{C_i\}$ have been defined such that they are pairwise-equivalent. While the TBox sees the C_i as distinct concepts, the UBox view of this knowledge sees a single concept C_U which combines all of the knowledge declared in each of the C_i (this is described in more detail in section 5.3.). This means that universal knowledge (other than the "implies" relations) can be distributed in any number of ways among the C_i's, and the semantics will always be the same.

The *preferred* way to model a set of equivalent concepts $\{C_i\}$ is to explicitly declare an additional concept C which specializes each of the C_i, and which contains all of the universal knowledge associated with the C_i, except for a clause `(:axioms (:implies C))` which appears in each of the C_i definitions. Our definition of the concepts `Diesel-Oil-Engine`, `Thing-With-Glow-Plugs`, `Very-High-Compression-Engine`, and `Diesel-Engine` in Figure A-1 illustrates this type of modeling.

5.1.3. Coverings and Disjointness Classes

A *covering* for a concept "A" is a set of concepts whose union contains A. Loom syntax requires that the concepts within such a covering specialize A, so that the union of the covering concepts *equals* A. The meaning of the clause `(:axioms (:covering B C))` within a `defconcept` for A is

$$\forall x[A(x) \rightarrow (B(x) \vee C(x))].$$

Declarations of unary coverings (coverings containing a single concept) are illegal in Loom because they are logically equivalent to "implies" relations, and hence are redundant.

A *disjointness class* is a set of concepts which are declared to be mutually disjoint. A disjointness class is always defined with respect to a concept which subsumes the members of the class. The meaning of the clause

`(:axioms (:disjoint B C))`

within a `defconcept` for A is

$$\forall x[B(x) \leftrightarrow \neg C(x)].$$

A *disjoint-covering* of a concept A enumerates a set of concepts which partition A, i.e., it is interpreted as the logical conjunction of a covering declaration and a disjointness declaration.

The Numeric-Comparison KB in Figure A-2 illustrates some declarations of coverings and disjoint-coverings. The covering defined for the relation `numeric-comparison` declares that the relations `greater-or-equal` and `less-or-equal` cover `numeric-comparison`. The disjoint-covering declaration for `greater-or-equal` states both that `greater-or-equal` is covered by `greater-than` and `equal`, and that the relations `greater-than` and `equal` are disjoint. Loom

provides functions for asking questions about (declared or derived) disjointness and covering relations, such as "Are concepts A and B disjoint?", "Do concepts A, B, and C cover concept D?", or "List all coverings for concept D".

Loom requires that the concepts or relations appearing in a covering, disjointness class, or disjoint-covering must all be *primitive*. The philosophical justification for this restriction is that if one or more of the members of the covering and/or disjointness class are not primitive then either (i) the covering and/or disjointness relations could have been logically inferred on the basis of other knowledge or (ii) such relation(s) could be derived. In the former case, the covering and/disjointness declaration is redundant, and should be dropped. In the latter case, there must have been something left unstated about the non-primitive concepts, which suggests that they are in fact primitive.

The implementors of the NIKL system encountered a practical reason for requiring members of a disjointness class to be primitive. That restriction prevented an anomaly which arose in a situation in which so-called "incoherent" concepts were being classified. The possibility of a similar anomaly arising in conjunction with covering declarations has not yet been explored.

We are considering omitting the `disjoint` clause altogether from the Loom language, owing to the observation that we have not yet encountered the use of a disjointness declaration in a context where an obvious covering relation did not also exist, i.e., where `disjoint-covering` could not have been substituted. Our syntactic requirement that a disjointness class be defined relative to a particular concept anticipates this future restriction.

5.1.4. Domain and Range Restrictions

"Domain" and "range" clauses which appear within an "axioms" clause state necessary conditions about a relation. The declaration

```
(defrelation M ...
    (:axioms (:domain A) (:range B)))
```

makes the universal statement

$\forall xy[M(x,y) \rightarrow A(x) \wedge B(y)]$.

Knowledge about domain and range constraints is referenced during the "model-building" phase, when the initial definitions of concepts and relations are being refined and checked for coherence. In this context, these constraints function as "integrity constraints".

5.1.5. Individual Concepts

Marking a concept as "individual" means that its extension has cardinality at most one. We have identified some inferences that can be made on the basis of individual markings on concepts, but none of these inferences are particularly useful. Thus, this feature currently serves only as a place-holder, awaiting a user who will conceive of a use for it.

The presence of the "individual" marking is a part of Loom's NIKL heritage. Because most applications of NIKL operated without an ABox -- individual concepts served in lieu of real ABox instances.

5.2. Stable and Non-Stable Classifiers

Consider the following scenario: A rather shady-looking character produces from his capacious overcoat a large black box, which he claims is a seventh-generation classifier of terminological knowledge, guaranteed to produce sound (although not necessarily complete) inferences very quickly. We decide to test out his BBC (black-box classifier). First we store into the BBC the definitions of two concepts which we call A and B. We than ask the BBC "Does B specialize A?", and it responds (very quickly) "No". Next, we enter a few more definitions into the BBC, and again ask the BBC "Does B specialize A?" This time, its rapid rejoinder is "Yes"! Should we buy his BBC (his price is *very* reasonable)?

The answer is no: Let us define a ***stable*** classifier (or recognizer) to be one which produces the same answers to subsumption questions independently of additions or subtractions to/from the knowledge base (here we assume that no concept definitions are *modified*, and that at no time does the knowledge base contain undefined references). "Stability" is a highly-desirable feature in a TBox, because it provides a certain guarantee that when TBox knowledge is shared across several knowledge bases (e.g., by several applications) it will retain the same "meaning" in each of those contexts. We propose that "stability" be considered a test which serves to exclude some reasoners from being considered TBox classifiers. ("Soundness" should be another TBox requirement). The Loom Tbox of an example of a stable classifier; our friend's BBC is not stable.

The Loom UBox classifier/recognizer is not stable! Consider the Cars KB in Figure A-1. Suppose we make the following assertions

```
(assert (Motor-Vehicle BPV) (2-Person-Vehicle BPV)
        (Battery-Powered-Engine E)
        (has-component BPV E))
```

Now we ask, "Is BPV an instance of 2-Person-Car?" The UBox recognizer will make the following inferences

```
(Battery-Powered-Vehicle BPV)
(Car BPV)                    because of the "implies" axiom
(2-Person-Car BPV)
```

and conclude "Yes". However, if we remove the definition for `Battery-Powered-Vehicle` (or if it never existed) and re-run the UBox recognizer, it will not conclude either `(Car BPV)` or `(2-Person-Vehicle BPV)`.[8] On the other hand, if we run the Loom TBox recognizer on the same knowledge base and assertions, it will fail in both cases to recognize that BPV is a car (or a 2-person car). This behavior occurs because the axiom "`Battery-Powered-Vehicle` implies `Car`" is invisible to the TBox. The stability of the TBox classifier derives from the restrictions we place on what kinds of knowledge are classed as terminological in the TBox, not from the particular inference algorithm chosen -- we deliberately exclude from the TBox classes of knowledge which introduce non-stable behavior.

[8]Note: This does *not* mean that it concludes "¬(Car BPV)". It merely *fails* to infer "(Car BPV)" -- the UBox classifier is not a nonmonotonic reasoner.

5.3. Modeling and Classification of Universal Knowledge

This section represents a long engineering note. We first describe the internal model adopted by Loom to represent universal knowledge, and then give some insight into the workings of Loom's UBox classification algorithm.

In a Loom concept network, separate objects, which we shall refer to as C_T and C_U, are defined to represent the TBox and UBox knowledge associated with a single concept C.[9] C_T contains exactly the definitional (terminological) component of C. C_U contains both the definitional *and* contingent knowledge knowledge associated with C. Thus, by construction, C_U always specializes C_T. An *implies* link links C_T to C_U, and has the meaning $\forall x[C_T(x) \rightarrow C_U(x)]$. In other words, C_T implies C_U.

Within a UBox concept, contingent restrictions and constraints are merged into a single definition, and are classified according to that definition. Suppose, for example, that we made the following declarations:

```
(defconcept C (:restriction R (:min 1))
   (:axioms (:restriction S (:min 1))))
(defconcept D (:restriction R (:min 1))
    (:restriction S (:min 1)))
```

[9]Browsers of Loom knowledge bases should be aware of the following: Loom maintains separate *name spaces* for TBox objects and UBox objects. In the TBox name space, only TBox objects are visible, and C_T has the name "C". In the UBox, only UBox objects are visible, and C_U has the name "C".

The classifier cannot distinguish between the objects C_U and D_T, and hence will merge these two concepts.

Implications are modeled as follows: Suppose we declare

```
(defconcept A ...
   (:axioms (:implies B)))
```

Rather than placing, say, an "implies" link between A_U and B_U, Loom captures the semantics of the implication axiom by merging all of the knowledge in B_U into A_U (in effect, "compiling out" the "implies" link). Equivalence relations add nothing new to the model, since they just consist of cycles of implication relations. If we declared that "A implies B", and also that "B implies A", Loom would merge B_U into A_U, and would also merge A_U into B_U, making A_U and B_U identical. The classifier would then merge these into a single UBox concept.

Loom's internal model of three of our original four categories of universal knowledge can thus be accomplished with the addition of only one new link, the "implies" link.[10] An important property of the model is that, in all cases, the "implies" links connect more general concepts to more specific ones: The Loom (and NIKL) TBox classifiers operate by picking an initial set of "starting points" (concepts) and then traversing down "subC" links which connect each concept to those concepts which directly specialize it. Loom's UBox classifier traverses down both "subC" and "implies" links. Because the "subC" and "implies" links form an acyclic directed graph, termination of the UBox classifier is guaranteed.

During the process of classifying/recognizing an object X in the UBox, the traversal of an "implies" link can cause knowledge to be acquired about X which is not entailed by its definition. This is the source of the "non-stability" in the UBox classifier. Recall the example in section 5.2 which traced the recognition of the object "BPV". One of the algorithm's starting points is the concept `2-Person-Vehicle`. If we visit its child `2-Person-Car` and make the test `(2-Person-Car X)` before having traversed the "implies" link between `Battery-Powered-Vehicle`$_T$ and `Battery-Powered-Vehicle`$_U$, we would receive a negative answer. Traversing that link causes us to acquire the knowledge `(Car BPV)`. After this point, the test `(2-Person-Car X)` returns in the affirmative. Hence, the first test to see if X was a 2-person car represented wasted effort.

One practical consequence of non-stability is that the ordering of subsumption tests is more critical for UBox classification than for TBox classification. Furthermore, it is not always the case that careful ordering of subsumption tests can avoid the necessity to repeat some subsumption tests (unless you have an "oracle" at your disposal). Theoretically, UBox classification could be significantly slower than TBox classification. We have not yet performed empirical tests which compare the relative performance of the two algorithms, but we expect that we will be able to achieve reasonable performance from the UBox.

6. Default Knowledge

Loom establishes a separate "box" for representing "default knowledge" -- knowledge representing statements that are "typically" true, but which are not axiomatic. Conceptually, this default knowledge consists of rules of the form "if nothing has been asserted or deduced which contradicts X, then assume X".

We will first discuss why the Loom architecture includes a Default Box. Then we will examine the semantics of the default value and closed-world-assumption constructs. Finally, we will preview what the operation of a non-monotonic classifier might look like.

[10] The fourth category "other" is handled by special-purpose data structures and algorithms which are outside of the scope of this discussion.

6.1. The Case for a Default Box

We reject the idea of combining assertional and default knowledge into a single "non-monotonic ABox". Such a strategy would contradict a philosophical goal of the Loom architecture: We wish to reserve the ABox for statements about *individuals,* and to extend the representational power of the non-ABox portion of the system so that all statements about "classes of individuals" can be represented somewhere else other than in the ABox. The nature of default knowledge is that it generally makes statements about classes of individuals. Thus, we must consider what the implications are of developing yet another box.

The prerequisites for defining a new "box" in the Loom knowledge representation framework are that (i) we can identify a significant body of knowledge which would be assigned to that box, and (ii) a specialized reasoning facility must exist to process the inferences associated with this knowledge. The Loom system does not yet meet these requirements, because it is able to respond to only two very specialized forms of default knowledge -- it includes a limited treatment of default values, and it recognizes certain closed-world assumptions. On the other hand, we already have some idea of what a (much more general) non-monotonic classifier would look like. Its behavior is sketched below, in section 6.3. Therefore, we anticipate that both prerequisites will be met in a future version of Loom.

6.2. Default Values and Closed-World Assumptions

A "default value" is a value which is assigned to fill a role/slot for some individual in the absence of any explicitly-asserted (or derived) knowledge about that role filler. For example, in our Engines and Cars KB, the form

```
(:defaults (:restriction type-of-fuel
                          (:vr Gasoline)))
```

in the `defconcept` declaration for `Internal-Combustion-Engine` declares that Gasoline is the default value for the role `type-of-fuel`. If for some constant "x" we have asserted `(Internal-Combustion-Engine x)`, and we have made no assertions of the form `(type-of-fuel x f)`, then the default assumption is `(type-of-fuel x Gasoline)`.

The act of assigning a default value can trigger a re-classification of an ABox object. For example, after making the assertion `(assert Elephant E1)`, the process of classifiying `E1` as an elephant could trigger a default assertion `color E1 Grey`, which might then cause E1 to be re-classified as a grey-elephant (if such a concept existed). We have yet to investigate whether default values may trigger cycles of reclassifications, and, if so, how the semantics of assigning default values should be restricted to prevent such cycles.

The Loom representation of closed-world assumptions is another example where we can elicit useful default behavior in the absence of a general-purpose non-monotonic classifier. Each ABox knowledge base is assumed to have either a "closed-world" or an "open-world" interpretation. "Open-world" means that in addition to the assertions about an individual that are explicitly stated in the knowledge base, there may be other relevant assertions which have been left unstated. For example, consider the Engines and Cars KB once again. Suppose we make the assertions

```
(assert (Internal-Combustion-Engine e)
        (Cylinder c1) (Cylinder c2)
        (Cylinder c3) (Cylinder c4)
        (cylinder e c1) (cylinder e c2)
        (cylinder e c3) (cylinder e c4))
```

Can we deduce `(4-Cylinder-Engine e)`? The answer is no if we adopt an open-world assumption, because the possibility exists that there are 4 (or 12, or whatever) more cylinders which are also components of the engine "e". On the other hand, adopting a closed-world assumption would allow us to conclude that the four

cylinders which are components of "e" are the only ones that exist, in which case the inference `(4-Cylinder-Engine e)` is valid.

Loom allows one to declare selective "regions" of closed-world semantics within an open-world knowledge base: The declaration

```
(defrelation M ...
   (:axioms (:domain D))
   (:defaults :closed-world-assumption))
```

has the following interpretation: "If "D(x)" has been asserted (or can be deduced) for some x, then for all y, "M(x, y)" is true only if it has been explicitly asserted, or can be derived." The `defrelation` declaration for the relation `cylinder` in the Engines and Cars KB includes such a closed-world assumption. This assumption allows the classifier to count instances of the `cylinder` relation when attempting to recognize an object as an instance of the concept `4-Cylinder-Engine`.

6.4. Preview of a Non-Monotonic Classifier

A non-monotonic classifier has not yet been developed for the Loom architecture. We provide here a preview of what its behavior will be like if and when it is constructed, with the intention of stimulating the demand for such a reasoner. Our example provides an illustration of how a classic problem in non-monotonic reasoning can be modeled by the Loom language.

In the process of classifying/recognizing an object "x", a non-monotonic classifier will reference both explicitly declared knowledge and default knowledge about "x", and hence may deduce classifications which are based on default assumptions. As is the case with UBox classification, the classifier may acquire additional information about "x" in the midst of the classification process. The possibility arises that the "acquired" knowledge will contradict one (or more) of the default assumptions. In this case, the classifier must retract any classifications it has already made which were based on these non-valid assumptions.

Consider the Birds KB in Figure A-5. Suppose we have made the assertion

```
(assert (Penguin Tweety))
```

The classifier may first deduce `(Bird Tweety)`, then pick-up the attached default implication and assume `(Flying-Animal Tweety)`, and then deduce `(Flying-Bird Tweety)`. Next, it may deduce `(Non-Flying-Animal Tweety)` from the definition of `Penguin`, and then discover that `Flying-Animal` and `Non-Flying-Animal` are disjoint. At this point, it must retract the earlier deductions `(Flying-Animal Tweety)` and `(Flying-Bird Tweety)`.

7. Conclusion

The Loom language introduces new expressivity and some new and powerful forms of inference into the KL-ONE paradigm for knowledge representation. The most significant achievement is the formulation of the UBox, which allows universal knowledge to be defined and reasoned about independently of the terminological knowledge. The UBox solves a long-standing problem of how to represent necessary and sufficient conditions, and provides a way for a user to introduce cyclic references into a knowledge base without derailing the classifier.

Looking towards the future, we have described the behavior of a Default Box, indicating how a classifier might be extended to perform non-monotonic classifications. Collectively, our results suggest that we have taken another step in an ongoing evolution of knowledge representation systems, wherein increasing numbers of specialized forms of reasoning can be organized within a principled knowledge representation framework.

References

[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.

[Kaczmarek 86] T. Kaczmarek, R. Bates, G. Robins, "Recent Developments in NIKL," in *AAAI-86, Proceedings of the National Conference on Artificial Intelligence*, AAAI, Philadelphia, PA, August 1986.

[Mac Gregor 87] Robert Mac Gregor and Raymond Bates, The Loom Knowledge Representation Language, 1987. (in preparation)

[Moore 86] Johanna D. Moore, NIKL Workshop Summary, 1986.

[Moser 83] M.G. Moser, "An Overview of NIKL, the New Implementation of KL-ONE," in *Research in Natural Language Understanding*, Bolt, Beranek, and Newman, Inc., Cambridge, MA, 1983. BBN Technical Report 5421.

[Robins 86] Gabriel Robins, The NIKL Manual, 1986.

[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.

[Vilain 84] Marc Vilain, *KL-TWO, A Hybrid Knowledge Representation System*, Bolt Beranak and Newman, Technical Report 5694, September 1984.

[Vilain 85] M. Vilain, "The Restricted Language Architecture of a Hybrid Representation System," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 547-551, Los Angeles, CA, August 1985.

[von Luck 87] K. von Luck, B. Nebel, C. Peltason, A. Schmiedel, *The Anatomy of the BACK System*, Technische Universitat Berlin, Technical Report KIT Report 41, January 1987.

A. Knowledge Bases

Engines and Cars Knowledge Base

```
(defrelation has-component :primitive (:inverse-of component-of))
(defrelation component-of :primitive)

(defconcept Horse-Power :primitive)
(defrelation horse-power (:range Horse-Power))

(defconcept Fuel :primitive)
(defrelation type-of-fuel (:range Fuel))
(defconcept Gasoline :primitive (:specializes Fuel))
(defconcept Diesel-Oil :primitive (:specializes Fuel))

;;; Engines
(defconcept Engine :primitive
  (:axioms (:restriction type-of-fuel (:number 1))
    (:restriction horse-power (:number 1))))

(defconcept Cylinder :primitive)
(defrelation cylinders (:specializes has-component) (:range Cylinder)
  (:defaults :closed-world-assumption))

(defconcept Internal-Combustion-Engine (:specializes Engine)
        (:restriction cylinders (:min 1))
  (:defaults
    (:restriction type-of-fuel (:vr Gasoline))))
(defconcept 4-Cylinder-Engine (:specializes Engine)
        (:restriction cylinders (:number 4)))

;;; Diesel-Engines
(defconcept Glow-Plug :primitive)
(defrelation compression-ratio :primitive
  (:axioms (:domain Internal-Combustion-Engine) (:range Integer)))
(defconcept Diesel-Oil-Engine (:specializes Engine)
        (:restriction type-of-fuel (:vr Diesel-Oil))
  (:axioms
    (:implies Diesel-Engine)))
(defconcept Thing-With-Glow-Plugs
        (:restriction (:vrdiff has-component Glow-Plug) (:min 1))
  (:axioms
    (:implies Diesel-Engine)))
(defconcept Very-High-Compression-Engine
        (:constraint greater-than (compression-ratio) 15)
  (:axioms
    (:implies Diesel-Engine)))
(defconcept Diesel-Engine :primitive
        (:specializes Internal-Combustion-Engine Diesel-Oil-Engine
                     Thing-With-Glow-Plugs Very-High-Compression-Engine))
(defconcept Battery-Powered-Engine :primitive (:specializes Engine))

;;; Cars
(defconcept Vehicle :primitive)
(defconcept Motor-Vehicle (:specializes Vehicle)
       (:restriction (:vrdiff has-component Engine) (:number 1)))
(defconcept Battery-Powered-Vehicle (:specializes Motor-Vehicle)
       (:restriction (:vrdiff has-component Engine) (:vr Battery-Powered-Engine))
   (:axioms (:implies Car)))
(defrelation occupants :primitive (:range Human))
(defconcept 2-Person-Vehicle (:specializes Vehicle)
       (:restriction occupants (:max 2)))
(defconcept Car :primitive (:specializes Vehicle)
  (:axioms
    (:implies Motor-Vehicle)))
(defconcept 2-Person-Car (:specializes Car 2-Person-Vehicle))
```

Figure A-1: Engines and Cars

Numeric Comparison Knowledge Bases

```
;;; Numeric Comparison Predicates
(defrelation numeric-comparison :primitive (:specializes Compute-Relation)
  (:axioms
     (:domain Real-Number) (:range Real-Number)
     (:covering greater-or-equal less-or-equal)))
(defrelation greater-than :primitive (:specializes greater-or-equal not-equal)
  (:annotation
    (:membership-test (lambda (domain range) (> domain range)))))
(defrelation less-than :primitive (:specializes less-or-equal not-equal)
  (:annotation
    (:membership-test (lambda (domain range) (< domain range)))))
(defrelation equal :primitive (:specializes greater-or-equal less-or-equal)
  (:annotation
    (:membership-test (lambda (domain range) (eql domain range)))))
(defrelation not-equal :primitive (:specializes numeric-comparison)
  (:axioms
    (:disjoint-covering greater-than less-than)))
(defrelation greater-or-equal :primitive (:specializes numeric-comparison)
  (:axioms
    (:disjoint-covering equal greater-than)))
(defrelation less-or-equal :primitive (:specializes numeric-comparison)
  (:axioms
    (:disjoint-covering equal less-than)))
```

Figure A-2: Numeric Comparison

Sets and Intervals Knowledge Base

```
;;; Sex
(defconcept Animal :primitive)
(defset Sex (:values Male Female) (:partitions Animal))

;;; Navy Rankings
(defconcept Navy-Person :primitive)
(defconcept Military-Rank :primitive)
(defrelation Rank (:range Military-Rank))
(defrelation Naval-Rank :primitive (:specializes Rank)
   (:axioms (:domain Navy-Person) (:range Naval-Rank)))

(definterval Naval-Rank :primitive (:specializes Military-Rank)
  (:values Seaman-Recruit Seaman-Apprentice Seaman Petty-Officer-Third-Class
           Petty-Officer-Second-Class Petty-Officer-First-Class Chief-Petty-Officer
           Senior-Chief-Petty-Officer Master-Chief-Petty-Officer
           Ensign Lieutenant-Junior-Grade Lieutenant Lieutenant-Commander
           Commander Captain Commodore Rear-Admiral Vice-Admiral Admiral)
  (:partitions Navy-Person (:suffix nil)))
(defset Naval-Officer-Rank (:specializes Naval-Rank) (:values [Ensign..Admiral]))

;;; Numbers
(defconcept Real-Number :primitive
  (:annotation
    (:membership-test (lambda (self) (numberp self)))))

(definterval Integer :primitive (:specializes Real-Number)
  (:values [-INFINITY..INFINITY])
  (:annotation
    (:membership-test (lambda (self) (integerp self)))
    (:predecessor-fn (lambda (self) (1- self)))
    (:successor-fn (lambda (self) (1+ self)))))
(definterval Natural-Number (:specializes Integer) (:values [0..INFINITY]))
(definterval Positive-Integer (:specializes Integer) (:values [1..INFINITY]))
(definterval Non-Negative-Integer (:specializes Integer)
   (:values [-INFINITY..-1] [1..INFINITY]))
```

Figure A-3: Sets and Intervals

Familial Relations Knowledge Base

```
;;; Person
(defconcept Person :primitive)

;;; Familial Relations
(defrelation parent :primitive
  (:axioms (:domain Person) (:range Person)))
(defrelation father (:specializes parent) (:range Male))
(defrelation grandparent (:composition-of parent parent))
(defrelation grandfather (:composition-of parent father))
(defrelation ancestor (:closure-of parent))
(defrelation child (:inverse-of parent))
(defrelation sibling (:composition-of parent child) (:specializes not-equal))
(defrelation brother (:specializes sibling) (:range male))
```

Figure A-4: Familial Relations

Birds Knowledge Base

```
(defconcept Animal :primitive
   (:axioms (:disjoint-covering Flying-Animal Non-Flying-Animal)))
(defconcept Flying-Animal :primitive (:specializes Animal))
(defconcept Non-Flying-Animal :primitive (:specializes Animal))

(defconcept Bird :primitive (:specializes Animal)
   (defaults (:implies Flying-Animal)))
(defconcept Flying-Bird (:specializes Bird Flying-Animal))
(defconcept Penguin :primitive (:specializes Bird Non-Flying-Animal))
```

Figure A-5: Birds

B. Semantics of Term-Defining Constructs

Loom Expression, e	Semantics of e, [[e]]
(defconcept (:specializes C_1 C_2))	$\lambda x.\ [[C_1]](x) \wedge [[C_2]](x)$
(defconcept (:restriction M (vr C)))	$\lambda x.\ \forall y([[M]](x, y) \rightarrow [[C]](y))$
(defconcept (:restriction M (min n)))	$\lambda x.\ \exists$ n *distinct* $y_i \wedge_i [[M]](x, y_i)$
(defconcept (:restriction M (max n)))	$\lambda x.\ \nexists$ n+1 *distinct* $y_i \wedge_i [[M]](x, y_i)$
(defconcept (:constraint CR (R_1 R_2) (S_1 S_2)))	$\lambda x.\ \forall y,z([[R_1]]\circ[[R_2]](x, y) \wedge [[S_1]]\circ[[S_2]](x, z)) \rightarrow [[CR]](y, z))$
(defconcept (:constraint CR (R_1 R_2) v))	$\lambda x.\ \forall y([[R_1]]\circ[[R_2]](x, y) \rightarrow [[CR]](y, v))$
(defrelation (:specializes M_1 M_2))	$\lambda x,y.\ [[M_1]](x, y) \wedge [[M_2]](x, y)$
(defrelation (:range C))	$\lambda x,y.\ [[C]](y)$
(defrelation (:inverse-of M))	$\lambda x,y.\ [[M]](y, x)$
(defrelation (:closure-of M))	$\lambda x,y.\ [[M]]^{+}(x, y)$
(defrelation (:composition-of M_1 M_2))	$\lambda x,y.\ [[M_1]]\circ[[M_2]](x, y)$

A Framework for Situation Assessment: Using Best-Explanation Reasoning To Infer Plans from Behavior

John R. Josephson

The Ohio State University
Columbus, Ohio

Abstract

We propose a computational framework for battlefield situation assessment, describing how the relevant knowledge can be organized, represented, and used in the service of problem solving. We describe how the reasoning processes can be controlled to avoid excessive and impractical amounts of search, and indicate how the computations can be distributed in a natural way to spread the burden over a community of separate processors and processing sites. This design extends previous work done at The Ohio State LAIR on diagnostic reasoning and representation of plan understanding, and applies it to the particular military information-processing problem, a species of the more general intellectual task of inferring plans and intentions from behavior.

Introduction

Nowhere is there a more compelling need for an AI system, than to multiply the effectiveness of the forces defending Western Europe against possible aggression from the Eastern Block. As this report is written the United States and the Soviet Union are actively engaged in negotiations whose avowed object is the elimination of medium-range nuclear weapons from Europe. Yet it has become increasingly clear that that the prospects for a more general nuclear disarmament are severely limited by the NATO allies' perceived need to rely on tactical nuclear weapons to defend Western Europe against the numerically superior forces of the Soviet Union and its allies. Thus any technological innovation that can contribute to multiplying the non-nuclear defensive effectiveness of the western allies (or at least to their apparent effectiveness) can contribute markedly to nuclear disarmament at the low end of weapons yield, and thus to breaking the path of escalation from convention to nuclear war. Equipping NATO field commanders with AI systems that enhance their ability to respond quickly and cleverly to developments on the battlefield would constitute just such a technological innovation.

This paper proposes a computational framework for battlefield situation assessment, the task of inferring the plans and objectives of adversaries and other players on the battlefield. we describe how a number of the important kinds of knowledge needed for the task can be organized, represented, and brought to bear at the right times to contribute to the problem-solving. Several distinct but interacting types of reasoning are needed, including: ***abduction*** or "best explanation" reasoning; ***planning*** (for the other guy) including ***route planning***, ***resource allocation planning***, ***tactical goal choosing***, ***plan schema instantiation***; and also ***map-based spatial reasoning*** about proximity, avenues of approach, formation, striking ranges of weapons, importance of various terrain features, and the like. The foregoing is not a complete list, and the list is heterogeneous with respect to level of abstraction. For example abduction, more specifically, the ***assembly of composite explanatory hypotheses***, is needed for the task of "***diagnosing***" enemy ***plans*** by trying to produce coherent composite explanations of his behavior. On our view ***classification reasoning*** is also needed in the service of plan diagnosis, to organize the reasoning processes whereby precompiled high-level plan schemata are recognized as plausibly useful for the interpretations of certain actions. That is, classification is needed for ***plan recognition***.

Throughout the design process we have been concerned to structure the control strategies to avoid excessive and impractical amounts of search. Search processes in this domain could easily get out of hand, and a formal definition of the problem space would show that the space is multi-dimensionally combinatorially explosive.[1] In order to control search there seems to be no escape from the necessity of

[1] The hypothesis composition problem alone is explosive in at least two dimensions [1].

decomposing the problem into manageable subtasks, *chunking* the knowledge base into meaningful and moderately sized units, *organizing* the knowledge for use by the problem solving processes, and *modularizing* the reasoning so that reasoning strategies can be tailored to the reasoning tasks. In short, the answer to how to avoid impractical amounts of search, is that we must at all costs avoid or control complexity. Since the problem set for the system is by its nature complex, our recourse must be to redefine the problem wherever it makes sense to do so, and modularize, modularize, modularize.[2] Moreover not just any modularization will do. Modules must perform meaningful and accomplishable functions in the system, and be able to act semi-autonomously without explosive amounts of interaction, or they will not really contribute to controlling the complexity.

It appears that one nice effect of this extreme modularity of the problem solving is that parallel and distributed implementations of the design are possible, not only taking advantage of the computing power of multiprocessor architectures, but also distributing responsibility for portions of the problem, in a natural fashion, to geographically distributed processing sites. In short we suggest giving each local commander, at each level of the organization, his own semi-autonomous decision support situation assessment system. The individual systems, through intermittent communication up, down, and locally sideways through the command hierarchy, will integrate into a larger system, itself more than the sum of its separate parts.

This design extends previous work done at The Ohio State LAIR on diagnostic reasoning [4, 12, 16] and on representing plan understanding [14, 5], and applies it to the particular military information-processing problem, a species of the more general intellectual task of inferring plans and intentions from behavior.

In the terms of this paper *situation assessment* is distinguished from *data fusion* as a distinct stage of military information processing. The data fusion process takes raw intelligence reports and produces a description of the battlefield situation in terms of various **actors**, at various scales of discrimination, with information about their identities, types, actions, locations, and motions. The fusion process takes care of counting and classifying the various actors, tracking them over time, unifying reports about the same actor coming in at different times, attributing actions to the correct actor, maintaining descriptions of actor states, and so on. In summary, the output of the fusion process is a description of the actors and their "observed" behavior.

[2] See Herbert Simon's essay on "The Architecture of Complexity" [18] for an explanation of why hierarchical decomposition is such a useful and general strategy.

We can think of this fusion output as being presented on a map board which is constantly updated to reflect the most recent conclusions. On this board appear symbols representing the actors and their locations. We can imagine that the board is automated to allow us to zoom in and out and examine the situation at various granularities of resolution. Each symbol on the board packages a rich data structure containing not just identity, tracking history, recent behavior, classification of the actor as to type (e.g. artillery battalion), pointers to its parts, and so on; but beyond that, each actor symbol on the board indexes directly or indirectly into everything that is known (or surmised) about that actor. Uncertainties are represented explicitly, (e.g. that this is probably the 33rd rifle brigade, but it might be the 44th.) To keep things manageable, the number of alternative values for a given attribute should be kept small, on the order of a best estimate plus one, or at most two, alternative values. While most of the information flow will be from the fusion process, through the board, to the situation assessment process, we can allow for the possibility of some information flowing the other way. For example the fusion process might judge that a unit is probably of type A, with type B as an available alternative interpretation; then subsequent situation assessment reasoning may find that it would make no coherent sense for the unit to be of type A, but that it does make sense for it to be of type B, and this information can be passed back to the board.

The fusion process is itself a difficult problem for an AI system to address, but taking it for granted anyway (it might be automated or not) this report describes a system that will take the output from the fusion process, and use it to infer an adversary's plans and intentions. Thus the function of the situation assessment process can be described as "plan diagnosis". Since the input to the assessment process is information about actors and their behavior, and the output can be thought of as a coherent explanation of that behavior, we can see the whole process as being a form of "best explanation" reasoning or *abduction*.

"Abduction"

Abduction or Inference to the Best Explanation is a form of inference that follows a pattern something like this:

> D is a collection of data (facts, observations, givens),
> H explains D (would, if true, explain D),
> No other hypothesis explains D as well as H does.
> ---
> Therefore, H is (probably) correct.

The strength of an abductive conclusion will in general depend on several factors, including:

- how good H is by itself, independently of considering the alternatives,
- how decisively H surpasses the alternatives,
- how thorough the search was for alternative explanations, and
- pragmatic considerations, including
  - the costs of being wrong and the benefits of being right,
  - how strong the need is to come to a conclusion at all, especially considering the possibility of seeking further evidence before concluding.

Abductions, as we have just characterized them, go *from data* describing something *to an explanatory hypothesis* that best accounts for that data.

Notice that calling an inference "abduction" carries with it the idea of its goal: a best explanation. Contrast this with characterizing an inference as "deduction", which carries instead the idea of a constraint that is satisfied: that the inference is guaranteed to be truth-preserving. Since there is no intrinsic incompatibility between explanatory goals and truth-preservation constraints, it is conceivable for there to be deductive abductions. In fact, if all of the alternative ways of explaining something are exhaustively enumerated, and all but one of the explanations are decisively eliminated, the overall pattern of inference is deductively valid.

Arguably abduction is itself an epistemologically fundamental form of reasoning, not reducible to deduction, probabilistic induction, or any combination of them [9, 10]. But whether or not abductions can be justified on logical grounds, they appear to be ubiquitous in the un-selfconscious reasonings, interpretations, and perceivings of ordinary life, and in the more critically self aware reasonings upon which scientific theories are based [10].

It is a common view that diagnostic reasoning in general is abduction [6, 15, 17]. The idea is that the task of a diagnostic reasoner is to come up with a best explanation for the set of symptoms. In this paper we take the point of view that the overall plan-diagnosis situation-assessment task is best understood as a form of abduction.[3]

"Explanation"

There are numerous senses of the term "explanation" in common use; several are relevant to this paper. At one extreme we may speak of a scientific theory explaining some physical phonemenon, as for example, how Newton's Theory of Gravitation explains the tides. In related senses we may speak of explaining historical events by economic theories, explaining some human behavior using a theory of motives, and explaining the actions of some administrative unit using a theory of institutional goals and missions. In each of these examples the abstract structure, the "theory", *explains* some phenomenon by placing it in a larger context, describing important things about what has made the phenomenon to be the way it is. Usually an explaining theory is some sort of conceptual structure which presents in some fashion the "causes" of the things being explained. When a theory explains by describing causes and causal relationships, we may reasonably speak of "causal explanation". The senses of "explanation" important for this paper are all senses of "causal explanation".

When somebody "explains" something to somebody, giving a causal explanation of some physical event for example, the *explainer* conveys (more or less accurately) a theory, a conceptual structure, to the *explainee*. If all goes well, the explainee *understands* something about the event that he/she didn't understand before. We may say that *to understand an event* (in this particular sense) *is to grasp a causal explanation of it*[4] A causal explanation is a structure made up of linked concepts, including concepts of the event and of its supposed causal antecedents.

Explaining purposive or goal-directed behavior introduces a new dimension into this account of explanation. The existence within an agent of a goal which influences the behavior of the agent, makes the presence of that goal an important part of the causal ancestry of the behavior it influences. Thus to adequately explain goal-directed behavior we need to mention the goals that have actively influenced the behavior. Explanations that make reference to goals are usually referred to as *teleological explanations*. In a situation assessment system, the explanations given for an adversary's behavior are teleological explanations.

---

[3]Note how abduction was used to justify itself here!

[4]"We suppose ourselves to possess unqualified scientific knowledge of a thing, ... , when we think we know the cause on which the fact depends, as the cause of that fact and no other, and, further, that the fact could not be other than it is. .... What I now assert is that at all events we do know by demonstration. By demonstration I mean a syllogism productive of scientific knowledge, a syllogism, that is, the grasp of which is *eo ipso* such knowledge. Assuming then that my thesis as to the nature of scientific knowing is correct, the premisses of demonstrated knowledge must be true, primary, immediate, better known than and prior to the conclusion, which is further related to them as effect to cause." — Aristotle [2]

Intelligent agents, insofar as they ARE intelligent agents, that is insofar as they accomplish *smart thinking* as opposed to *stupid thinking*, do things for good reasons, whether they know it or not. What makes them smart is a sort of appropriateness of the thinking processes for the task at hand, and for the way the world is. When an intelligent agent makes a decision by *explicitly* considering reasons for making the decision one way or another, those reasons form a significant part of the causal ancestry of the decision. *For intelligence, reasons are causes.* Thus when we ask an intelligent system to explain its reasoning processes by giving reasons, we are once again asking for a form of causal explanation.

Design Considerations for Automated Situation Assessment

We strongly suggest that the system design be oriented towards producing a **"realistic appraisal"** of the situation, implying that the system should have the following characteristics.

- The system should keep track of where its interpretations are most certain, and where they are less so. It should have robust, common-sensical behavior in where it places the most certainty, and use these places as anchor points for further inference. The system shouldn't be "flighty" in its reasoning, uncritically engaging in long chains of inference without anchor points. In particular the system should be well grounded in the hardest evidence about the adversary's behavior; hard evidence should not be ignorable. The system should be strongly driven to explain the best attested behavior in some detail. (The best attested behavior, by the way, will not just include items at the smallest grainsize; for example sometimes we could be surer that there is an attack going on than about what the details are. Many different items of behavior, all attesting to the existence of a particular adversary plan, could in principle render that plan more certain than any single piece of evidence for it.) The system should seek these anchor points, infer a step or two beyond them into the realm of educated guessing, but go no further; some behavior should be left uninterpreted if necessary to avoid unwarranted speculation.

- The system should attribute only reasonably "plausible" plans to the adversary. This will require appraising the feasibility and utility of hypothesized plans, and also appraising the likelihood of these plans based on conformity to known characteristic ways of behaving.

- The system should lean neither towards optimism nor pessimism in its appraisal, but should aim instead towards a realistic balance of these tendencies. We believe that a basic system, structured to be realistic, will provide a firm foundation for pessimistic or optimistic variations on the basic problem solving that might be intentionally engaged in for special purposes. In particular it would be a big mistake to bias the main system towards pessimism in order to assist with "blunder avoidance" since this would introduce the new danger of blundering by overreacting to mere possibilities. A better way to do things would be to maintain, alongside the main realistic situation assessment, a worst-case estimate based primarily on capabilities.

- The system design should avoid making unrealistic idealizing assumptions about the adversary such as attributing to him perfect communications and coordination of actions, or perfect knowledge of environmental conditions or friendly force deployment, or the ability to always make flawless plans. In short, the system should not, in any simple way, suppose that the adversary is acting consistently.

- The problem solving architecture should support a system which strongly "tends towards the right answer". What this means is clear enough, but it is difficult to state what this implies for system architecture. It implies at least that knowledge structures should be redundant so that, if the system can't figure things out one way, there is a good chance that it will do it in another. It also implies that problem solving control strategies should have characteristics such that, as more and more information is available, in principle revealing the adversary's plans with greater and greater clarity, the system will produce hypotheses about those plans with greater and greater accuracy, and increasing confidence.

Some other desirable system characteristics:

- It should keep track of what the main alternatives are to its interpretations of events, so that interpretations can change rapidly if necessary, and so that events can be locally reinterpreted where this is indicated, in order to achieve as much as possible a unified and coherent interpretation of the situation.

- It should use control strategies with good computational characteristics to avoid the potentially very explosive combinatorics of hypothesis assembly and criticism. Computational strategies should be feasible in the sense that they should be efficient and scale up well. Criticism is potentially very expensive and needs to be used judiciously. "Jumping to conclusions" is not nearly as computationally expensive as being careful to systematically rule out alternative interpretations. An ideal system would be a good guesser to start with, and a judicious critic; able to use small amounts of criticism to good advantage, and more criticism, when computational resources are plentiful, to even greater advantage. It is *not feasible to explicitly generate all possible composite hypotheses.*
- It is important to be able to project the inferred plans forward in time, both for purposes of counterplanning and for monitoring the unfolding of events to confirm or revise hypotheses.
- It should use reasoning processes which are explainable. Insofar as its reasoning processes are well designed, and reasonably common-sensical, this should not present too much of a problem. It becomes a major problem only if its reasoning strategies are counterintuitive or incomprehensible. For example important conclusions shouldn't fall like magic out of unfollowable number manipulations.

Elements of the Design

Watching the Board: Actor-Centered Abducers

As we said earlier, output from the data fusion process is presented to the situation assessment system as the behavior of actors in the map region of interest. This activity can be thought of as being displayed on a map board which has symbols on it representing the actors, and packaging what we know about them. The map can be thought of as having variable resolution, including symbols for military units over a range of scales. For example army divisions, regiments, battalions, and companies might all be represented and linked hierarchically within one dynamic mapboard data structure.

In order to organize the problem solving, we propose assigning one *abducer* to watch each actor symbol on the board. The abducer's job is to track the actor's activities, and continually strive to explain its actions by composing hypotheses about its plans and objectives. This way the problem of explaining the totality of an adversary's behavior can be decomposed into the distinct subproblems of explaining the behavior of each agent. Besides being a useful way to decompose the overall abductive task into manageable and semi-independent subtasks, this decomposition allows for parallel processing of the input data stream to the system, which has clear advantages for speed of computation.

The watching abducers can be thought of as forming a hierarchy corresponding to our best estimate of the adversary's hierarchy of organization and command. Abducers communicate up, down, and, when useful, across their hierarchy in order to cooperate in forming a coherent overall account of the enemy's behavior. Abducer intercommunication is used to propagate the inferential leverage provided by high-confidence local conclusions wherever they may appear in the hierarchy.

In the activity of each watching abducer, merely finding a plausible plan that includes the observed actions is not sufficient, not even if it coheres well with everything else we know at that point. Beyond that, we need to know *correctly* and *realistically* what the adversary's plan is, so practically we have to know how sure we are that our judgment is correct, and for example that there is no other good explanation for the actions. Finding a plausible including plan isn't enough, we also have to subject the inference to criticism so we know what's sure and what isn't; what's the only plausible interpretation of events, and what's conjecture. Overall the system's task is to form a coherent theory of the actions of the adversary, coherent at all levels, and to evaluate the confidence status of that theory in whole, and in each of its parts.

The proposed design for each abducer is derived from work on abduction engines which has its origins in medical diagnosis. [11, 19, 12, 16] In brief, each abducer is to be a specialized means-ends problem solver whose goal is to explain the significant findings as well as possible by forming, criticizing, maintaining, and improving a compound explanatory hypothesis. Along the way it keeps careful account of just how good an explanation it has built, which parts are firm, and which are only guesses. This basic strategy for forming and criticizing composite hypotheses has already stood the test of a working implementation for a real-world task [20], and has been described in some detail elsewhere [12, 16]. In a companion paper to this one we show a way of producing concurrent realizations of these sorts of abduction engines. (Concurrent processing within each abducer, not just among them as described above.) The new domain can be expected to introduce new challenges for designing this sort of abduction engine, and many details remain to be worked out, but the basic approach has already been proved to be workable.

Classification Problem Solving to Control Plan Recognition

The abducers will need sources of explanatory hypothesis fragments to synthesize into local composite explanations. The organizing principle is that more general schemata are at the "root" or top of the hierarchy, and more detailed ones are below; the hierarchy is ordered by the specialization or type-subtype relation. A tank battalion, for example, is associated with certain prestored hierarchies of plan schemata appropriate just for tank battalions. The actor's watching abducer will use these hierarchies at run time as sources of plausible hypothesis fragments. See Figure 1 for a sketch of a hierarchy derived from a U.S. Army field manual [8].

Associated with each each node in the hierarchy is precompiled recognition knowledge that measures the confidence with which that node can form a hypothesis for the activity currently under consideration. This recognition knowledge is a place where can locate knowledge that will allow us to make a quick decision whether the hypothesis can be ruled out or confirmed. If the hypothesis can be neither ruled out nor confirmed by a quick check of the situation, more involved and expensive types of reasoning will be employed at the appropriate times. Each node is represented by a *classification specialist*, a problem solving agent with embedded knowledge for its particular classification task. When one of the classification hierarchies is activated hy a working abducer, the classification problem solving proceeds top-down following what we have called an *establish-refine* control regime. Each classification specialist either rules out its hypothesis, pruning the search tree at that level of generality, or establishes its hypothesis and passes activation and control along to its subspecialists in parallel.[5] A third possible action for a classification specialist is to suspend processing, based on intermediate levels of confidence, to be reawakened by the abducer if initially more promising hypotheses fail to work out. By using this form of control the hypothesis space can be quickly explored, and a small number of plausible hypotheses found which are worthy of further investigation. There already exists a tool, CSRL, for implementing this sort of classification problem solving [3].

This highly modular and controlled way of performing the necessary initial plan recognition is in marked contrast to other approaches to plan recognition that have been proposed [21, 13, 6].

Need for an Intelligent Map Overlay

In order for the compiled plan recognition knowledge to function within each classification specialist, the specialist will need to make pointed queries to the map board to determine the facts relevant to its decision. But the level of abstraction of the facts needed for recognition will probably not match well to the level of abstraction explicitly represented on the map board. For example in order to decide whether 'attack' is an appropriate high-level hypothesis, the recognizer might want to know whether the enemy is moving towards or away from friendly positions. This would probably not be a fact that is stored explicitly and in those terms on the map board, it will be necessary to infer it from lower-level descriptions. Thus we postulate an intelligent database overlay for the map to provide forms of inference that support presenting the data to the plan recognizers at a level of abstraction above that of the raw data. This is a form of the *data abstraction task* identified by Chandrasekaran and Mittal [4] and Clancey [7]. A number of forms of spatial reasoning will be necessary here, just which sorts will have to be determined empirically.

Planning Components to Instantiate Plan Schemata, Find Routes, Allocate Resources, and Determine Targets and Objectives

When a classification node matches and establishes its sponsored plan schema at a certain level of confidence, it will often be necessary to go into the represented hypothesis in significantly more detail. It's not enough to know that the actor is plausibly following a plan for a certain type of attack, we want to know where the attack is coming, what the targets are, and what are the likely avenues of approach. Once a plan schema has been found to be plausibly applicable to the case, it will be necessary to *instantiate* the plan schema in some detail to determine,

- whether the plan can in fact be carried out; and
- what the probable details are, so that important elements like target, etc. can be uncovered, and so that the unfolding of the plan can be projected into the future.

Thus the system must be able to take the adversary's point of view to choose targets, plan plausible routes, allocate resources and so on. It is likely that target/objective identification is a specialized need of the plan schema instantiation process,

[5] This is the third opportunity we have found for parallelism, the other two being parallel abduction on parallel input streams, and concurrent processing within each abduction engine. This time advantage is being taken of the parallelism which is natural to the hierarchical classification task. A fourth opportunity for parallelism is provided by parallel evaluation of precompiled recognition features for each plan fragment, but we will not discuss that here.

and will need it's own specialized problem solver with its own knowledge organization and problem-solving strategies. There might well be other specialized subtasks calling for specialized reasoning modules, but for now let us suppose that all such reasoning can be lumped together into one large planning module capable of taking a plan schema as input, and instantiating the schema for the concrete situation by choosing targets, etc. This planner will return a plan to the abducer (if possible) with information about the degree of feasibility of the plan and with details filled in as far as this can be done without combinatorially exploring a space of branching alternatives.

The abducer and the planner communicate through a shared language of plan fragment representation. From the abducer's point of view plan fragments are hypotheses about behavior, to be constrained by what is plausible and achievable; while from the planner's they are proposed courses of action, to be filled in with details, while being constrained by the facts as best they are known.

Representing Plans, Goals, Behavior, States, and Intentions

The proposed language for representing plans, plan functions, behaviors by which plan objectives are achieved, intermediate states, and the roles of various actors in plans, is the Sembugamoorthy and Chandrasekaran Functional Representation Language. This has been described in [14] and has been elaborated for plan representation in [5].

The basic idea is that an overall plan (or device) is represented as an organized network of plan fragment frames. The main scaffolding of this network is a set of what/how frames, each frame associating a plan functionality (*what* is achieved) with the behavior of a particular agent (the *how*). Agent behavior is represented as a directed network of plan states, each state transition link representing a plan step, i.e. something that needs to be accomplished in order for the overall behavior to proceed as planned. Each of the state-state links is annotated by being frame-frame linked to specification of *how* that particular state change is supposed to be accomplished. For example a change from plan state 'target unscathed' to state 'target softened up' might be annotated with a link to a certain artillery subunit frame, thus specifying the agent that will be responsible for accomplishing the transition, and packaging how it will be done.

Besides frames describing various types of agents and their behaviors, there should be frames for packaging detailed sub-behaviors including maneuvers and routes, and for packaging knowledge of how other state transitions occur (for example night falls naturally without action being required.) Besides a state link annotation of how the transition will be accomplished, a link should be annotated with information about how long it will take, and with the rationale for the particular step's presence in the behavior (for example to establish a precondition for a later step.)

Figure 2 shows a partially ordered network of plan states with links annotated to represent some of the types of frames that can be given responsibility for various state transitions.

Summarizing the problem solving process

- Something changes on the map board, causing the appropriate abducer to wake up and try to accommodate the new information into its understanding of what the actor is doing.
- This abducer checks whether this new activity is already anticipated by its current hypothesis. If so, then this hypothesis can be updated in detail, and revised as to confidence level. If not, then the hypothesis will have to be reconsidered, and perhaps a wholly new hypothesis formed.
- Unanticipated activity represents something that needs to be explained, and appropriate sources of hypotheses for the type of agent and activity are consulted to find plausible explanatory hypotheses.
- Plausible hypotheses are explored by the planner to see if details can be successfully worked out, and to see of they will succeed in explaining the activity.
- If they are useful for explaining things, successful hypotheses will be pursued by moving lower in the plan classification hierarchy, and invoking the planner on viable alternatives to try to fill in details. On each invocation the planner will only fill in details as far as it can go without significant branching of alternatives, announcing disjunctive alternatives and stopping, rather than pursuing disjunctions within disjunctions.
- The more detailed plausible hypotheses determined in this manner, generated from prestored schemata in consultation with the planner, become resources available to the abducer for inclusion in a best composite hypothesis for the activities of the particular actor.
- The abducer forms its best explanation using the plausible plan fragments, and taking account of the coherence constraints and suggestive information made available by other abducers above, below, and lateral in the abducer hierarchy.

- Overall this process of forming local best explanations in all of the active abducers in parallel should, hopefully, settle down before very long, because each abducer quits when it has done the best it can locally, and because the action of all of those local abducers, each trying to form a local best explanation of what is going on, and each taking account as best it can of the hypotheses formed by its conceptual neighbors, collectively tends towards producing the best possible answer globally.

  If our representations are expressive enough, and our hypothesis-improvement control is aptly wrought, then the best answer globally, towards which the system tends, will be the "right answer", the actual plans of the other guy. Thrashing about of the system thus has two limits. From the outside the limit is the way the world actually is, whatever is really happening out there, which the system is designed to infer, and towards which it hopefully tends. From the inside the limit is imposed by clever strategies by which the problem solver quits trying to improve the hypothesis after it has done the best it can with the information available, and after a modest commitment of computational resources.

- Overall, the process is one where islands of higher certainty are established, and a wave of probable reasoning drives outward from them to see what else can be plausibly inferred.

Acknowledgments

This research is supported by the Defense Advanced Research Projects Agency under RADC Contract F30602-85-C-0010.

References

[1] D. Allemang, M. C. Tanner, T. Bylander and J. R. Josephson.
On the Computational Complexity of Hypothesis Assembly.
In *Proc. Tenth International Joint Conference on Artificial Intelligence.* Milan, August, 1987.
to appear.

[2] Aristotle.
Posterior Analytics.
The Basic Works of Aristotle.
Random House, New York, 1941, pages 110-186.
Translated by G.R.G. Mure.

[3] T. Bylander and S. Mittal.
CSRL: A Language for Classificatory Problem Solving and Uncertainty Handling.
AI Magazine 7(3):66-77, August, 1986.
An earlier version, "CSRL: A Language for Expert Systems for Diagnosis" by T. Bylander, S. Mittal and B. Chandrasekaran, appeared in "Computers and Mathematics with Applications", Special Issue on Practical Artificial Intelligence Systems, Vol. 11, No. 5, pp. 449-456, 1985. An earlier version by the same title also appears in *Proc. of The International Joint Conference on Artificial Intelligence*, August, 1983, pp. 218-221.

[4] B. Chandrasekaran and S. Mittal.
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.

[5] B. Chandrasekaran, John Josephson and Anne Keuneke.
Functional Representations as a Basis for Generating Explanations.
In *Proceedings of the 1986 IEEE International Conference on Systems, Man and Cybernetics.* Atlanta, Gerogia, October 14-17, 1986.
Invited paper.

[6] Eugene Charniak and Drew Mc[illegible].
Introduction to Artificial Intelligence.
Addison Wesley, 1985.

[7] Clancey, William J.
From GUIDON to NEOMYCIN and HERACLES in Twenty Short Lessons.
AI Magazine 7(3):[illegible], August, 198[illegible].

[8] U.S. Department of the Army.
Field Manual 30-5, Combat Intelligence, Appendix J.
October 1973

[9] Gilbert Harman.
The Inference to the Best Explanation.
Philosophical Review LXXIV:88-95, January, 1965.

[10] John R. Josephson.
Explanation and Induction.
PhD thesis, The Ohio State University, 1982.

[11] J. R. Josephson, M. C. Tanner, J. W. Smith, M.D., J. Svirbely and P. Straum.
Red: Integrating Generic Tasks to Identify Red-Cell Antibodies.
In Kamal N. Karna (editor), *Proceedings of The Expert Systems in Government Symposium*, pages 524-531. IEEE Computer Society Press, 1985.

[12] J.R. Josephson, B. Chandrasekaran, J.W. Smith Jr., M.C. Tanner.
A Mechanism for Forming Composite Explanatory Hypotheses.
IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Causal and Strategic Aspects of Diagnostic Reasoning :pages unknown, To Appear, 1987.
Available as a Technical Report from Ohio State LAIR.

[13] Henry A. Kautz and James F. Allen.
Generalized Plan Recognition.
In *Proceedings of AAAI-86*, pages 32-37. AAAI, 1986.

[14] V. Sembugamoorthy and B. Chandrasekaran.
Functional Representation of Devices and Compilation of Diagnostic Problem Solving Systems.
In J. Kolodner and C. Reisbeck (editor), *Experience, Memory and Reasoning*, pages 47-73. Lawrence Erlbaum Associates, 1986.
An earlier version, "A Representation for the Functioning of Devices that Supports Compilation of Expert Problem Solving Structures," appears in the *Proceedings of MEDCOMP'83*, IEEE Computer Society, September, 1983.

[15] Pople, H.
On the Mechanization of Abductive Logic.
Proceedings of the Third International Joint Conference on Artificial Intelligence :147-152, 1973.

[16] W. Punch, M. Tanner and J. Josephson.
Design Considerations for Peirce, a High Level Language for Hypothesis Assembly.
In *Expert Systems In Government Symposium*, pages 279-281. IEEE Computer Society Press, October, 1986.

[17] Reggia, J.
Abductive Inference.
In Kamal N. Karna (editor), *Proceedings of The Expert Systems in Government Symposium*, pages 484-489. IEEE Computer Society Press, 1985.

[18] Simon, H.
The Sciences of the Artificial.
MIT Press, 1969.

[19] Smith, J.W.
RED: A CLASSIFICATORY AND ABDUCTIVE EXPERT SYSTEM .
PhD thesis, Ohio State University, August, 1985.

[20] J. W. Smith, M.D., J. R. Svirbely, C. A. Evans, P. Straum, J. R. Josephson and M. C. Tanner.
RED: A Red-Cell Antibody Identification Expert Module.
In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, pages 126-135. The Ohio State University, January 1985.

[21] Robert Wilensky.
Planning and Understanding.
Addison-Wesley, Reading, Massachusetts, 1983.

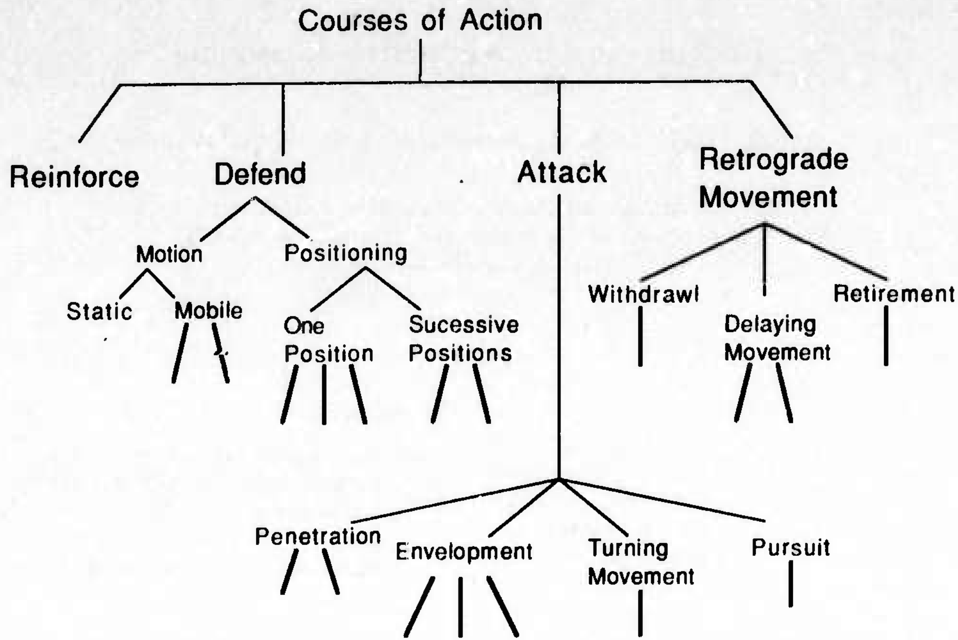


Figure 1: Plan Recognition Hierarchy

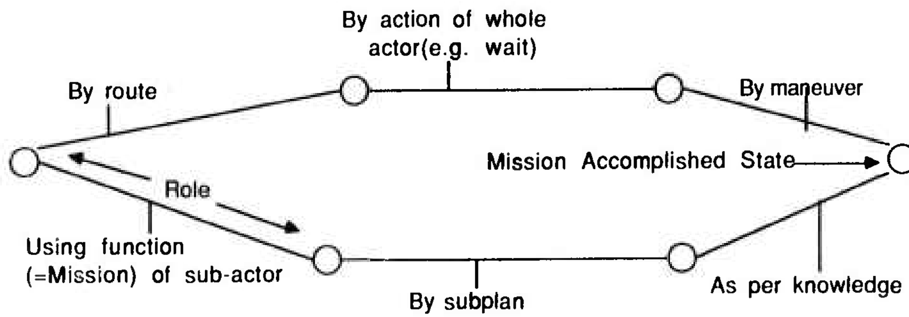


Figure 2: Representing Plan Understanding

Concurrency in Abductive Reasoning

Ashok Goel, John R. Josephson, and P. Sadayappan

Laboratory for Artificial Intelligence Research
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract

The information processing task in abductive reasoning is to infer a best explanation for a set of data. Some typical subtasks of this are generating hypotheses that can account for various portions of the data, and synthesizing a composite hypothesis that best explains the whole data set. In this paper we provide task-specific concurrent algorithms for some of the subtasks of abductive reasoning. In particular we present a blackboard architecture and a marker algorithm for the task of synthesizing a composite hypothesis.

1. Introduction

Abductive inference has received significant attention in research on knowledge-using reasoning, and construction of knowledge-based systems [Charniak and McDermott, 1985; Josephson *et al.*, 1987; Pople, 1977; Reggia, 1983]. The information processing task in abductive reasoning is to infer a hypothesis that best explains a set of data. A typical subtask is to generate hypotheses that account for different subsets of the data. Another subtask is to use these hypotheses in synthesizing a composite hypothesis that best explains the entire data set. However synthesis of a composite explanatory hypothesis in the presence of certain types of interactions between component hypotheses may be computationally very expensive [Allemang *et al.*, 1987]. This suggests exploiting concurrency for the construction of abductive inference making systems. Indeed with the increasing availability of concurrent machines, exploration and exploitation of concurrency in abductive reasoning is quite timely. Moreover analyzing the processing dependencies to determine where concurrent mechanisms apply, can be expected to increase our understanding of abductive problem solving generally.

We use the term "concurrency" here to imply non-serial processing which has characteristics of both parallel *and* distributed processing. In AI research concurrency is being explored at several different levels of organization:

- Neural architecture level: e.g. research on parallel and distributed processing in the connectionist paradigm. At this level, the grain size of concurrently executable processes is very small, and the parallelism between them is massive.
- Language architecture level: e.g. research on parallelism at the level of Lisp, or Prolog.
- Symbolic architecture level: e.g. research on parallelism at the level of production rules.
- Knowledge architecture level: functionally accurate, cooperative systems which use a blackboard architecture for control and communication in *distributed problem solving* [Lesser and Corkill, 1983], is an example of this kind of work. At this level, the grain size of concurrently executable processes is medium to large, and the parallelism between them is moderate to limited.

Our current research on concurrency in abductive reasoning is also at the knowledge architecture level of abstraction. Thus our analysis is in the language of functional specifications of problem solving tasks and subtasks, mechanisms for problem solving in the form of appropriate knowledge and control structures, and communication between cooperating problem solvers.

2. Abductive Reasoning

2.1. The Form of Abduction

Abduction is a form of logical inference that may be characterized as follows [Josephson *et al.*, 1987]:

D is a collection of data
(facts, observations, givens),
C explains D
(would, if true, explain D),
No other hypothesis explains D
as well as C does.

Therefore, C is (probably) correct.

Abductive inference appears to be ubiquitous in knowledge using reasoning processes. Abduction occurs in diagnostic problem solving, where the data is in the form of symptoms, and the explanatory hypotheses are diseases or component malfunctions. Data interpretation (as in DENDRAL) where the data is in the form of sensor readings, and the explanatory hypotheses are about object structures; and military situation assessment, where the data is in the form of events, and the explanatory hypotheses are plans ascribed to the adversary, are also instances of abductive reasoning. Some aspects of perception, and some aspects of natural language understanding, appear to be abductive in character as well.

2.2. Abductive Task and Subtasks

Our research on abduction takes place in the context of a theory of **generic tasks** in knowledge-using problem solving [Chandrasekaran, 1986]. A generic task corresponds to a primitive type of reasoning, associated with which are organizations of knowledge and controls of problem solving appropriate for it. **Classification** of a set of data describing a specific situation onto a precompiled taxonomy of hypotheses for instance is one generic task; another is the **abductive assembly** of a composite explanatory hypothesis using as components hypotheses that account for different subsets of the data. The organization of knowledge and control of problem solving appropriate for the classification task are different from that for the assembly task.

Generic tasks provide a high-level vocabulary for characterizing complex reasoning processes, and provide high-level building blocks for constructing integrated knowledge-using systems. Josephson *et al.* [Josephson *et al.* 1987] have shown that the abductive task can be decomposed under some circumstances into the generic tasks of hierarchical classification, and assembly/criticism of a best composite explanatory hypothesis.

2.3. Abductive Assembly Systems

Probably the best known knowledge-using system that performs a form of abductive assembly is the INTERNIST system for diagnosis in internal medicine [Miller *et al.*, 1982]. INTERNIST will continue to conjoin further diseases to a growing diagnostic conclusion until all of the important findings have been accounted for. The DENDRAL system [Buchanan *et al.*, 1969] performs abductive assembly in its task of elucidating molecular structure from mass spectrogram. DENDRAL assembles an explanatory hypothesis in the literal sense that it assembles a model of a molecule that represents an hypothesis about the parent molecule causing the various lines in the mass spectrographic data. This model molecule is assembled from fragments representing hypotheses about what is causing various spectral lines.

The RED system, an integrated expert system for identifying red-cell antibodies for use in medical blood banks, explicitly uses a classification and assembly mechanism for performance of the abductive task. In RED a classification module systematically searches a space of precompiled hypotheses to find ones plausibly applicable to the case, determines the *prima facie* likelihoods of the plausible hypotheses that are found, and determines what each plausible hypothesis can explain of the data for the case. An assembly module considers the hypotheses with high likelihoods as candidates for inclusion into a composite explanatory hypothesis, and assembles the composite hypothesis that is a best explanation for the data set.

The MDX2 system, an integrated expert system for diagnosis of a class of diseases in internal medicine, also uses the classification and assembly mechanism [Sticklen, 1987]. MDX2 contains multiple classification modules that perform their duties in different areas of medicine. It also contains an assembly module that directs the activities of the classification modules, and assembles a composite explanatory hypothesis using component hypotheses from the different classification modules.

PEIRCE is a knowledge representation *language* or *tool* under development at Ohio State for constructing problem solving systems which assemble a composite explanatory hypothesis, using as components hypotheses that account for different subsets of the data.

3. Concurrent Classification

3.1. Hierarchical Classification

The RED and MDX2 systems use hierarchical classification to accomplish a part of the problem solving. In hierarchical classification precompiled hypotheses are organized into a taxonomic hierarchy [Gomez and Chandrasekaran, 1981]. Associated with each hypothesis is a specialist for that hypothesis, a problem solving agent with embedded knowledge for confidence evaluation of its hypothesis. Starting with the specialist at the top of the hierarchy, each specialist, when invoked, matches its hypothesis with a subset of data relevant to the hypothesis, and computes a likelihood value for the hypothesis depending on the quality of the match. If the likelihood value for the hypothesis is high enough,

then the hypothesis is said to be "established", else it is "rejected". If the hypothesis is established, then the specialist attempts to refine its hypothesis by sending messages invoking each of its subspecialists, which then repeat the process. If a hypothesis is rejected, then its subhypotheses are implicitly rejected as well, thus pruning the search. In this way, following the top-down prune-or-pursue control regime that has been called "establish-refine", the hypothesis space is efficiently searched.

Thus, a community of specialists cooperate to perform the task of hierarchical classification. Since knowledge is *distributed* among the hierarchically organized specialists, and since the control of problem-solving is top-down, the refinement of established hypotheses can be done *in parallel*. An algorithm for concurrent classification is given in Figure 1.

```
Match hypothesis with relevant subset of data
Compute likelihood value for hypothesis
If likelihood value is high
then
   Establish hypothesis
   if the specialist is a leaf specialist
   then
      STOP
   else
      Invoke all subspecialists
      STOP
   end if
else
   Reject hypothesis
   STOP
end if
```

Figure 1: Concurrent Classification

3.2. Multiple Classification

The MDX2 system contains several classification modules, each responsible for the classification task in its respective subarea of medicine. Multiple hierarchy classification is an instance of *distributed* problem-solving at two levels of organization: at the level of distinct classification modules, and at the level of distinct classification specialists within each module. We have just provided a concurrent algorithm for the second level. Clearly it is possible to run the different classification modules concurrently as well.

4. Concurrent Assembly

4.1. A Basic Serial Hypothesis Assembler

Let $D = \{d_i\}$, $i = 1,2,...,n$ be a set of data items, and let $H = \{h_j\}$, $j = 1,2,...,m$ be a set of hypotheses. We assume that each $h \in H$ is associated with information specifying which specific elements of D it can account for, and specifying the likelihood with which it can account for them. For the basic assembler we assume that the elements of H are mutually compatible, represent explanatory alternatives where their explanatory capabilities overlap, and otherwise do not interact with each other.

The task of the assembler is to build a best composite hypothesis C for explaining the elements of D, using the members of H as candidate parts. Note that there is no *a priori* guarantee that a unique best explanation exists. The assembler builds the composite hypothesis C using a specialized means-ends machine whose goal is a complete explanation. The assembler detects differences between the goal state (all of D has been explained), and the present state (some d_U has not been explained). It then selects an h from H which can explain the unexplained d_U, and integrates this h into the growing composite hypothesis C. Since, as we have postulated, the h_i are non-interacting, "integrating" h into C just amounts to logically conjoining it with what is already there.

There are three things that can happen when trying to explain some d_U:

1. There may be no $h \in H$ that can account for it. Then d_U is *unexplainable.*

2. There may be only one $h \in H$ that can account for it. Then this h is *essential.*

3. There may be more than one $h \in H$ that can account for d_U. Then the h accounting for d_U which has the highest likelihood value should be selected for integration into C. If the likelihood values for two or more h are the same, then selecting between them is based on some measure of overall explanatory power, or if that will not break the tie, then selection is made at random.

4.2. A Concurrent Hypothesis Assembler

4.2.1. An Architecture for Concurrent Assembly

There are two types of questions that are raised during the basic assembly. The first type is from the viewpoint of each $d \in D$, and is of the form "Which hypothesis in H can best explain me?". For concurrent assembly this type of question can be asked and answered for each element of D independently of others, and *in parallel.* The second type of question is from the viewpoint of each $h \in H$, and is of the form "Which elements of D should I be used to explain?". Again, this type of question can be asked and answered for each $h \in H$, independently of the others, and *in parallel.* Let **P** be a set of n processors corresponding to the d_i, and let **Q** be a set of m processors corresponding to the h_j. Then d_i, $i = 1,2,...,n$ and h_j, $j = 1,2,...,m$ may individually reside on the $n + m$ processors. We will also need an additional processor **R** for collecting results. We

assume that each of the $n+m+1$ processors is equipped with a local memory, and performs its task using only local resources unless otherwise noted.

The communication between the processors, and the control of problem solving can be achieved by using a *blackboard* architecture. In this particular framework for concurrent assembly the blackboard is used only as a shared data structure on a shared memory. The blackboard may be divided into two sides, a data side, and a hypothesis side. The blackboard contains the state of the problem-solving at any given time, initially containing d_i, $i=1,2,...,n$ on the data side, and h_j, $j=1,2,...,m$ on the hypothesis side. The hypothesis side also contains, for each $h \in H$, a list of the specific elements of D for which it can account, and the likelihood value with which it can account for them. The initial information on the blackboard may be posted by classifiers (concurrent or not), or by some other source of plausible hypotheses.

Each of the $n+m+1$ processors has access to both sides of the blackboard; each processor, when idle, is "looking" at the blackboard. A processor gets invoked when appropriate marks are placed on the blackboard; each processor, when invoked, performs its task and writes its results on the blackboard by placing appropriate marks. Finally, the hypotheses with appropriate marks are collected into C.

The semantics of the marks that may be placed on the blackboard are as follows:

- The mark of h_K on some $d_L \in D$ implies that the datum d_L is explainable by hypothesis h_K.
- The mark of **Explained** on some $d_L \in D$ implies that the datum d_L has been explained.
- The mark of **Unexplainable** on some $d_L \in D$ implies that the datum d_L is unexplainable using H.
- The mark of **Essential** on some $h_K \in H$ implies that the hypothesis h_K is in C and is essential to C (i.e. is indispensable, no complete explanation is possible without it.)
- The mark of **Best** on some $h_K \in H$ implies that the hypothesis h_K is in C and represents a most likely explanation for some data item.
- The mark of **In** on some $h_K \in H$ implies that the hypothesis h_K is in C.

1.2.2. An Algorithm for Concurrent Assembly

We present now present an algorithm for concurrent assembly.

1. Mark what each hypothesis can explain:

Q:

for each $h \in H$,
  for each $d \in D$,
    if h explains d,
    then mark d with the name of h.

2. Find unexplainable data, and essential hypotheses:

P:

for each $d \in D$,
  if d is not marked with any h,
    then mark d as **Unexplainable**,
    else if d is marked with only one
      $h \in H$,
        then mark h as **Essential**.

3. Find data items explained by essential hypotheses:

Q, using the subset of processors corresponding to $h \in H$ marked **Essential**:

for each h marked **Essential**,
  for each $d \in D$ such that
  h explains d,
    mark d as **Explained**.

4. Select additional best explanation hypotheses to cover more of the data:

P, using the subset of processors corresponding to $d \in D$ not marked **Explained** or **Unexplainable**:

for each d not marked **Explained**
or **Unexplainable**,
  if there is a *most* likely h,
  say h_d, among the hypotheses marked
  in step 1 as explaining d,
    then mark h_d as **Best**.

5. Find data items explained by best hypotheses:

Q, for the subset of processors corresponding to $h \in H$ marked **Best**:

for each h marked **Best**,
  for each $d \in D$ such that
  h explains d,
    mark d as **Explained**.

6. Select additional hypotheses by guessing to cover the rest of the data:

P, using the subset of processors corresponding to $d \in D$ not marked **Explained** or **Unexplainable**:

for each d not marked **Explained** or **Unexplainable**,
choose an h, say h_d,
among the hypotheses marked in step 1
as explaining d, and
mark h_d as **In**.

7. Collect results forming the composite hypothesis C:

R:

Collect $h \in H$ marked **Essential** or **Best** or **In** into C.

4.3. Criticism of Composite Hypothesis

Once the composite hypothesis C has been assembled, it may be tested for parsimony, and possibly improved, and further, component hypotheses may be tested for essentialness (some not previously counted as essential may yet gain that status), and C may be improved as a result of this too.

A composite hypothesis is *parsimonious* if it has no explanatorily superfluous parts. An hypothesis in C is explanatorily superfluous if removing it from C does not reduce the explanatory capability of C. After testing for parsimony, the explanatorily superfluous hypotheses are removed from C. Where it is not possible to simultaneously remove all of the explanatorily superfluous hypotheses without leaving some data unexplained, then an ordering can be made by appealing to their firmness in C, that is, **Best** hypotheses should be retained in favor of ones which are merely **In** (**Essential** hypotheses cannot turn out to be superfluous.) Similarly, where firmness in C does not resolve the choice, retention can be based on the initial determination of likelihood for each hypothesis as it comes from the hypothesis source.

Algorithmically, one way to accomplish this parsimony testing and improvement is to first test all non-essential hypotheses (in parallel) for superfluousness. If all of the superfluous hypotheses thus found can be simultaneously removed from C without leaving any datum unexplained, then well and good, they are all removed, leaving a parsimonious hypothesis. If they cannot all be removed, then just the **In** hypotheses that are superfluous are removed *en masse*, if that will not leave anything unexplained. If the superfluous **In** hypotheses cannot all be removed without leaving something unexplained, then they are removed one by one, ordered by initial likelihood, and starting with the least likely, retaining only those which, at the point that they are considered, cannot be removed without destroying explanatory coverage. Once the process of removing superfluous **In** hypotheses is complete, the **Best** hypotheses are again tested in parallel for superfluousness (superfluous ones may have become non-superfluous in the context of a C improved by removing superfluous **In**'s.) Once again superfluous **Best**'s are removed, either *en masse*, if that will not leave anything unexplained, or one by one using initial likelihood as the ordering principle. The result of this process will be to improve C to the point that it is completely parsimonious.

If there are incompatibilities between hypotheses (see below), a data item may initially (in step 2 of the assembly algorithm described above) appear to have several potentially explaining hypotheses, yet only one of these explanations is really viable. This can occur for example if all but one of the potential explainers is inconsistent with some other hypothesis which is **Essential**. Thus hypotheses in C not marked **Essential** are tested to see if indeed complete and consistent composite explanatory hypotheses can be built without using them. If not, then such ones also deserve to be counted as **Essential**, and allowing for the possibility that leveraging early islands of certainty (the essentials) might make a difference for the final composition of C, the algorithm described above should be restarted after step 2 with the newly discovered essentials now marked. (Criticism for parsimony will need to be repeated, but criticism for essentialness will not. Essentialness of an hypothesis is a global property of the setup, and is independent of the composition of C; thus it will not change on a subsequent assembly of C.)

4.4. Interacting Hypotheses

Several distinct types of interaction are possible between two hypotheses h_1, $h_2 \in H$.

1. h_1 and h_2 are mutually compatible, and represent explanatory alternatives where their explanatory capabilities overlap. This was our assumption for the basic assembler.

2. h_2 is a subhypothesis of h_1. This can happen if the source of hypotheses is a hierarchical classifier as in Red and MDX2.

3. The inclusion of h_1 in C suggests the inclusion of h_2. Such an interaction may arise if the assembler has knowledge of a statistical association between h_1 and h_2.

4. h_1 and h_2 cooperate additively where their explanatory capabilities overlap. This happens in Red's domain.

5. h_1 and h_2 are mutually incompatible.

6. h_1 and h_2 cancel the explanatory capabilities of each other in relation to some $d \in D$. For example h_1 (being true) might imply that some data value will increase, while h_2 implies that the value will decrease.

For each of the above interactions we have developed marker algorithms using the blackboard architecture. However, the algorithms that we have developed for incomptability and cancellation interactions work only for disjoint, pair-wise interactions. The problem with more involved forms of interaction arises because knowledge in our architecture is *distributed* among the $n+m+1$ processors, and each processor performs its task *locally*, while these interactions in general require *global* computation. One way to accommodate the incompatibility and cancellation interactions between hypotheses may be to augment the blackboard architecture with use of *critics* that have a global view of the state of problem-solving as it appears on the blackboard.

4.5. Hierarchical Assembly

In many domains it is possible to form groups of strongly interrelated data in the data set, for instance the groups of data corresponding to different classification modules in MDX2. In such domains it is possible to build and use a *hierarchy* of several small assemblers, rather than one large, flat assembler. Let us consider a two-level hierarchy of assemblers. At the lower level in the hierarchy, assemblers corresponding to different data groupings form composite hypotheses to account for the data types for which they are specialized, and the top assembler forms a composite from the sub-composites. Notice that it is possible for a datum to appear in more than one data grouping, and similarly, it is possible for a component hypothesis to appear in more than one composite hypothesis. Weak interactions at the lower level are reconciled at the highest level of the hierarchy, where all of the data needs to be explained.

This scheme is generalizable to a hierarchy of assemblers with a finite number of levels. Problem decomposition activations flow downwards through the hierarchy and composite hypotheses flow upwards. An approach similar to this is being explored in the construction of PEIRCE. The idea is to use a hierarchical organization of knowledge to perform a computationally complex task efficiently. Problem solving knowledge is *distributed* over the assemblers in the hierarchy, who cooperatively perform the overall assembly task. Notice that the approach to concurrent assembly described earlier complements this hierarchical assembly. Each assembler can perform its task using the blackboard architecture and marker algorithm that we have provided. Further, it is possible to exploit parallelism between the sibling assemblers at the same level in the hierarchy.

5. Conclusions

We have found concurrency in abductive reasoning in different forms and at different levels as summarized below:

- The classification task may viewed as an instance of distributed problem solving. Classification may be performed by a community of specialists organized in a taxonomic hierarchy. We have provided a concurrent algorithm for the classification task.

- At a higher level of organization, classificatory problem solving may be distributed among multiple classification modules. It is possible to exploit concurrency at this level as well.

- The assembly task may be performed concurrently. Data items and component hypotheses can reside individually on separate processors. The data processors, and separately the hypotheses processors, can execute in parallel, accomplishing the assembly task in several distinct waves of processing as described above.

- It is possible to perform the task of assembly by a hierarchical organization of assemblers, where the processing at each level goes on in parallel.

There are several promising lines of research from here:

- The blackboard architecture can be augmented using critics with a global view of the state of problem solving in order to accommodate incompatibility and cancellation interactions between hypotheses.

- A complexity analysis of the concurrent algorithms can be conducted.

- The concurrent algorithms can be simulated, and their efficiencies tested empirically.

- Constraints and opportunities imposed by available multiprocessor architectures can be investigated.

- A hierarchical assembler can be build.

Acknowledgments

We are deeply grateful to B. Chandrasekaran for providing the overall conceptual framework for this style of functional-architecture AI, and for providing the opportunity to pursue this particular line of research. We are grateful to Tom Bylander for the idea of using a marker algorithm for the assembly task. This research was supported by research grants from the Defense Advanced Research Projects Agency and the Air Force Systems Command, Rome Air Development Center (RADC contract F30602-85-C-0010), and the U.S. Air Force Office of Scientific Research (AFOSR-87-719026).

References

[Allemang *et al.*, 1987] Dean Allemang, Michael Tanner, Tom Bylander, and John Josephson. "On the Computational Complexity of Hypothesis Assembly", Technical Report, Laboratory of Artificial Intelligence Research, Department of Computer and Information Science, The Ohio State University, January 87. To be presented at *IJCAI-87*.

[Buchanan *et al.*, 1969] Bruce Buchanan, G. Sutherland, and Edward Feigenbaum. "Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry". *Machine Intelligence 4*, B. Meltzer and D. Mitchie, editors, Edinburgh University Press, Edinburgh, 1969.

[Chandrasekaran, 1986] B. Chandrasekaran. "Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design". *IEEE Expert*, 1(3):23-30, 1986.

[Charniak and McDermott] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence*, Addison-Wesley, Reading, Massachusetts, 1985.

[Gomez and Chandrasekaran, 1984] Fernando Gomez and B. Chandrasekaran. "Knowledge Organization and Distribution for Medical Diagnosis". *Readings in Medical Artificial Intelligence: The First Decade*, Chapter 13, pages 320-339, William Clancey and Edward Shortliffe, editors, Addison-Wesley, Reading, Massachusetts, 1984.

[Josephson *et al.*, 1987] John Josephson, B. Chandrasekaran, Jack Smith Jr., and Michael Tanner. "A Mechanism for Forming Composite Explanatory Hypotheses". Technical Report, Laboratory for Artificial Intelligence Research, Department of Computer and Information Science, The Ohio State University, 1987. To appear in *IEEE Transactions on Systems, Man, and Cybernetics*, Special Issue on Causal and Strategic Aspects of Diagnostic Reasoning)

[Lesser and Corkill, 1983] V. Lesser and D. Corkill. "The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem-Solving Networks". *AI Magazine* 4(3):15-33, Fall 1983.

[Miller *et al.*, 1982] R. Miller, J. Pople Jr., and J. Myers. "INTERNIST-1, An Experimental Computer-Based Diagnostic Consultant for General Internal Medicine". *New England Journal of Medicine*, 307:468-476, 1982.

[Pople 1977] Harry Pople, "The Formation of Composite Hypothesis in Diagnostic Problem-Solving: An Exercise in Synthetic Reasoning". In *Proceedings of the fifth International Joint Conference on Artificial Intelligence, IJCAI-77*, pages 1030-1037, 1977.

[Punch *et al.*, 1986] William Punch III, Michael Tanner, and John Josephson. "Design Considerations for PEIRCE, a High-level Language for Hypothesis Assembly". In *Proceedings of Expert Systems in Government Symposium*, pages 279-281, IEEE Computer Society, October 1986.

[Reggia, 1983] James Reggia. "Diagnostic Expert Systems Based on a Set Covering Model". *International Journal of Man-machine Studies*, 19:437-460, November 1983.

[Smith *et al.*, 1985] Jack Smith Jr., John Svirbely, Charles Evans, Pat Strohm, John Josephson, and Michael Tanner. "RED: A Red-Cell Antibody Identification Expert Module". *Journal of Medical Systems*, 9(3):125-138, 1985.

[Sticklen, 1987] Jon Sticklen. "MDX2: An Integrated Medical Diagnostic System". Phd. Dissertation in preparation, Laboratory for Artificial Intelligence Research, Department of Computer and Information Science, The Ohio state University, 1987.

An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures*

Harold D. Brown, Eric Schoen, Bruce A. Delagi

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Worksystems Engineering Group
Digital Equipment Corporation
Maynard, Massachusetts 01754

Abstract

This report documents an experiment investigating the potential of a parallel computing architecture to enhance the performance of a knowledge-based signal understanding system. The experiment consisted of implementing and evaluating an application encoded in a parallel programming extension of Lisp and executing on a simulated multiprocessor system.

The choosen application for the experiment was a knowledge-based system for interpreting pre-processed, passively acquired radar emissions from aircraft. The application was implemented in an experimental concurrent, asynchronous object oriented framework. This framework, in turn, relied on the services provided by the underlying hardware system. The hardware system for the experiment was a simulation of various sized grids of processors with inter-processor communication via message-passing.

The experiment investigated the effects of various high-level control strategies on the quality of the problem solution, the speedup of the overall system performance as a function of the number of processors in the grid, and some of the issues in implementing and debugging a knowledge-based system on a message-passing multiprocessor system.

In this report we describe the software and (simulated) hardware components of the experiment and present the qualitative and quantitative experimental results.

1. Introduction

This report documents an experiment investigating the potential of a parallel computing architecture to enhance the performance of a knowledge-based signal understanding system. This experiment was done within the Expert Systems on Multiprocessor Architectures Project of Stanford University's Knowledge Systems Laboratory.

The computational characteristics of complex knowledge-based systems are poorly understood, especially in parallel computational environments. Our Architectures Project is performing a number of experiments to try to gain some understanding of these characteristics and, in particular, of the potential for concurrent execution of such systems. A primary goal of the project is to develop software and hardware system architectures which exploit this concurrency to increase the performance of knowledge-based signal understanding and information fusion systems.

The Architectures Project is organized according to a hierarchy of computational abstraction levels as shown in Table 1-1. Each experiment represents a narrow, vertical slice through these levels and consists of a specific system choice for each level.

Table 1-1: Computational levels.

Level	Research questions
Application	Where is the potential concurrency in knowledge-based signal understanding tasks? How does the problem solver recognize and express application dependent concurrency?
Problem-solving framework	What are suitable framework constructs for organizing and encoding concurrent signal understanding tasks? What are appropriate granularities for knowledge, knowledge application and data to maximize concurrency? What types of strategies for control of knowledge application are needed to assure acceptable solution quality without introducing excessive execution serialization?
Knowledge representation and management	What kinds of knowledge representation mechanisms are suitable for exploiting concurrency in inference and search?
System programming language	How can general-purpose symbolic programming languages be extended to support concurrency and help manage the resource allocation and reclamation tasks on a distributed memory multiprocessor?
Hardware system architecture	What multiprocessc architectures best support the organization and concurrency in knowledge-based signal understanding applications?

For the reported experiment, the choosen application is a knowledge-based ELINT (ELectronics INTelligence) system for interpreting processed, passively acquired radar emissions from aircraft. The ELINT application is implemented in CAOS, an experimental concurrent, asynchronous object-oriented framework built on Zetalisp [1]. The CAOS framework, in turn, relies on the services provided by the underlying hardware system environment. For this experiment, the hardware system environment is a simulation of a parallel architecture, called CARE [2]. CARE simulates a communications grid of processing sites where each site

*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. Bruce Delagi is currently a visiting research scientist at Stanford from Digital Equipment Corporation. Computing resources were funded in part under NIH grant RR-00785 from the Division of Research Resources Biomedical Research Technology Program.

contains a Lisp evaluator, private memory, and a communications and process scheduling subsystem. Message-passing is the only means of inter-site communication. CARE is simulated using a general, event-based simulator, SIMPLE [3]. SIMPLE is written in Zetalisp and executes on a Symbolics 3600 or a Texas Instruments Explorer Lisp machine.[1] Figure 1-1 illustrates the relationship between the various software components of the experiment.

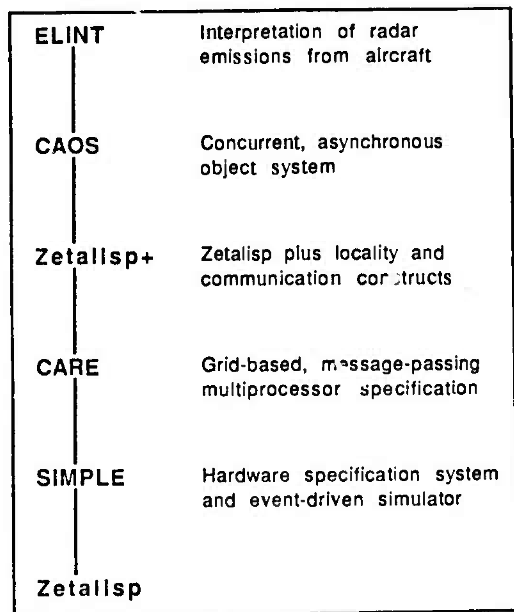


Figure 1-1: The software component hierarchy of the experiment.

The ELINT-CAOS-CARE experiment investigated both qualitative and quantitative aspects of the performance of the overall system. The CARE architecture uses dynamic, cut-through (as opposed to store and forward) routing through the communication grid for interprocessor message transmission. Message transmission time is indeterminate. As a consequence, without the imposition of significant message sequencing protocols (and the corresponding serialization of execution), operations are intrinsically non-deterministic in the sense that two executions of the same program on the same input data can result in different problem solutions depending on different message arrival orders. For many knowledge-based systems, in particular, the ELINT system, there is no such thing as *the* correct problem solution but only *satisficing* (i.e., acceptable) problem solutions. One primary objective of the experiment was to investigate the trade-offs between the imposition of various synchronizations (and the resulting loss of concurrency) and the quality of the problem solution. A second primary objective was the more usual investigation of the speedup of the overall system performance as a function of the number of processing sites in the CARE grid. A third objective was to gain some understanding of the difficulties in implementing and debugging a reasonably complex knowledge-based system on a multiple address space, message-passing multiprocessor system such as that represented by CARE.

In the following sections we describe, in decreasing hierarchical order, each component of the experiment. Section 2 describes the ELINT application. Section 3 gives an overview the CAOS programming framework and its approach to concurrency. ELINT's implementation in CAOS is described in Section 4, and Section 5 describes the salient features of the CARE architecture and its simulation environment. In Section 6 we present the results of the ELINT-CAOS-CARE experiment.

[1] A version of the SIMPLE simulator which runs on a local area network of multiple Lisp machines has also been implemented [4].

2. The ELINT Application

The driving application for our vertical slice experiment is a prototype, knowledge-based ELINT system for interpreting processed, passively acquired, real-time radar emissions from aircraft. This ELINT system is one component of a multi-sensor information fusion system, TRICERO [5] developed several years ago. ELINT was originally implemented in AGE [6], an expert system development tool based on the blackboard paradigm [7, 8]. ELINT is a relatively simple, but non-trivial, knowledge-based system. Much of its knowledge is implemented procedurally. However, if ELINT had been implemented as a production rule system, we estimate that its knowledge base would consist of about one thousand rules.[2]

ELINT's basic analysis technique is to correlate a large number of passively observed radar emissions into the smaller number of individual radar emitters producing those emissions. It then correlates the emitters into the yet smaller number of clusters of co-located emitters. ELINT maintains the track and activity histories of the clusters

2.1. ELINT's Inputs

The inputs to the ELINT system are multiple, time-ordered streams of processed observations from multiple collection sites. Each observation is presented in a record format. The fields of an input observation record are shown in Table 2-1.

Table 2-1: Elint observation record.

Field	Contents
Observation-Time	An integer time-tag indicating when the radar emission was sampled
Observation-Site	The symbolic name of the collection site acquiring the observation
Site-Location	The positional coordinates of the collection site at the time of observation
Emitter-Identifier	An integer identifing the radar emitter producing the emission
Line-of-Bearing	The line of bearing from the collection site to the observed emitter
Emitter-Type	A symbolic radar emitter type designator
Emitter-Mode	The operational mode of the emitter at the time of observation
Signal-Quality	A symbolic indicator of the signal quality of the observed emission

The **Site-Location** field is necessary since the collection sites can be mobile. The **Emitter-Identifier** is a unique integer identifier assigned by the collection sites to each distinct observed emitter. This identifier is used by the collection

[2] In general, there are currently no adequate metrics for measuring the complexity of knowledge-based systems. One crude measure used for rule-based systems is the number of rules. Although the number of rules does somewhat indicate the amount of knowledge, it does not give much indication of the complexity of the reasoning.

sites to indicate multiple observations of the same emitter both over time and from different collection sites. In particular, two concurrent observations of the same emitter from different collection sites should have the same identifier. Both the intra-site and inter-site determination of whether two observed emissions are from the same emitter are based on the electronic characteristics of the emissions and on signature analysis. This determination may be in error, and the ELINT system must cope with such identifier errors. The **Emitter-Type** of a radar emitter indicates the functional class of the emitter, for example, Air-Intercept (AI), Navigation (NAV) or Identification-Friend-Or-Foe (IFF), and, if known, the equipment type class of the emitter. Certain classes of emitter types can have multiple operational modes. The **Emitter-Mode**, if applicable, is emitter-type specific. For example, an AI radar can be either in Search Mode or Lock-on Mode depending on whether it is scanning for a target or whether it is automatically tracking a specific target. The **Signal-Quality** of an observation is a subjective, qualitative measure of the strength of the observed emission, for example, *strong*, *normal*, or *fading*.

All of the input information required for the ELINT system is obtainable from the raw radar signal data using current, passive radar signal collection and processing techniques. These techniques are largely automated and employ special-purpose hardware.

2.2. ELINT's Outputs

The primary outputs of the ELINT system are periodic status reports about the tracks and activities of clusters of emitters in the area under surveillance. A cluster is defined as a collection of emitters which are co-located over time. That is, two emitters are in the same cluster if for some given minimum number of consecutive time units (three in the current ELINT system) their corresponding time-tagged locational fixes are within a distance determined by the line-of-bearing resolution of the observation site equipment (one degree resolution in the current ELINT system). Conceptually, two emitters are in the same cluster if if they are on the same aircraft or are on two tactically associated and co-located (over time) aircraft, for example, a lead aircraft and his wingman.[3]

The periodic output reports contain, for each cluster, information about the cluster's current heading, position and track; an estimate of the number and types of aircraft in the cluster;[4] an indication of the cluster's current activity; and an indication if the cluster represents an immediate threat, for example, if it is within a certain proximity of a friendly aircraft, if its AI radar is in Lock-on Mode, or if its missile guidance radar is on.

2.3. ELINT's Processing Flow

The basic reasoning strategy used by the ELINT application is data-driven accumulation of evidence for the existence, the tracks, and the activities of emitters and clusters based on input observations and infered information. The primary processing flow is a kind of pipeline where the pipeline stages are observations, emitters and clusters.

Upon receipt of a new observation, the system first determines if the observed emission *matches* (i.e., has as a source) a known emitter (i.e., an emitter on ELINT's "situation board"). This match is based on the **Emitter-Identifier** assigner by the collection site to the observation, and it is verified using the emitter's characteristics and its track and heading histories. Depending on the outcome of the match, one of the following actions is taken:

1. If the observation does not match a known emitter, then a new emitter which is the source of the observed emission is hypothesized on the situation board and initialized from the information contained in the observation.

2. If the observation does match an emitter on the situation board and the match is verified, then the information contained in the observation is used to update the attributes of the matched emitter, including increasing the confidence level of the hypothesis that the emitter represents. Moreover, if the new observation is the second (or greater) observation of the emitter for the current time and it is from a different collection site than the previous observation(s) at that time, then a locational fix for the emitter is computed using the observed lines of bearing. If, in addition, the **Emitter-Type** and/or **Emitter-Mode** indicate a near-term threat to a friendly aircraft, then a threat report is output.

3. If the observation matches a known emitter but fails the match verification test, then an error in the **Emitter-Identifier** is indicated and the situation board is modified so as to undo any incorrect inferences based on the error. Also, an identifier error report is output to the collection sites.

On a periodic basis, the status of each emitter on the situation board is evaluated and various actions are taken:

1. If there have been no recent observations of the emitter, then the confidence level of the emitter is reduced. If, as a consequence of this reduction, that level falls below a given *no-confidence* threshold, then the emitter and all of the consequences infered from it (including cluster association) are deleted from the situation board.

2. If the confidence level is above a given *full-confidence* threshold and the emitter is not currently associated with a known cluster, then an attempt is made to *match* the emitter with a cluster on the situation board. This match is based on the track and heading histories and the type attributes of the emitter and the cluster. If a match is made, then the emitter is associated with the matched cluster and the emitter's current attributes are used to update the attributes of the cluster. If the match fails, then a new cluster is hypothesized on the situation board and the emitter is associated with it.

3. In the remaining case of a recently observed emitter with an associated cluster, the current attributes of the emitter are used to update the attributes of its associated cluster.

Also on a periodic basis, the state of each hypothesized cluster on the situation board is examined. If all of the emitters associated with the cluster have been deleted, then the cluster is deleted from the situation board. Otherwise:

---

[3] An aircraft can be operating with some (or all) of its radars off. In general, it is impossible to distinguish between, for example, two co-located aircraft, one with an AI radar on and one with a NAV radar on, and one aircraft with both its AI and NAV radars on. Hence, our ELINT system does its assessments based on emitter clusters rather than aircraft.

[4] Knowledge relating an aircraft type, for example F-15 or MIG-3, with the number and types of radars it carries is available. Using this knowledge and the identified emitter types in a cluster, it is possible to roughly estimate bounds on the number and types of aircraft in the cluster.

1. The cluster is checked to see if it should be *spl* into two (or more) clusters based on the curre locations of its associated emitters. If so, ne clusters with the appropriate associated emitters are hypothesized on the situation board.

2. The track history, heading history, speed history and activity history of the cluster are updated; and, if any new emitters have been recently associated with the cluster, an estimate of the types and numbers of aircraft comprising the cluster is derived.

3. A current status report for the cluster is output.

The ELINT processing flow lends itself naturally to concurrent execution. The parallel implementation of ELINT using CAOS is described in Section 4. The CAOS system itself is described in the following section.

3. The CAOS Programming Framework

CAOS is a framework which supports the encoding and the execution of multiprocessor expert systems. It represents an early attempt to bridge the gap between the application specification and the multiprocessor system programming primitives. The design of CAOS is predicated on the belief that many highly parallel architectures (e.g., hundreds of processors) will emphasize limited communication between processor-memory pairs rather than uniformly shared memory. We expect that such an architecture will favor relatively coarse-grained problem decomposition with little synchronization between processors. CAOS is intended for use in real-time, data interpretation applications such as continuous speech recognition and radar and sonar signal interpretation (see, for example, [9, 10]). CAOS is based on an object-oriented programming paradigm, and it draws many of its ideas from the Flavors system [1] and the Actors paradigm [11].

A CAOS application consists of a collection of communicating, active *agents*, each responding to a number of application-dependent, predeclared messages. An agent retains long-term local state. Each agent is a multi-process entity, that is, an arbitrary number of processes may be active at any one time in a single agent.[5] Conceptually, an agent can be thought of as virtual, multiprocess processor and memory pair. It responds to externally sent messages, and these message responses can alter the state of its local memory and can include the sending of messages to other agents.

CAOS is designed to express parallelism at a relatively coarse grain-size. For example, in the ELINT experiment, the message handlers (i.e., the *methods*) which implement the message responses are written as Lisp procedures, each averaging about one hundred lines of primitive Lisp code. CAOS supports no mechanism for finer-grained concurrency such as within the execution of agent processes, but neither does it rule it out. We could easily imagine message methods being written, for example, in QLisp [12], a concurrent dialect of CommonLisp which supports finer-grained concurrency.

3.1. CAOS' Approach to Concurrency

A CAOS application is structured to achieve high degrees of concurrency in the application execution 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 concurrently operating 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 the 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.

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 (i.e., machinery, processing hardware, knowledge, etc.). In an optimal pipeline of n processing elements, the throughput of the pipeline is n times the throughput of a single processing element in the pipeline.

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 almost no work at all. As a result, the n-element pipeline achieves less than an n-times increase in throughput. We discuss a partial remedy for this situation below.

3.1.2. Replication

Concurrency gained through replication is ideally orthogonal to concurrency gained through pipelining. Any size processing structure, from an individual processing element to an entire pipeline, is a candidate for replication. Consider a task which must be performed on the 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. Replication is more costly than pipelining, but it does avoid some of the problems associated with developing a pipelined decomposition of a task.

Our work 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

[5] The active processes in an agent are not scheduled preemptively. Instead, an executing agent process either runs to completion or until it is "blocked" awaiting some remote service (see Section 5).

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 conclusion rather than a composite conclusion. Any 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. Examples of this situation are shown in Section 4 where we describe the CAOS agent types for the ELINT application.

3.2. Programming in CAOS

CAOS is basically a package of operators on top of Lisp. These operators are partitioned into three major classes -- those which declare agent classes, those which initialize agents, and those which support communication between agents. We now describe briefly the CAOS operators for each of these classes. A more complete description of these operators is given in [13].

3.2.1. Declaration of Agents

Agents classes, like most object-oriented classes, are declared within an inheritance network. Each agent class inherits the attributes of its (multiple) parents. The root CAOS agent class, **vanilla-agent**, contains the minimal attributes required of a functional CAOS agent. All other CAOS agents have the **vanilla-agent** as a parent, either directly or indirectly. Another CAOS-declared agent class, **process-agenda-agent**, is a specialization of **vanilla-agent**, and includes a priority mechanism for scheduling the execution of messages. The **vanilla-agent** schedules its messages in a FIFO manner only.

Application agent classes are declared by augmenting the following primary attributes of CAOS-declared or other ancestral agent classes:

Local-Variables: An instance agent's local variables store its private state. The agent's message handlers may refer freely to only those variables declared locally within the agent. Each local variable may be declared with an initial value.

Messages-Methods: The only messages to which an agent may respond are those declared in the agent's class declaration. Associated with each declared message name is the name of the message's *method* (i.e., the message's message handler). In CAOS, a method name must refer to a defined Lisp procedure. This declaration simplifies the task of a resource allocator which must load application code onto each CARE site.

Clocks-Methods: An agent may periodically invoke actions based on internal clock "ticks." For example, the periodic update of emitter agents and the periodic output of cluster status reports are invoked by clock ticks. A clock is defined by its tick interval. Whenever an internal agent clock ticks, the set of methods associated with that clock are scheduled for execution.

Critical-Methods: This attribute declares certain sets of methods as being mutually "critical regions" for their owning agents.[6] Each such set of critical methods has an associated *lock*. Before an owning agent agent executes a critical method, this lock is checked. If it is unlocked, the agent locks it and executes the method. Upon completion of the method, the agent unlocks the lock. If the lock is locked, the method is queued in a FIFO queue awaiting the unlocking of the lock.

[6] A design goal for ELINT in CAOS was to avoid the use of critical methods, and our ELINT implementation does not use any. The CAOS initialization routines, however, do use such methods.

There are a number of additional basic agent attributes. However, most of these are used only internally by CAOS.

3.2.2. Initialization of agents

An initial CAOS configuration is specified by a two-component initialization form. The first component of the form creates the *static* agent instances. Some agent instances are created during system initialization and exist throughout a CAOS run. Such agent instances are called static agents as opposed to *dynamic* agents which are created (and possibly deleted) during program execution. For programmer convenience, we allow code in agent message handlers and default values of local-variables to reference such static agents by name. Before an agent instance begins running, each symbolic reference to the declared static agents is resolved by the CAOS runtimes.

The second component of the form is a list of expressions to be evaluated sequentially when CAOS's static agent instantiation 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. For example, in the ELINT application the initialization messages open log files and start the processing of ELINT observations.

Agent instances may also be created dynamically during execution. The creation operator accepts an agent class name and a location specification.[7] The *remote-address* of the newly-created agent instance is returned. The remote-address of an agent includes the CARE site coordinates where the agent resides and a pointer to the agent in the address space of that site. A dynamically created agent may *not* be referenced symbolically, however, its remote-address may be exchanged freely.

3.2.3. Communications Between Agents

Agents communicate with each other by exchanging messages. CAOS does not guarantee when messages reach their destinations. Due to excessive message traffic or processing element failure, messages may be delayed indefinitely during routing. It is the responsibility of the application program to detect and recover from such delayed messages.

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* which represents a "promise" for an eventual value.[8] Processes attempting to access the value of a future are blocked until that future has had its value set. Futures are first-class data types, and they may be manipulated by non-strict Lisp operators (e.g., list) even if they have not yet received a value. It is possible for the value of a CAOS 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.

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 a remote-address to the target agent (or its name in the case of a static agent), the message's routing priority, and the message's name and arguments. The sender continues executing while the message is delivered to the target agent.

[7] Currently, agents may be created only "at" or "near" specified CARE sites. CAOS makes no attempt at dynamic load balancing.

[8] Futures are also used in Multilisp [14]. The HEP Supercomputer [15] implemented a simple version of futures as a process synchronization mechanism.

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 it is immediately returned a local pointer to the future which will eventually receive a value from 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. This latter operator will *block* the process (i.e., suspend its execution and "swap it out") if the future has not yet received a value. When the future finally receives a value, the blocked process is rescheduled for resumed execution.

post-value: The **post-value** operator is similar to the **post-future** operator except that the sending process is immediately blocked until the target agent has returned a value. This operator is defined in terms of **post-future** and **value-future**, and it is provided for programming convenience.

It is possible to detect delay of value-desired messages by attaching a timeout to the associated future. The operators **post-clocked-future** and **post-clocked-value** are similar to their untimed counterparts but allow the caller to specify a *timeout-period* and *timeout-action* to be performed if the future is not set within the timeout-period. Typical timeout-actions include setting the future's value to a default value or resending the original message using the **repost** operator.

There also exist versions of the basic posting operators which allow the same message to be sent to multiple agents simultaneously. These versions exploit the multicast facilities of CARE (see Section 5).[9]

Multipost sends a side-effect message to a list of agents while **multipost-future** and **multipost-value** send value-desired messages to lists of agents. In the latter two cases, the associated future is actually a list of futures, and the future is not considered satisfied until all the target agents have responded. The value of such a message is an association-list where each entry in the list is composed of an agent's remote-address or name and the returned message value from that agent. There exist clocked versions of these operators (called, naturally, **multipost-clocked-future** and **multipost-clocked-value**) to aid in detecting delayed multicast messages.

3.3. 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 describe here only briefly the runtime structure of CAOS. This structure is described in greater detail in [13].

The implementation of CAOS described in this report is written in Zetalisp [1] and the primitive CARE operators using Zetalisp's object-oriented programming tool, Flavors[1].

Each CARE site contains a CAOS Site-Manager. A Site-Manager is realized as a Flavors instance. Its instance variables store site-global information needed by all agents located on the site. In addition, each Site-Manager includes CARE-level processes which perform the functions of creating new agents on its site and translating static agent symbolic names into agent addresses.

Each CAOS agent is also realized as a Flavors instance. A CAOS agent is a multiprocess entity. Most of the processes are created in the course of problem-solving activity. These processes are refered to as user processes. At runtime, however, there are always two special processes associated with each CAOS agent -- the ***agent input monitor process*** and the ***agent scheduler process***. The agent input monitor process watches the CARE stream by which the agent is known to other agents. It handles request messages and responses from value-desired messages from these agents. CAOS user processes are created in response to request messages from other agents or clocked methods. The agent scheduler process collaborates with the CARE site's operator processor in the scheduling of these user processes (see Section 5).

[9] Neither CAOS nor CARE currently support a "predicated multicast" mode wherein messages would be sent to all agents satisfying a particular predicate. Messages can only be multicast to a fully-specified list of agents. Receiving agents can, of course, apply arbitrary predicates to the message in order to determine their consequent action.

4. ELINT's Implementation in CAOS

We describe now the agent types and their organization for the ELINT application as implemented in the CAOS framework. This implementation illustrates some of the benefits and some of the drawbacks of the framework. As discussed in Section 2, ELINT is an expert system whose domain is the interpretation of passively-observed radar emissions. ELINT is meant to operate in real time. Emitters appear and disappear during the lifetime of an ELINT run. The primary flow of information in ELINT as implemented in CAOS is through a pipeline with replicated stages. Each stage in the pipeline is an agent. The basic ELINT agent pipeline is illustrated in Figure 4-1.

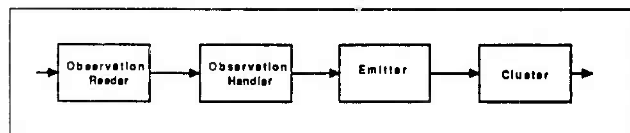


Figure 4-1: The basic ELINT agent processing pipeline.

4.1. ELINT Agent Types

The ELINT agent types described here are those used by the CT control strategy version of ELINT in CAOS (see Section 6).

Observation-Reader Agent

Observation-reader agents are an artifact of the simulated environment in which our ELINT implementation runs. Their purpose is to feed radar observations into the system. Observation-readers are driven off system clocks. At each clock "tick" (one ELINT time unit), they supply all observations for the associated time interval to the proper observation-handler agents. This behavior is similar to that of radar collection sites in an actual ELINT setting.

Observation-Handler Agent

The observation-handler agents accept radar observations from associated radar collection sites. Of course, in the simulated environment the observations actually come from observation-reader agents. There may be several observation-handlers associated with each collection site. The collection site chooses to which of its observation-handlers to pass an observation based on some scheduling criteria, for example, round-robin.

The contents of an ELINT observation was described in Section 2. In particular, each observation contains an identifier number assigned by the collection site to distinguish the source of the observation from other known sources. This source identifier is usually, but not always, correct. When an observation-handler receives an observation, it checks the observation's identifier to see if it already knows about the emitter which is the observation's source. 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 agent to create a new emitter agent and then passes the observation to that new agent.

Emitter-Manager Agent

There may be many emitter-manager agents in the system. An emitter-manager's task is to respond to requests from observation-handlers to create new emitter agents with associated source identifier numbers. If there is no such emitter agent in existence when the request is received, the manager will create one and return its remote-address to the requesting observation-handler agent. If there is such an emitter agent in existence when the request is received, the manager will simply return its remote-address to the requestor. This situation arises when one observation-handler requests an emitter that another observation-handler had previously requested. Emitter-managers must also handle the case of "almost concurrent" requests for the same emitter. This case occurs when a request is received for an emitter agent which is currently being created by another process on another CARE site in response to a slightly earlier request.

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 drawing a conclusion based on some evidence. ELINT must create its emitters in such a way that the individual observation-handlers do not each end up creating copies of the "same" emitter, that is, creating multiple emitter agents with the same associated source identifier (see Section 3.1.2). Consider the following strategies that the observation-handler agents could use to create new emitter agents:

1. The handlers could create the emitter agents themselves immediately as needed. Since the collection sites may pass observations with the same source identifier to any observation-handler, it is possible for multiple observation-handlers to each create its own copy of the same emitter. This strategy is not acceptable.

2. The handlers could create the emitter agents themselves, but inform the other handlers that they have done this. This scheme breaks down when two handlers try simultaneously (or almost 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 processing bottleneck.

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 source identifiers.

This last strategy is the one used used in our implementation of ELINT. The algorithm for choosing which emitter-manager to use is based on a many-to-one mapping of source identifiers to emitter-managers.[10]

[10] The algorithm simply computes the source identifier modulo the number of emitter-managers and maps that number to a particular manager.

Emitter Agent

Emitter agents hold the state and history of the observation sources they represent. As each new observation is received by an emitter agent, it is added to a list of new observations. On a periodic basis, this list of new observations is scanned for interesting information. In particular, after enough observations are received, the emitter may be able to determine the heading, speed, and location of the source it represents. The first time it is able to determine this information, it asks a cluster-manager agent to either match the emitter to an existing cluster agent (as described in section 2.3) or create a new cluster agent to hold the single emitter. Subsequently, it sends an update message to the cluster agent to which it is associated indicating its current heading, speed, and location.

Emitters maintain a qualitative confidence level of their own existence (*possible, probable, positive* and *was-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 by an emitter 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 associated of its deletion.

Cluster-Manager Agent

The cluster-manager agents play much the same role in the creation of cluster agents as the emitter-manager agents play in the creation of emitter agents. However, it is not possible to compute an invariant to be used for a many-to-one mapping between emitters and cluster managers. If ELINT were to employ multiple cluster-managers, any strategy for which of the many managers an emitter agent chooses to request a cluster match could still result in the creation of multiple instances of the "same" cluster (i.e., multiple cluster agents representing the same physical cluster of emitters). Thus, we have chosen to implement ELINT using only a single cluster-manager. Fortunately, new cluster creation is a relatively rare event, and the single cluster-manager has never been observed to be a processing bottleneck.

As described above, requests from emitters to associate themselves with clusters are specified as match requests over the extant clusters. Emitters are matched to clusters on the basis of their location, speed, and heading histories. 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 about the current state of those clusters. Thus, the cluster-manager asks all of its clusters to (concurrently) perform a match.

If none of the clusters responds with a positive match, the cluster-manager creates a new cluster for the emitter. If one cluster responds positively, the emitter is added to the cluster and it is so informed of this fact. If more than one cluster responds positively, this usually indicates that there is not yet sufficient resolution of the emitter's history to uniquely associate it with a cluster. In this case the emitter to cluster matching operation is tried again after more observations of the emitter have been processed.

Cluster Agent

The radar emissions from a cluster of emitters often indicate the activities of the aircraft represented by that cluster. For example, emissions from a missile guidance radar indicate that an air-to-air attack is imminent. Each cluster agent periodically applies heuristics about types of radar signals to try to determine the current activities of its represented aircraft, and, in particular, if these activities represent a threat to friendly aircraft. This activity information, the aircraft type information, and the merged track parameters of the emitters associated with each cluster are the primary outputs of the ELINT system. Also, each cluster periodically checks to see if all constituent emitters have been deleted. If so, it deletes itself.

Time-Manager Agent

Many of the knowledge-based actions taken by an ELINT agent make use of the agent's *last-observed* time, that is, the time stamp of the most recent observation associated directly or indirectly with the agent. For example, if an emitter agent determines that it has received no new associated observations for several data time intervals (i.e., that it is "out-of-date"), it will consider itself as no longer exisiting and it will delete itself and all of its relational links from ELINT's situation board.[11]

In an asynchronous message passing system such as CARE, it is difficult for an agent to determine whether it is out-of-date because it has not been observed recently or because messages to it which would result in an update of its last-observed time are delayed due to overall system load or local load imbalances. One solution to this problem would be for each observation-handler agent to send an "end-of-observation-time-interval" message to each of its known emitter agents whenever it observes the crossing of an observation time interval boundary.[12]

This solution was rejected for the reported implementation of ELINT because of a perceived excessive message overhead.[13] Instead, our ELINT experiment uses a time-manager agent. Whenever an observation-handler agent observes a new input observation time stamp, it reports this new time to the time-manager via a message. The time-manager maintains a conservative, global current observation time which is the minimum of the the reported time stamps. Whenever any agent considers taking a drastic, non-reversible action which is based on its being out-of-date (e.g., deleting itself), it requests a confirmation from the time-manager that its (the requesting agent's) last-observed time is sufficiently older than the time-manager's global current observation time. The requesting agent does not perform its considered action until it receives the confirmation. If in the interim, the requesting agent receives any messages which result in an update of its last-observed time, the confirmation is ignored.

Reporter Agent

Instances of the reporter agent class are used to asynchronously output various ELINT reports to displays and/or files, for example, threat reports and periodic situation board reports. In addition, instances of a specialization of the reporter class, **debug-trace-reporter**, are used during application program debugging to asynchronously output debugging traces in a manner that minimally impacts system timing dependencies.

4.2. ELINT Agent Organization

The ELINT agents are basically organized as a pipeline with replicated stages where each stage is an agent. Inter-pipeline dependencies and dependencies between replicated stages are managed by emitter-manager and cluster-manager agents. The amount of replication (i.e., the number of agents) at each pipeline stage is a function of that stage. For some stages, the number of replicated agents at that stage is fixed during system initialization. For example, the numbers of observation-handler agents, emitter-manager agents, and cluster-manager agents are pre-determined based on the number of collection sites and their output data rates. The numbers of emitter stages and cluster stages vary during the course of execution since the corresponding emitter agents and cluster agents are created and deleted as the radar emitters and collections of radar emitters which they represent appear and disappear over time.

The overall organization of the ELINT agents is illustrated in Figure 4-2.

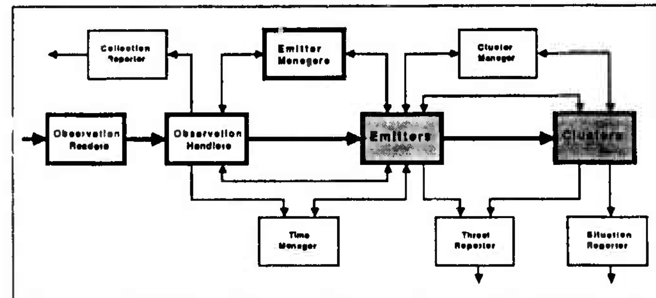


Figure 4-2: The overall ELINT agent communication organization.

5. An Overview of CARE

The CARE architectural specification and its simulation environment provide a parameterized and instrumented multiprocessor simulation testbed designed to aid research in alternative parallel architectures. The testbed executes within SIMPLE, a hierarchical, event-driven simulator [3].

A CARE architecture is a grid of tens to hundreds of processing sites interconnected via a dedicated communications network. The network uses dynamic, buffered, cut-through routing, and it supports multicast inter-site message transmission. The ELINT experiment, for example, was performed on various square CARE grids of hexagonally connected sites, that is, each site, excluding those at the edges of the grid, is connected to six of its eight nearest neighbors.

As shown in Figure 5-1, each CARE site consists of an *evaluator*, a general-purpose processor-memory pair; an *operator*, a dedicated communications and process scheduling processor which shares memory with the evaluator; and network interfaces -- *net-inputs* and *net-outputs* -- that accomplish pipelined message transmission, flow control, deadlock avoidance, and routing. Each net-input at a site may establish a connection with a net-output at any site, and all such connections at a site may be simultaneously active.

Application-level computations take place in the evaluator. The operator performs two duties. As a communications processor, it is responsible for initiating and receiving messages. As a scheduling processor, it queues application-level processes for execution in the evaluator. Message routing is performed by the net-input and net-output network interfaces.

In our simulation of CARE, the evaluator is treated as a "black box" Lisp processor. None of its internal operation is simulated. The Lisp machine hosting the simulation serves as the evaluator in each processing site. The operator, however, is functionally simulated, and the network interfaces are 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, the

[11] This action reflects the expectation knowledge that if an emitter within the area of observation is observed at time t, then it is expected that it will be observed at time $t+1$.

[12] Since each input observation stream is in observation-time sequential order, each observation-handler eventually knows when such a time boundary is crossed.

[13] This overhead may be more perceived than actual. A more recent implementation of ELINT uses such "end-of-observation-time-interval" messages. Initial results seem to indicate that the associated cost is not excessive (see [16]).

network routing algorithm, and the speeds of the process creating and switching mechanisms. 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. Alternative network topologies can be studied by using SIMPLE's graphic interfaces and composition operators to configure CARE components into any topology that can be wired.

The CARE simulation environment 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.

A more detailed description of CARE is given in [16], and the technology considerations underlying the CARE architecture are discussed in Appendix I.

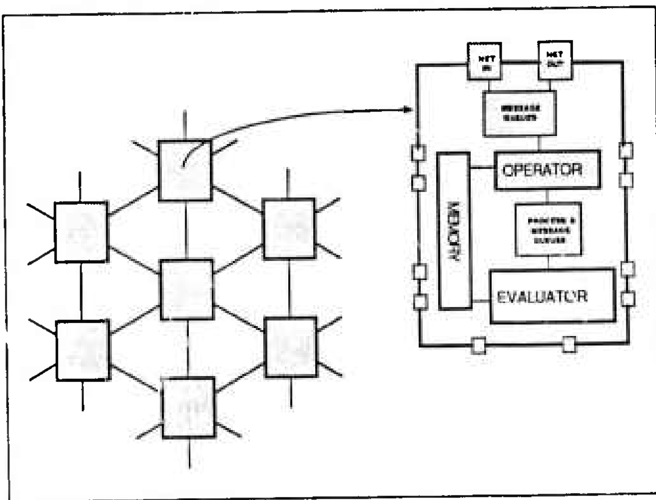


Figure 5-1: A hexagonally connected CARE grid.

6. Results and Conclusions

The CARE architectural simulation testbed and the CAOS system we have described have been fully implemented, and they are in use by several groups within our Architectures Project. CAOS-CARE executes on the Symbolics 3600 family of machines as well as on the Texas Instruments Explorer Lisp machine. ELINT, as described in Sections 2 and 4, has also been fully implemented, and we have analyzed its performance on various size CARE grids.

6.1. Evaluating CAOS

CAOS is a rather special-purpose environment, and it 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 dimensions of expressiveness, efficiency, and 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 should not 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 essentially programming in Lisp using objects 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 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 to 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. This conclusion has lead to an evolution of both CAOS and CARE which is described briefly in Section 6.3 and in detail in [16].

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. For example, does a 100-processor realization of a particular architecture perform ten times better than a 10-processor realization of the same architecture? Does it perform only five times better, only just as well, or does it perform even worse? In hardware systems, scalability is typically limited by various forms of contention in memories, busses, etc. The 100-processor system might be no faster than the 10-processor system because all interprocessor communications are routed through an element which is only fast enough to support ten 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 high data rates is by increasing the number of processors.

We believe CAOS scales well with respect to the number of available processors. The potential limiting factors to its scaling are increased software contention, such as the inter-pipeline bottlenecks described in Section 3, and 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 communications loading was not a problem in our ELINT-CAOS-CARE experiment.

Unfortunately, processor loading remains an issue. A configuration with poor load balancing in which some CARE sites 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 as agents are simply assigned to processing sites on a round-robin basis with no attempt to keep potentially busy agents apart. We currently have no solution to the problem of processor load balancing beyond that of carefully "hand crafting" a site allocation strategy for each application and then "tuning" that strategy via succesive refinement.

6.2. Evaluating ELINT Under CAOS

The input data set used for most of our ELINT-CAOS runs was based on a scenario involving 16 aircraft mounting a total of 88 radar emitters with between 4 and 45 emitters active and observed during any one data time interval. The scenario takes place in a 60 by 80 mile area over 36 time units, and it involves 1040 separate emitter observations.

Our experience with ELINT indicates that the primary determiner of throughput and solution 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 "control" strategies were used in our experiment:

1. NC: This "no control" strategy represents limited inter-agent control. Agents initiate actions independently. Whenever an agent wants to perform an action, it does so as soon as processing resources are available. For example, whenever an observation-handler agent needs a new emitter agent, it simply creates it with no attempt to coordinate this creation with other observation-handlers. As a result, multiple, non-communicating copies of an emitter may be created, and each copy receives a only portion of the input data it requires. The NC strategy was expected to produce qualitatively poor results, and it was primarilly intended only as a baseline against which to compare more realistic control strategies. What was surprising was that the strategy also produced quantitatively poor results (see below).

2. CC: In this strategy, agents cooperate in the creation of new agents via manager agents as described in Section 4. The manager agents assure that only one copy of an agent is created, irrespective of the number of simultaneous creation requests. All requestors are returned a reference to the single new agent. Originally, we believed the CC (for "creation control") strategy would be sufficient for ELINT to produce satisficing high-level interpretations. Our experiment results showed that this was not always the case (see below).

3. CT: The CT ("creation and time control") strategy was designed to additonally manage the skewed views of real-world time which develop in agent pipelines. For example, this strategy prevents an emitter agent from deleting itself when it has not received a new observation in a while even though some observation-handler agent has sent the emitter an observation which it has yet to receive. The agents corresponding to the CT strategy are those described in Section 4.

Table 6-1 illustrates the qualitative effects of the various control strategies and grid sizes. The table presents the six major performance attributes by which the quality of an ELINT run is measured. Since the input data for the ELINT experiment were generated from known scenarios, it was possible to compare the results of an ELINT run with "ground truth."

Table 6-1: ELINT Solution Quality Versus Control Strategies and Grid Sizes.

Qualitative performance attribute	Control strategy/grid size					
	NC/16	CC/16	CC/36	CT/4	CT/16	CT/36
False alarms	1%	0	0	0	0	0
Reincarnation	49%	42	2	0	0	0
Confidences	19%	20	90	89	93	95
Fixes	48%	42	99	100	100	100
Threats	65%	63	81	87	87	90
Fusion	0%	0	77	85	88	89

The major qualitative performance attributes are:

False Alarms: This attribute is the percentage of emitter agents that ELINT should not have hypothesized as existing with respect to the total number of emitter agents hypothesized.

ELINT was not severely impacted by false alarms in any of the control configurations in which it was run as the knowledge used for hypothesizing new emitters was quite conservative. That is, the knowlege was such that it prefered missing a true, but low confidence, emitter to creating a false alarm emitter.

Reincarnation: This attribute is the percentage of recreated emitter agents, that is, emitters which had previously existed but had erroneously deleted themselves due to lack of recent observations, with respect to the total number of emitters created. Large numbers of reincarnated emitters indicate some portion of ELINT is unable to keep up with the data rate. This can be caused by the data rate being too high globally so that all processing sites are overloaded or by the data rate being too high locally due to poor load balancing so that some subset of the processing sites are overloaded.

The CT control strategy was designed to prevent reincarnations. Hence, none occurred when CT was employed on any size grid. When the CC strategy was used, only the 36 site grid was large enough for ELINT to sufficently keep up with the input data rate so that emitters were not erroneously deleted due to overload.

Confidence Level: This attribute is the percentage of correctly deduced confidence levels for the existence of an emitter with respect to the total number of times such confidence levels were determined.

For each hypothesized emitter, ELINT maintains a dynamic confidence level for the existence of the emitter based on accumulating evidence (see Section 4.1). 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.

Fixes: This attribute is the percentage of correctly-calculated positional fixes of emitters with respect to the total number of times fixes could have been determined from the ground truth data.

A fix can be computed whenever an emitter has seen at least two observations from different collection sites in the same data time interval. If, for example, an emitter is undergoing reincarnation, it will not accumulate enough data to regularly compute fixes. Thus, the approaches which minimized reincarnation tended to maximize the correct calculation of fix information.

Threats: As described in Sections 2 and 4, certain emitter and

cluster events represent immediate threats. This attribute is the percentage of recognized threats with respect to the total number of threat events based on the ground truth data.

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.

The overall goal of the control strategy experiments was to see if it was possible to determine strategies where the quality of the output results were relatively insensitive to grid size and load balance but still achived significant concurrency.

We interpret from Table 6-1 that the 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 36 site grid. We believe the added complexity of the CT strategy, while never detrimental, is primarily beneficial when the interpretation system might be overloaded by high data rates or poor load balancing.

Table 6-2 gives the simulated execution times for the ELINT runs used to derive the data in Table 6-1, and Table 6-3 gives the total CAOS message counts for these runs.

Tables 6-2 and 6-3 clearly show that the processing cost of added control is far outweighed by the benefits in its use. Far less message traffic is generated, and the overall simulated time is reduced. Note that for the runs whose execution times are shown in Table 6-2, the input data rate was .1 seconds per ELINT time unit. Since the input data set used for these runs spanned 36 time units, the last observation was fed into the system at 3.6 (simulated) seconds. Hence, this is the minimum possible simulated execution time for these runs.

Table 6-2: Simulated ELINT execution times for various control strategies and grid sizes.

Control strategy	Grid size		
	4	16	36
NC		>11.19 sec.	
CC		10.87	5.12
CT	11.80	8.10	4.17

Table 6-3: CAOS message counts for ELINT executions with various control strategies and grid sizes.

Control strategy	Grid size		
	4	16	36
NC		>16118 msg.	
CC		7375	4823
CT	4516	4703	4616

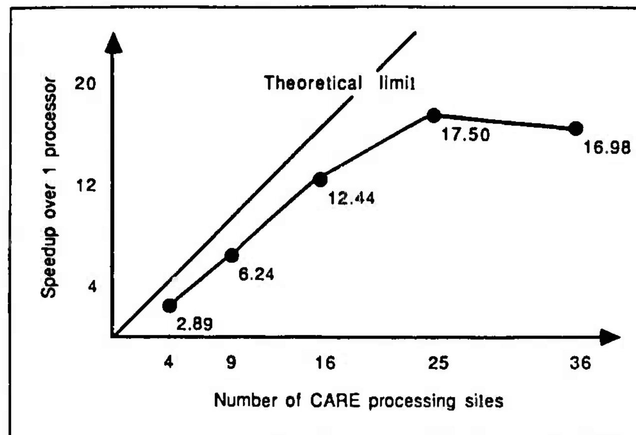
Table 6-4 and Figure 6-1 show the quantitative effect of processor grid size when the CT control strategy is employed. These results were produced with the input data rate set ten times higher (.01 seconds per ELINT time unit) than that used to produce Table 6-2. The minimum possible simulated execution time for the runs used to produce Table 6-4 is 0.36 seconds.

Table 6-4: Simulated ELINT execution time versus grid size for production runs using CT control strategy.

Grid size	Execution time
1	9.476 sec.
4	3.237
9	1.517
16	.761
25	.541
36	.557

As shown in Figure 6-1, the speedup achieved by increasing the processor grid size is nearly linear in the 1 to 25 processor site range. However, the 36 site grid was slightly slower than the 25 site grid.[14]

Figure 6-1: The relative speedup of ELINT executions on various size CARE grids.



In this last case, there was not sufficient data per ELINT time interval to warrant the additional processors. That is, there was not enough concurrency to exploit 36 processors. This can be seen from Table 6-5 which gives timing results for larger data sets with more emitters and observations during each time interval and, hence, more potential for concurrency.

[14]Because of the intrinsic non-determinism of a CARE architecture, we observed variations in the solution qualities and the run times between different runs of the same input data set on the same size CARE grids. For such runs, the variations in solution qualities never exceeded a fraction of a percent. However, the varitions in run times where as much as five percent. This accounts for the slightly longer execution time on 36 versus 25 processors.

Table 6-5: Simulated ELINT execution times and speedup for larger data sets.

Number of Observations	1-site grid execution time	36-site grid execution time	Speedup of 36 over 1
1040	9.476 sec.	.557 sec.	17.0
2080	25.10	.948	26.5
4160	55.87	2.259	24.7

As shown in this table, for an input data set representing twice as many emitters and observations than the basic data set, the 36 site grid achived a speedup factor of 26.5 (as opposed to a speedup of 17.0 for the basic data set) over a single processor. However, for a data set four times larger than the basic data set, the speedup factor was only 24.8. This was because this larger, and hence more concurrent, data set saturated the 36 site grid. That is, the 2080 observation data set already provided enough concurrency to fully exploit the 36 site grid.

6.3. Some Open 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 synchronize 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?
- Are there efficient mechanisms for dynamically balancing processor loads?

We have started to investigate these questions in the context of a new CARE environment. One of 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* which are computations with less state than those of processes.

When a context is forced to suspend to await a value from a remote service, it is aborted, and restarted from scratch later when the value is available. This behavior encourages more fine-grained decomposition of problems written in a functional style where 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 a complex and expensive scheduling strategy. Early indications are that the new CARE environment with a slightly modified CAOS environment performs around two orders of magnitude faster than the configuration described in this paper. The evolution of CARE and CAOS based on the results of our ELINT-CAOS-CARE experiment is described in greater detail in [16].

Acknowledgements

Our thanks to Russell Nakano, Sayuri Nishimura, James Rice and Nakul Saraiya who helped implement and maintain the CARE environment. Also, we wish to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Laboratory for their excellent support of our computing environment. We express special graditude to Edward Feigenbaum. His continued leadership and support of the Knowledge Systems Laboratory and the Architectures Project made it possible us to do the reported research.

APPENDIX

1. Technology Considerations Underlying the CARE Architecture

The CARE simulation testbed can be used to simulate shared memory as well as message passing multiprocessor architectures. For example, it has been configured to simulate a single address space, shared global memory architecture where the processors (and their local cache memories) are connected to the shared memory's controllers via a switching network. However, the intended focus of the CARE testbed is on message passing, multiprocessor architectures where each processor has significant local memory. This focus is based on technology considerations -- primarily communication versus processing costs.

The base for development of general purpose multiprocessor systems, as for computer systems generally, is given by the design constraints and opportunities established by evolving semiconductor design and manufacturing processes. The VLSI design medium brings a new perspective on cost -- switches are cheap while wires are expensive. Communication costs dominate those associated with logic. Communication is currently the resource in shortest supply, and it will become more of a constraint rather than less as semiconductor lithographies decrease.

The consequence of relatively expensive communication is that performance is enhanced if the design establishes that whenever a lot of information has to move in a short time, it does not have to move far. Significant locality of high bandwidth links is a design goal. Among the highest bandwidth links in a computer system are those connecting the processor and memory. Thus, close coupling of processors with local memory is preferred.

To reduce demand on the communications resource to supportable levels, local memory sizes for multiprocessors can be expected to increase to the 100K byte level and beyond, and block transfers between backing store and such several hundred kilobyte local memories will be used to make the most efficient use of both memory structures and communications facilities. Moreover, the functionallity of memory controlers will expand to include, for example, management of request queues, the dispatching of results, and execution of synchronization primitives; and thus, the distinctions between a memory controller and a small, simple processor will become blurred.

The proportion of area for a simple, high performance processor to the total area of a site with, for example, 256K bytes of local storage can be reasonably estimated at around 15%. From (i) this estimate of the incremental cost of adding a processor to a block of memory, (ii) the significant size of the total local storage in the system, (iii) the blurring of distinctions between fast, simple processors and memory controllers of increasing complexity, and (iv) the tendency towards block tranfers between local memory and backing store, it follows that the level of the storage hierarchy now labeled as "random access memory" is likely to be subsumed by a combination of large local memories and fast, block access backing stores in multiprocessor systems.

The performance of the available communication resource merits special attention in the design of multiprocessor systems. For example, dynamic routing which selects available inter-site links as needed is useful in balancing load, and thus it allows more of the communication resource of the system to be exploited throughout a computation. Cut-though routing which makes a routing decision on the fly as a packet is received reduces buffer requirements in the system and minimizes latency experienced in network transit. Flow control via signalling transmission delays back to the source based on local blockage information together with single "word" buffering and transmission validation at each network input and output port allows the source to complete a transmission in a time that does not depend on the size of the network. Point to point multicast which sends (approximately) the same packet to multiple targets using common resources to the largest degree possible can significantly enhance overall communication performance. A communication resource with these features provides a multiprocessor system with "virtual busses" that are established precisely as and when they are needed.

These technology considerations have led us to focus our attention on the class of multiprocessor hardware system architectures exemplified by CARE.

References

1. Weinreb, D. and Moon, D. (1981) Lisp machine manual, 4th ed. Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

2. Delagi, B., et al. (1986) CARE user's manual. Technical Report, Knowledge Systems Laboratory, Stanford University.

3. Saraiya, N. (1986) Simple user's manual. Technical Report, Knowledge Systems Laboratory, Stanford University.

4. Saraiya, N. (1986) AIDE: A distributed environment for design and simulation. Technical Report, Knowledge Systems Laboratory, Stanford University.

5. Williams, M., Brown, H. and Barnes, T. (1984) TRICERO design description. Technical Report ESL-NS539, ESL, Inc.

6. Aiello, N., Bock, C., Nii, H. P. and White, W. (1981) Joy of AGEing. Technical Report, Heuristic Programming Project, Stanford University.

7. Nii, H. P. (1986) Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. AI Magazine, vol. 7, no. 2, pages 38-53.

8. Nii, H. P. (1986) Blackboard systems part two: Blackboard application systems. AI Magazine, vol. 7, no. 3, pages 82-106.

9. Erman, L., Hayes-Roth, F., Lesser, V. and Reddy, D. R., (1980) The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. ACM Computing Surveys, vol. 12, pages 213-253.

10. Nii, H. P., Feigenbaum, E., Anton, J. and Rockmore. A. (1982) Signal-to-symbol transformation: HASP/SIAP case study. AI Magazine, vol. 3, no. 3, pages 23-35.

11. Lieberman, H. (1981) A preview of Act1. Artificial Intelligence Laboratory Memo 625., Massachusetts Institute of Technology.

12. Gabriel, R. and McCarthy, J. (1984) Queue-based multiprocessing Lisp. In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming. Austin, Texas.

13. Schoen,E. (1986) The CAOS system. Technical Report, Knowledge Systems Laboratory, Stanford University.

14. Halstead, R. H., Jr. (1984) MultiLisp: Lisp on a multiprocessor. In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming. Austin, Texas.

15. Denelcor, Inc. (1981) Heterogeneous element processor: Principles of operation. Boulder, Colorado.

16. Delagi, B., et al. (1986) Lamina: Streams and objects for concurrency. Technical Report, Knowledge Systems Laboratory, Stanford University.

An Instrumented Architectural Simulation System*

Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd

Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California 94305

Worksystems Engineering Group
Low End Systems and Technology
Digital Equipment Corporation
Maynard, Massachusetts 01754

Abstract

Simulation of systems at an architectural level can offer an effective way to study critical design choices if (1) the performance of the simulator is adequate to examine designs executing significant code bodies -- not just toy problems or small application fragments, (2) the details of the simulation include the critical details of the design, (3) the view of the design presented by the simulator instrumentation leads to useful insights on the problems with the design, and (4) there is enough flexibility in the simulation system so that the asking of unplanned questions is not suppressed by the weight of the mechanics involved in making changes either in the design or its measurement. A simulation system with these goals is described together with the approach to its implementation. Its application to the study of a particular class of multiprocessor hardware system architectures is illustrated.

1. INTRODUCTION

Simulation systems are quite often developed in the context of a particular problem. To a degree, this is true for SIMPLE, an event based simulation system, and CARE, the computer array emulator that runs on SIMPLE.[1] The problem motivating the development of both SIMPLE and CARE was the performance study of 100 to 1000-element multiprocessor systems executing a set of signal interpretation applications implemented as "1000 rule equivalent expert systems" [Brown86].

A set of constraints pertinent to this problem governed the design of SIMPLE/CARE. The applications represented significant bodies of code and so simulation run times were expected to be an important consideration. Moreover, the issues involved with the interactions of multiprocessor system elements were sufficiently unexplored prior to simulation that simplifications in the CARE system model, specifically with respect to element interactions, were suspect. This need for detail was, of course, in tension with the need for simulation performance. The ways that simulated system components would be composed into complete systems was initially difficult to bound. Further, it was clear that the models of these components would be elaborated over time and would undergo substantial change as design concepts evolved. It was also clear that the ways of examining the operation of these components would change independently (and at a great rate) as early experience indicated what alternative aspect of system operation *should* have been monitored in any given completed run.

The design goals that emerged then were (1) that the simulation system should support the management of substantial flexibility with regard to simulated system structure, function, and instrumentation and (2) that, in order to accomplish runs in acceptable elapsed times, the detail of simulation should be particularly focused on the communications, process scheduling, and context switching support facilities of the simulated system -- that is, on just those aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

1.1. Design Time Interaction And Run Time Operation

Encapsulation of the state of design components with the procedures that manipulate that state is one clear way to manage design evolution. Such encapsulation partitions the design along well defined boundaries. Components (by and large) interact with other components only through defined *ports*. Connections between components terminate at such ports. When a system simulation is initialized, connections are traced so that for every port, the simulator knows the connected (terminating) ports together with their containing components. Once such initialization is complete, that is, throughout the simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of connected ports of other components.

Partitioning issues of **system structure, component behavior,** and **instrumentation** into separate domains of consideration helps in managing a design that is both fluid and complex. **System structure,** that is, the relationship between components, can be specified through use of an interactive, graphics structure editor and is largely independent of component function per se. **Component behavior** is encapsulated in a set of definitions pertinent to the given class of component. Each component in a SIMPLE simulated system is a member of a class defined for that component type. **Instrumentation** is automatically and invisibly made part of the definition of each simulated component that is to be monitored during a run. This is done by arranging that the class of every component to be monitored is a specialization of the general *instrumented-box* class. The basic data structures and procedures for monitoring simulated components and maintaining the organizational relationships between each component and its related instrumentation are inherited through this general, ancestral class and are thus made a separate, substantially independent consideration in the design.

A further partitioning of concerns is employed to separate out the definition of the application programming language

---

*This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Greg Byrd was supported by an NSF Graduate Fellowship and by the Stanford University Department of Electrical Engineering.

[1]SIMPLE and CARE were developed by the authors at the Knowledge Systems Lab of Stanford University. SIMPLE is a descendent of PALLADIO [Brown83] optimized for the subset of PALLADIO's capabilities relevant to hierarchical design capture and simulation. It is written in Zetalisp [Weinreb81] and currently runs on Symbolics 3600 machines and TI Explorers.

interface and its support (as provided by CARE) from the underlying information flow control governing component behavior. The behavioral descriptions of components (which are expressed as sets of condition/action rules) deal generically with gating information, independently of the structure of the information, between ports of the component and its internal state variables. This is separated in the component model definitions from the functions performed to create and manipulate the information so gated. The simulated implementation of the application programming language support facilities, on the other hand, relies only on the specifics of the information and its structure and plays no part in gating it between the components of the system. Changing the definition of the application language is thus done independently of changing component flow control behavior. The application programmer and the implementer of the application language interface may use whatever data structures seem suitable to them, be they numbers and keywords or procedure bodies and execution environments. The simulation system doesn't care.

The *component probe* definitions, that is, the specifications of what information should be captured for each component type, are separated from the descriptions of the behavior of such components. In designing for flexibility in the instrumentation system, it turned out to be important to further divide the information presentation from the information collection issues. The mapping from particular component probes to particular *instrument panels* and the transformations to be applied to the information as it passed from a given kind of probe to a given panel (and between panels) is captured in the *instrument specification*. This is a definition of what kinds of panels are included in an *instrument*, how they fit on an *instrument screen*, how they are labeled and scaled, and what information from which kinds of probes are displayed on each panel. The instrument specification also indicates what kinds of probes are to be connected to which kinds (that is, which classes) of components in the system.

mechanisms of a multiprocessor applications language. These specify the interface used to provide the program input to the multiprocessor system being simulated.[2] The definitions used to generate component probes are associated with each library component to be monitored. There may be several such definitions, each appropriate to measuring a different aspect of the associated component's operation. An instrument specification selects from these definitions, elaborates them with selections from a set of *probe operation modules* to include any pre-processing (for example, a moving average) to be calculated by the probe, and indicates under what conditions what information from the probe is to be sent to which panels of the instrument and how it is to be transformed and displayed there. Instrument specifications also partition the screen among the panels of the instrument. The end product of these design time interactions is an *instrumented circuit* and an *instrument*. The instrument comprises a set of instrument panels and a set of constraints relating them to the instrument screen. The instrumented circuit ties together instances of components, probes, and panels for a simulation run.

For each defined class of component and its associated probes, the design time interactions produce code bodies that accomplish simulation operations during a run. It is an attribute of the underlying Lisp base of the simulation system that changes in these definitions have immediate effect even during a simulation run -- an important capability during debugging.

2. STRUCTURE AND COMPOSITION

Design time interactions to specify a system include the establishment of component relationships. Such specifications can be said to accomplish the composition of the system from its components and so define its structure. SIMPLE supports hierarchical composition: components may be described in terms of a fixed set of relationships among their

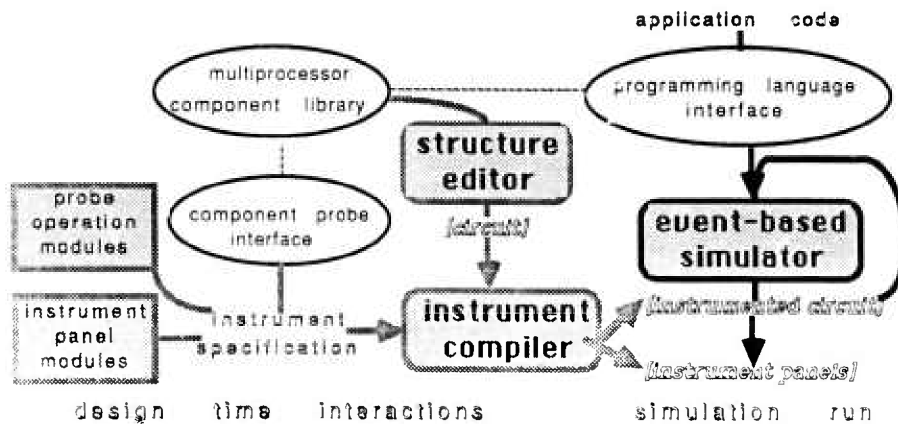


Figure 1-1: Design Time Interactions and Run Time Representations

Putting together all the definitions of components, component probes, panels, instruments, applications interfaces, and inter-component relationships is done in a set of design time interactions by a system architect. These interactions are used by the simulation system to generate efficient run time representations so that simulation performance goals can be met. Figure 1-1 illustrates the partition between design time interactions and simulation run time operation. Structure editing pulls together components from the component library to produce a *circuit*. Associated with some components in the library, there are definitions for the syntax and underlying

[2] The language primitives supplied can be used to define multiprocessor language interfaces for either shared-variable or value-passing paradigms. As supplied, the language interface built on these primitives supports value-passing on streams between objects but alternative interfaces can be (and have been) easily defined in terms of the given primitives.

sub-components. Additionally, such composite components may have function beyond what can be inferred strictly from their composition. All this can then be included a higher level composite (as shown in figure 2-1) and so on indefinitely until the top level "circuit" the system structure is reached.

The behavior induced on a composite component from its parts changes according to the behavior of its parts. Thus, for example in figure 2-1, if at any time during a simulation the function of CARE *operator* components is changed by redefining their operation, the behavior of the nine-site grid is in immediate correspondence.[3]

net-output, the *fifo-buffer*, the *operator*, and the *evaluator*. The net-input, net-output and fifo-buffer accept (or block), route, and buffer transmissions. They do so in accordance with a dynamic, flow-controlled, multicast, cut-through communications protocol as described in [Rynd87a]. The evaluator does the real work of the application: evaluating the application of functions to their parameters. The operator does the overhead work associated with such evaluations: for example, scheduling processes and sending and receiving (but not routing) messages.

In keeping with the objective of focusing simulation cycles on the aspects of the simulation particularly relevant to multiprocessor operation, the behaviors of the net-input,

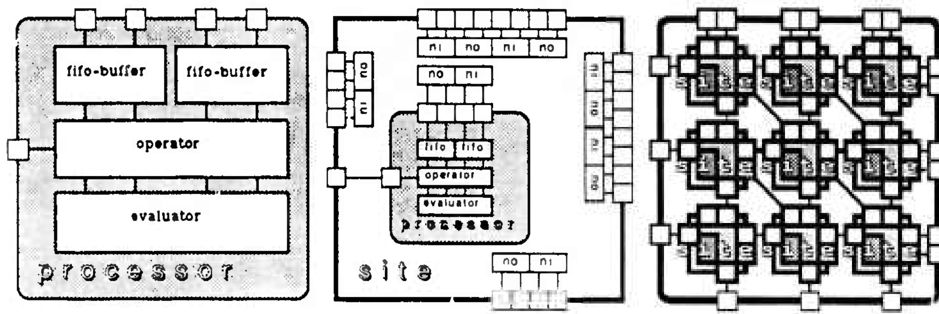


Figure 2-1: Hierarchical Composition

Composition is described graphically and interactively in SIMPLE by picking a previously specified component type from a menu, placing it in relationship to other components with "mouse" movements, and, through the same means, specifying the connections between its selected ports and those of other components (as indicated in figure 2-2).

Through another menu selection, ports can be defined for the new composite component so that it, in turn, can be fitted into yet higher level structures. Such external ports can be connected directly to ports of sub-components "within" the composite. If this is done, information appearing on that external port will be the responsibility of the connected sub-component. By this same means, a component previously described as a base level component, can be redefined as a composite of yet lower level elements as its design is elaborated with further details.

Components and (internal) connections can also be deleted from a library component and replaced with substitute components. After all sub-components and connections have been added, deleted, elaborated, and replaced as required, the completed structure can then be entered into a library of components and used in turn to compose higher or equivalent level components.

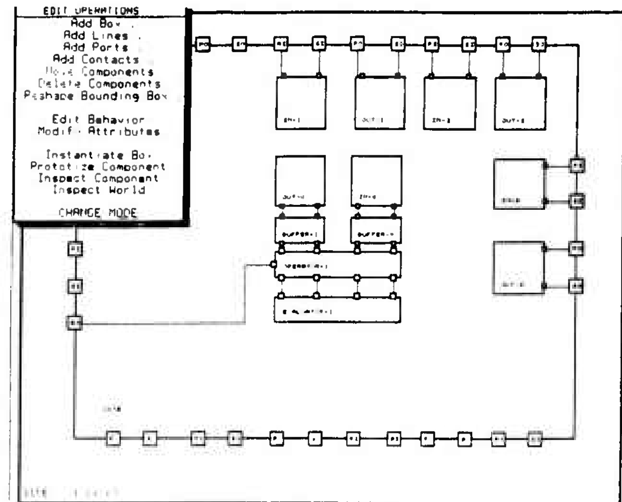


Figure 2-2: Graphic Structure Specification

2.1. CARE Base Components

CARE supplies a small library of system level base component types. Currently these are the *net-input*, the

net-output, and fifo-buffer component classes are defined in fair detail, that is, at the register transfer level. Routing operations are described procedurally and assumed to occur within a time set by a parameter to the simulation. As indicated previously, the simulation of the operator and evaluator is broken into two aspects: the control of the flow of information and the functions performed on that information. The former is described in terms of SIMPLE behavior rules (as documented in section 3), register transfer by register transfer. The latter is described directly in terms of procedures and the simulated time taken by such procedures is modeled. In the case of the operator, this is done as a function of the number of storage cells manipulated during an operator procedure. In the case of the evaluator,

---

[3] However, for reasons concerning simulation performance and because of their relatively low frequency, changes in the number and names of the internal state variables of components and the structural relationships between sub-components of a composite are not reflected in an already instantiated circuit. Changes in the internal structure of a CARE *site* library component, for example, will be reflected only in circuits instantiated after the change took effect. For this reason and to reduce long term storage requirements and load time for the fundamentally iterative circuits that we primarily study, we do not keep files of instantiated circuits. They are instantiated as needed from a high level library component with the same prototypical structure.

this is done as a function of the execution time used by the machine executing the simulation, that is, the simulation vehicle.

2.2. CARE Composite Components

The prototypical composite component supplied with CARE is the *site*. As supplied, it includes net-inputs and net-outputs for up to eight "neighboring" components (generally other sites), a net-input and a net-output with associated fifo-buffers for local receptions and transmissions, and, finally, an operator and evaluator as described above. Specializations of the site, for example, the *torus-site*, exist in the library to fit the site into alternative topologies by supplementing the site routing and wiring procedures as appropriate to the topology.

2.3. Automatic Composition in CARE

Although any connection of components can be created by the means noted previously, for some repetitive, well patterned systems of connections, composition can be automated. The CARE library includes a component, the *iterated-cell*, which represents a template for the creation of composite components by iteration of a unit cell. The unit cells (for example, the torus-site) are specializations of other components (for example, the site) as just discussed. The specializations include a method for responding to a request to provide a wiring list. Such a list associates each source port of a cell with the corresponding destination port (in terms of port names) and the position of the destination cell relative to the source cell in the iterated structure. The iterated cell component uses this information to make the required connections between each of its constituent cells.

3. SPECIFYING BEHAVIOR

SIMPLE is an event based simulator. The behavior of a simulated component is described in terms of responses to the events pertinent to that component. A component's response may include consequent events to be handled by the simulator as well as direct operations on component state. Assertion of consequent events and the responses to them (involving further consequences) drives the simulation. When there are no more events to handle, the simulation is complete.

To maintain modularity in a simulation system, responses to simulation events should be local to the affected component and its defined ports, that is, its connection to the remainder of the simulated system. The composition system of the simulator maintains the relationship between ports of one component and those of other components connected to them. Assertions relative to a port of a component are thus systematically translated to events pertinent to components connected to it. This is the general mechanism for event propagation between components. In a limited number of cases, a direct operation on a related component may be appropriate. With fair warning about its possibility of abuse, a facility is provided to accomplish this.

3.1. Behavioral Rules

The behavior of a component is described in terms of its responses to pertinent events. Each event stipulates the component affected, its port or state variable signalled with an assertion, the asserted value, and the simulated "time" of the event. The time of an event may be thought of as the "current" simulation time. Differences in event times represent the temporal relationship between events. Event times in SIMPLE simulations are monotonically increasing.

For each type of component, there is a procedure to handle pertinent events. The arguments to the procedure are those stipulated by the event (as just described). The procedure tests for conditions and, as satisfied, asserts or directly effects consequent actions. The conditions may include arbitrary predicates on the event parameters and the state variables of the component.

Event based simulators are based on the assumption that state and port variables remain unchanged until explicitly modified. Synchronous designs, that is, those in which the opportunities for state change are temporally quantized to a clock, can be modeled in such implicitly asynchronous, event based simulators by asserting the clock signal on a port of each and every clocked component of the simulated system. If only some of the components in a system need take action on each clock signal, there is an obvious inefficiency in this approach that is crippling for systems with even a modest number of components.

If, however, event times in an event based simulator are restricted to integers, the clock can be assumed. All that is needed is a way to detect the event for which a boolean combination of conditions as strobed by an assumed clock is first met. Primitive condition predicates are supplied for detecting an "edge" (a value changed by the current event) with a coincident "level" (a value set before the current event) of two ports or state variables of a component in either of the two possible event sequences. The predicate both-states in the example evaluator behavior rule shown in figure 3-1 has these semantics.

Figure 3-1 illustrates the generality of SIMPLE behavioral descriptions. The underlying object-oriented programming system, Flavors [Weinreb81], in which SIMPLE is implemented provides for direct reference of component state variables. The conditions and actions of behavior rules for a component then need only name the component's port or state variable (as stipulated in the definition of that component type) to get or change the appropriate value in the component instance for which the event is pertinent. Actions may include arbitrary procedures: for example, the procedures user-evaluate and queue-take in the given example.

```
;;If the evaluator is ready and there is at least one runnable process...
((or (both-states Evaluator-Status⁴ 'ready Evaluator-Queue-Status 'some)
     (both-states Evaluator-Status 'ready Evaluator-Queue-Status 'full))
;;... make it current, start evaluation, and adjust status as per removal.
 (setq Evaluator-Status 'busy)                            ;block rule
 (assert-state Evaluator-Status 'busy now)                ;next event
 (setq Current-Evaluation (queue-take Evaluator-Queue))   ;note process
 (user-evaluate Current-Evaluation now)                   ;execute it
 (send self :evaluator-queue-decreased now))              ;note change
```

Figure 3-1: Example Condition/Action Behavior Rule

3.2. Using Methods

The environment for the execution of the procedures defining responses to events includes the state variables and ports of the component instance for which the event is pertinent. These procedures are Flavor *methods* [Weinreb81] (in this case corresponding to the :ApplyRules message) of the component type and, as just noted, refer implicitly to the state variables of the component instance handling the event. Other methods may be defined for simulated components: for example, the :evaluator-queue-decreased method invoked in figure 3-1. Such methods have proved to be a natural way to realize the functional operations of components not described by behavior rules.

The composition system leaves information about the enclosing and contained component instances for each simulated component in system defined state variables of that component. With this information, methods directly referencing the ports and state variables of such related components may be invoked as needed. This is a useful but

---

[4] By convention, component state variables are written in capitalized form.

sharp-edged facility. The warning about loss of modularity given previously applies here.

4. INSTRUMENTATION

The results of a simulation are primarily the insights it provides into the operation of the simulated system. The "insight" we frequently experienced using an early version of the simulation system was that more interesting results could have been produced by the run just completed if only the instrumentation had been different. With this in mind, the design for the current version of the simulation instrumentation system was aimed at flexibility. This was attained without significant performance impact by building efficient run-time system structures before each run, as outlined in section 1.1, from the declarations defining the instrumentation.

The organization of the instrumentation system is pictured in figure 4-1. The simulator interacts with component instances through assertions, that is, calls on an `assert` function, in behavior rules (the methods associated with `:ApplyRules` messages). All instrumented components are specializations of an *instrumented-box* (as well as other classes). After each invocation of `:ApplyRules` for such components, the `:ApplyRules` method for a generic instrumented-box is applied. This causes invocation of the `:trigger` method for each *component-probe* associated with that component. Since this flow of measurements is accomplished by means invisible to the the writer of behavior methods for a component, the concerns surrounding component design are effectively partitioned from component instrumentation. The remainder of this section details these "invisible" means used to accomplish measurement flow during a simulation run as the measurements are staged from components through component probes to instrument panels.

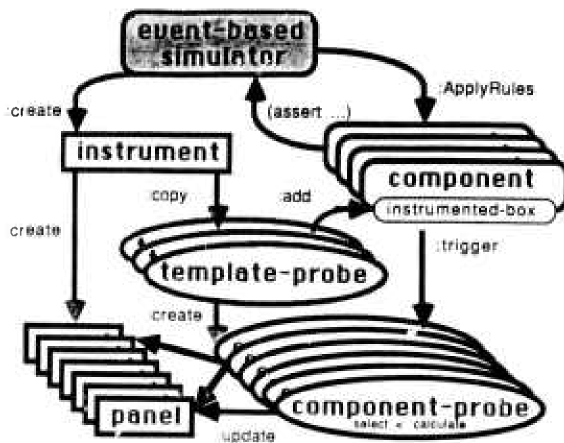


Figure 4-1: Instrument System Organization

4.1. Component Probes

The first filtering of events is done by component probes. Some events cause no further measurement activity since, as it turns out, not all events merit action on the part of the instrumentation system. The parameters of the event and the ports and state variables of the instrumented component dealing with the event are available to the component probe as are the state variables of the probe itself. Each piece of the selected information is tagged with an identifying keyword and passed along as the parameters of the `:trigger` method along with a keyword identifying the type of component probe, a number representing the current event time, and a pointer to the component with which the information is to be associated in the display. This pointer might be to some component related to the one actually handling the event, for example, the component enclosing it.

Component probes may be composed of predefined probe operation modules to do standard calculations (for example, moving averages) and then to forward the results to selected panels. In order to automate the composition of probes to accomplish such operations, each of these operations is chained together by invoking the method for that probe that is associated with the system-defined message name of the generic next operation. Thus, the `:trigger` method calls the `:calculate` method of the probe which, in turn, calls its `:select` method which, finally, calls the `:update` method of the selected panels associated with the probe. Probes are composed by naming them as specializations of appropriate probe operation modules (for example a `:calculate` module for moving averages) as desired. The default, if no specializations are stipulated, is to pass through information without change to all the panels associated with a probe.

Information flow between components and panels is accomplished by the component probes associated with each instrumented component. The creation of such component probes and their association with appropriate components (by execution of `:add` methods) accomplishes the instrumentation of a circuit. This is done when an instrument is created. During simulation initialization, the components of the circuit (and their sub-components) to be instrumented are (recursively) examined by each *template probe* defined for the instrument to see if they are to be monitored. If so, the `:copy` method for the given template probe is invoked to create a new instance of the appropriate component probe and add it to the probes connected to the component. Each template probe previously received the identifiers for the panels to which its clones should send information. These will be the panels identified when a component probe invokes the `:update` method.

4.2. Instrument Specifications

The operations performed by an instrument panel are to:

- *Find* information previously stored according to the component pointer supplied by the `:update` method;
- *Link* new data structures as needed (to save such information) to other such structures of the panel;
- *Save* in these data structures the results of expressions that reference indicated keyed information from the `:update` parameters and the prior contents of the structures;
- *Send* the results of periodic analyses on the information associated with a panel for display by the same panel or by some other; and
- *Show* processed information in the manner specified for the panel.

The defaults for the panel operations supply the most commonly required specifications implicitly, so simple operations are simply specified. These defaults can be overridden as needed and either predefined or user specified alternatives for the panel operations can be selected in their place. Arbitrarily complex (Lisp) expressions can be used to specify the transformations between the information provided by a probe and that saved and displayed by the panel.

These transformations and all the default overrides for the panel operations that are stipulated in the instrument declaration are scanned when a new instrument is created for a simulation session. They are compiled at that time into

code bodies referenced by run time control blocks associated with each panel. A simulated system is instrumented by examining all of its components and attaching to each component the copies of template probes specified by the instrument definition that are appropriate for the component (by means of calls on the :copy and :add methods for the probe). This can be a many to many relationship as shown in figure 4-2.

Component probes to measure "load" and "latency" are specified in the given example for each operator and evaluator in the circuit. The "load" and current "connection" for each net-output is also to be monitored. Some panels, for example the one showing "consumer-limited" processes, receive inputs from only one type of component probe, those measuring evaluator latency. Others, such as the one measuring "process-latency" receive inputs from more than one kind of probe (in this case, from probes measuring operator latency as well as those measuring evaluator latency). A way must thus be provided to distinguish the type of probe sending information to a panel; this is described in the next section.

Some probes send information to only one panel, for example, the net-output connection probes. Others monitor information which is needed by several panels, for example, the operator latency probe. Transformation of the raw information provided by a probe will need to be specialized to the information expected by each panel receiving it. A general way to stipulate these transformations is stipulated in the next section.

evaluators of the system.

The balance between the "availability" of the evaluator and operator of each site, that is, the complements of their respective loads, is displayed during the simulation as events are processed that change this measure. In order to avoid capturing information at too fine a temporal granularity, previously gathered information for a given site is overwritten if it is within a given sampling interval of the new information. Information that is beyond a given history range is dropped. The scale of availabilities displayed is fixed between 0 and 1.0. The panel specification to declare all this and to also stipulate the axis labels is shown in figure 5-2.

5.2. Scrolling Line Plot Panels

An example of a *scrolling line plot panel* is shown in the right half of figure 5-1. This panel sums the loads seen by the resources in the simulated system and displays this as a strip chart, the "system history". Some of the same probe load information used by the previous panel is used in this panel as well, but with different transformations defined in the panel specification as shown in figure 5-3.

Line plot panels may have two independently scaled vertical axes. For the system history panel shown, the sum of network loads as indicated by the net-output components of the system is plotted against the left axis and the sum of the processing loads provided by the current average of the sums of the operator and evaluator loads is plotted against the right axis.

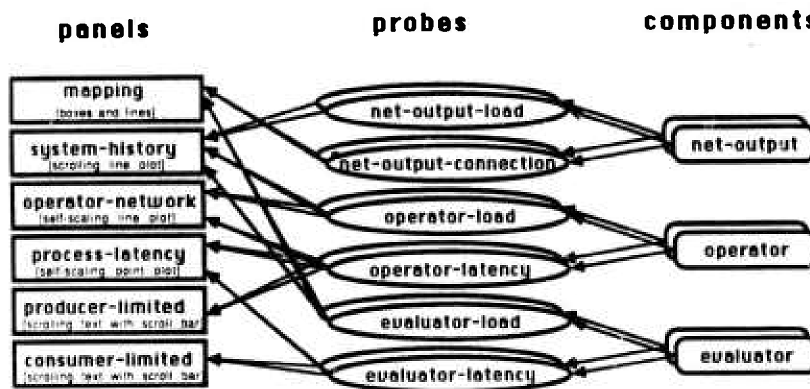


Figure 4-2: Instrument Probe and Panel Relationships

5. EXAMPLE PANELS

Some example panels are described in this section to give a feel for the instrumentation possibilities available in CARE and elaborate on how the requirements described in the previous section for probe type identification at a panel and per panel specialization of the information provided by a probe are handled.

5.1. Point Plot Panels

The first panel (shown in the left half of figure 5-1) is an example of a *point plot panel* used to generate a scatter plot. As an option, only points representing simulated activity over a limited past history from the most recent event time are kept for display. In this example, resource load[5] information is provided by the operator-load and evaluator-load component probes attached respectively to the operators and

Event time is plotted on the horizontal axis. The update-history function uses the component pointer to find the information previously saved for that component and records the current event time as the (:simulator :time) so that it may be used to display information correctly on the horizontal axis. The current sums of the evaluator loads and the operator loads measured by the system are stored in a record for the given event time (or a prior event time within the specified sampling interval) by the calls to the save-sum function specified as part of the *save* operation.

---

[5]Resource load is defined as (1 - 1 / (1 + aggregate-queue-length)) where the aggregate queue-length is the sum of the lengths of all queues providing work for the resource.

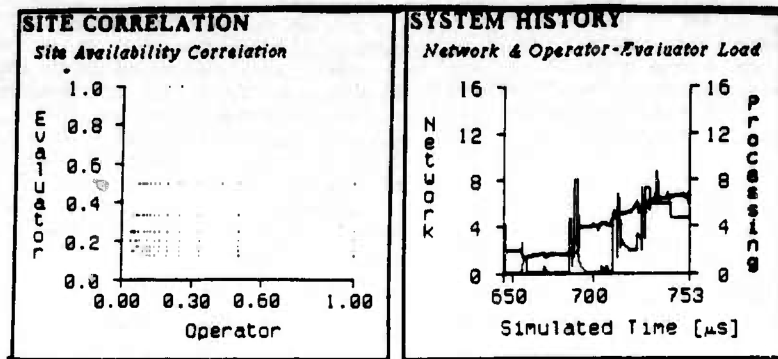


Figure 5-1: Point Plot and Scrolling Line Plot Panels

```
'((("Operator") (0 1.0) (- 1 (:operator-load :busy)))      ;Bottom axis
  (("Evaluator") (0 1.0) ((- 1 (:evaluator-load :busy)))) ;Left axis
  :find (find-sample-distinct (:simulator :time) ,sampling-interval)
  :show (recent-history (:simulator :time) ,point-panel-history-range 0))
```

Figure 5-2: Site Correlation Panel Specification

```
'((("Simulated Time [us]") (,history-range) (:simulator :time))  ;Bottom
  (("Network") (0 ,sites) (:net-output-load :busy save-sum))     ;Left
  (("Processing") (0 ,sites)                                      ;Right
     (average (:evaluator-load :busy save-sum)
              (:operator-load :busy save-sum)))
  :find (update-history (:simulator :time) ,sampling-interval)
  :show (recent-history (:simulator :time) ,history-range 0))
```

Figure 5-3: System History Panel Specification

5.3. Self Scaling Line Plot Panels

Figure 5-4 illustrates both the self scaling of displays and the use of a display analysis operation. For this self scaling line plot panel, two pieces of data are collected for each operator in the system: the load on the operator, shown on the right axis, and the latency of the information it has most recently received. This last item is provided by the operator latency probe in two parts: (1) the interval between the creation of the information and its receipt by the net-input feeding the operator and (2) the interval between such receipt and the operator taking action on it. There are thus two curves plotted on the left axis. The specification stipulates a list for the left axis display. The elements of this list are the "net delay" and the sum of this measure and the "operator delay" monitored by the operator latency probe. Since both delays are non-negative, their sum must be at least as large as either one taken alone: the two curves may be superimposed but can not cross. The difference between the two curves is the incremental delay added by the operator.

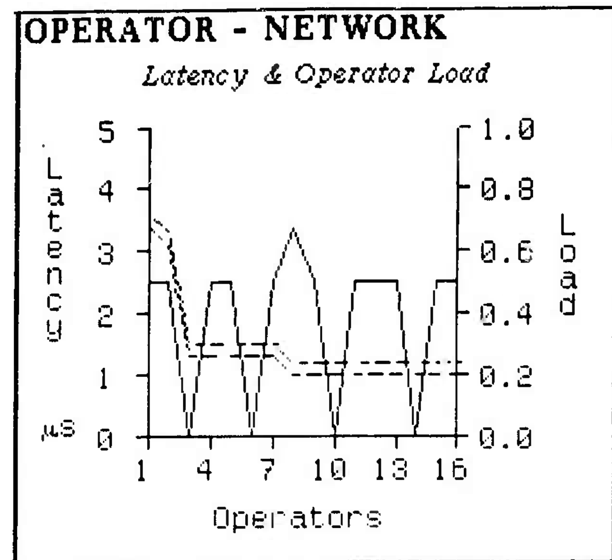


Figure 5-4: Self Scaling Line Plot Panel

The panel specification for the operator-network panel is shown in figure 5-5. In addition to transformations shown previously, an analysis function is stipulated for the *send* operation of the panel. The information saved from each of the probes sending `:update` messages to the panel is to be sorted from the greatest to the least values of the associated sum of delays described above. This information is to be saved as the operator latency rank and used as such to determine the position on the horizontal axis that the delay and load information will be displayed.

```
'((("Operators") (1 .sites) (:operator-latency :rank))
  ((("Latency" "us")) (0 nil) ;Second string: 90 degree baseline shift
   ((:operator-latency (:net-delay (+ :net-delay :operator-delay)))))
  (("Load") (0 1.0) (:operator-load :busy))
  :send (sort-arrays
         ((,#'> (:operator-latency (+ :net-delay :operator-delay))))
         ((:operator-latency :rank))))
```

Figure 5-5: Operator-Network Panel Specification

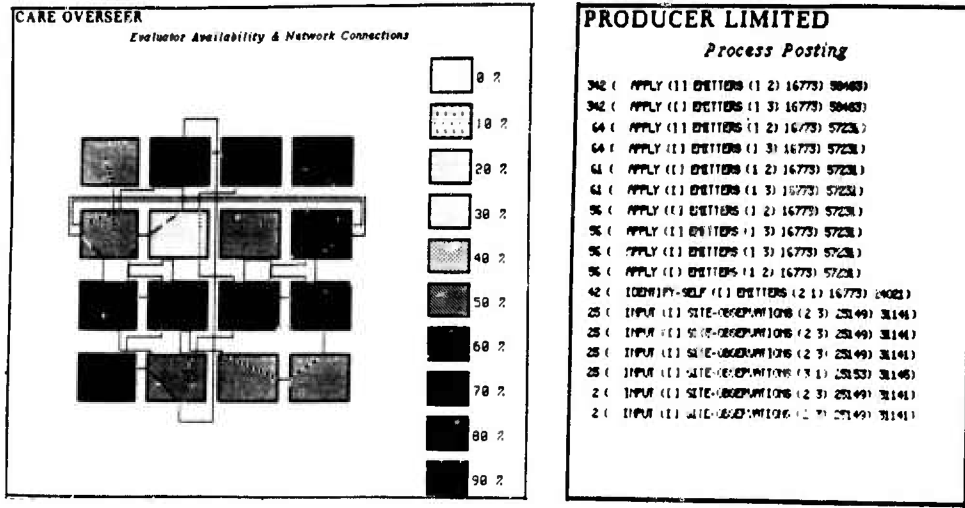


Figure 5-6: Boxes and Lines Panel and Scrolling Text Panel

5.4. Boxes and Lines Panels

Perhaps the most intuitively satisfying of the types of panels available is the *boxes and lines panel*, a graphic representation of a circuit showing its components and their interconnections. An example of such a panel is shown the left part of figure 5-6. This class of panels uses information left behind by the structure editor when the circuit was defined. Its form is thus automatically generated. The position of the components ("boxes") and the connections between them ("lines") in the display are used to animate system operation. In the example shown, the shading (or color) of the boxes is used to indicate the availability of the *evaluators* in the simulated system as the simulation proceeds. Darkest shades indicate highest availability, that is, empty queues for utilization of the resource; lighter shades indicate lower availability, that is, longer queues. The lines between boxes indicate communication paths that are in use, that is, not ":free" at the time of the most recent *show* operation for the panel.

The panel specification for the *mapping panel*, an instance of a boxes and lines panel, is shown in figure 5-7. There are two specifications for the panel: one for the boxes and one for the lines. The specification for boxes in the panel stipulates that the availability of evaluators in the sites corresponding to the boxes displayed controls the shading of those boxes. The scale is defined to run from 0 to 1.0. The specification for lines in the panel uses the connection information reported for the net-output to determine line placement on the display. When the status is reported as :free, the connection information is dropped from the panel and the corresponding lines are removed.

5.5. Scrolling Text Panels

Sometimes, the most appropriate way to display information is to show it as text. Based on a similar facility provided by the underlying Lisp system, the *scrolling text panel* provides a

```
'((("Evaluator Available") (0 1.0) (- 1 (:evaluator-load :busy))))
'((("Packet Trace") nil (:net-output-connection :points))
  (("Packet Status") nil (:net-output-connection :status))
  :find (find-and-remove ,#'eq (:net-output-connection :status) :free)))
```

Figure 5-7: Mapping Panel Specification

scrollable window into lines of text. In the right part of figure 5-6, the delay in each process execution while waiting for something to do, that is, the event time interval spent waiting for an appropriate task to appear on a certain stream of tasks, is shown together with the process that finally produced the awaited work. This information is sorted so that the text lines appear from the greatest stream waiting interval to the least.

The values and formats used for display in a scrolling text panel are defined much as in previously defined panels. Format control strings take the place of scale information. As usual, values are described by a list of forms, each one of which specifies the transformations to perform on information received from probes. The example specification in figure 5-8 shows the generality with which probe information can be incorporated in Lisp expressions to produce transformation specifications. The information used to generate the value for the second field of the text display is based on the origin of the task packet that arrived on the stream the process was waiting for.

Many of the CARE parameters are specified as *overrides*. If not specified, the corresponding performance is taken as measured on the simulation machine. Thus, the *evaluation override*, that is, the time to perform an evaluation can be specified as non-nil in order to fix the time that each user evaluation will take. (This is useful in making runs repeatable for debugging). The time that it takes to switch context can be specified as the *stack group switch override*. Similarly, the time to create a process control block and a stack context for that process can be taken as given rather than measured by specifying respectively the *process block creation override* and the *stack group creation override*.

The time required for operator execution is modeled in terms of the number of words the operator must manipulate in handling a given message. The manipulation time per word is specified by the *operator word touch time*. Lastly, the performance of the communication subsystem is specified as *communication cycles*. This is done in terms of the minimum number of evaluator data path clock times (that is, event

```
'((() ("~4D ~A")
   ((fix (:stream-waiting :interval))    ;first field
    (let* ((origins (packet-origin (:stream-waiting :packet)))
           (origin (if (listp origins) (first origins) origins)))
      (remote-address-local origin))))  ;second field
  :send (sort-arrays ((,#'> (:stream-waiting :interval))) nil))
```

Figure 5-8: Producer Limited Process Panel Specification

5.6. Noting Simulation Parameters

The CARE component models are parameterized through menu interaction as shown in figure 5-9 to allow easy variation of their performance characteristics relative to each other. Additionally, the site model parameterizes alternative routing strategies: *directed*, that is, blocking when progress can not be made toward the goal; *spiraling* around the goal if progress toward it is blocked; and *dithering*, that is, routing away from the goal even if only the last link towards it remains to be acquired. The rate at which each site accepts application data is also a parameter, the *data rate* and can be used by an application to control how hard it drives the simulated system.

times) required for a 32-bit word to pass a given point in the network. Thus the parametric specification, "4 communication cycles", dictates that 8 bits may cross such a boundary each time the evaluator passes through one event time. If the communications path were narrower or the base communication clock rate were lower, a higher number would be specified.

The last example of SIMPLE panels is the annotation panel as illustrated in figure 5-10. This is used to (automatically) record the date, time, and parameters of the simulation run as well as any other information the user chooses to keyboard into it.

Simulation Parameters	
Data Rate [µs]:	25.0
Evaluation Override [µs]:	NIL
Stack Group Switch Override [µs]:	1.0
Process Block Creation Override [µs]:	4.0
Stack Group Creation Override [µs]:	20.0
Operator Word Touch Time [µs]:	0.2
Communication Cycles:	4
Routing:	**DIRECTED** SPIRALING DITHERING
Exit ☐	Quit ☐

Figure 5-9: Parameter Menu

NOTES
6/25/86 08 54 4d 32 DIRECTED Cycles, Acceleration 2, Creation 2000µs, Switch 250µs, Evaluation 25µs, Data 15µs

Figure 5-10: Annotation Panel

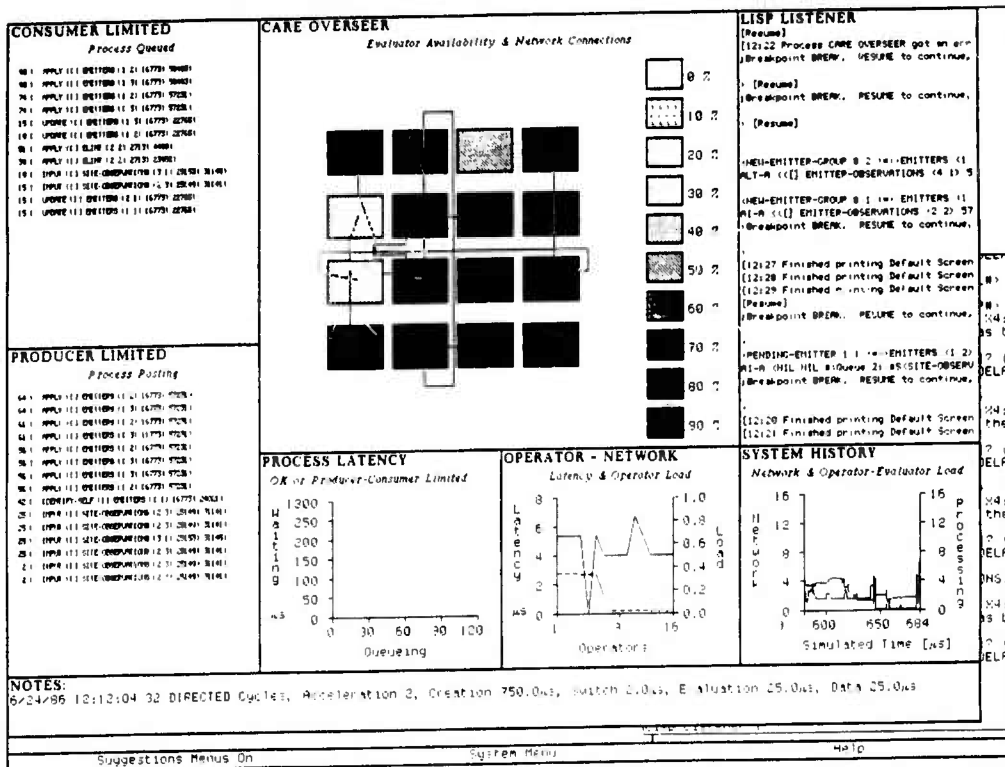


Figure 5-11: Overseer Instrument

5.7. An Instrument Screen

All these panels are put together in an instrument screen according to a set of layout constraints manipulated by the underlying window system. The finished screen might look like figure 5-11. The instrument screen is redrawn at a rate set by the user. By experience, it is often better to update the screen at a frequency low enough to let the user interpret each screen comfortably than at the maximum rate possible. This approach also restricts the computing resources consumed by the instrumentation system. More focused approaches to controlling instrumentation load on the system include the ability to freeze selected panels and disconnect selected probes during a simulation run.

6. USING PROGRAM DEVELOPMENT TOOLS

The SIMPLE/CARE simulation system is integrated into the underlying Lisp machine program development environment. The objects and data structures at both the component model and application language interface have abstraction interfaces that provide summary state information when they are displayed in text form. These text abstractions are "mouse sensitive" in the development machine environment and so can be inspected at successively finer levels of detail as desired.

In figure 6-1, the net-output components of the site at grid coordinates (3 2), the particulars of the net-output on the east side of the site (that is, net-output-3), and a summary of all the sub-components of the site at (3 2) are being inspected. This same kind of view into the progress of a simulation is provided in the debugging process and may, as shown in figure 6-2, refer to the conceptual entities of the application that is driving the simulated system.

In the example shown in figure 6-2, a distributer process running on the evaluator at site (1 1) has made an improper call on the update-locale function during execution of its :start method. It might have been appropriate to investigate this situation in terms of the modeled components. That could be done, for example, using the debugger to inspect the evaluator component, its enclosing site, related net-output components, or whatever else at the component model level seemed relevant. In this case, what was done was to use a few mouse clicks to indicate interest in the source file for the distributer :start method generating the problem. It was brought up for review and control was then transferred to an editor using the underlying program development environment as shown in figure 6-3.

Because of the implementation system chosen for the realization of SIMPLE/CARE, at any point in the simulation, procedures either in the application or in the component models can be modified, incrementally recompiled (within a few seconds), and be made effective for all calls on them -- even those in the interrupted stack frame. Thus simulation execution can be backed up to some previous point in the stack frame and retried (given that intermediate side effecting code, if any, is safely re-executable).

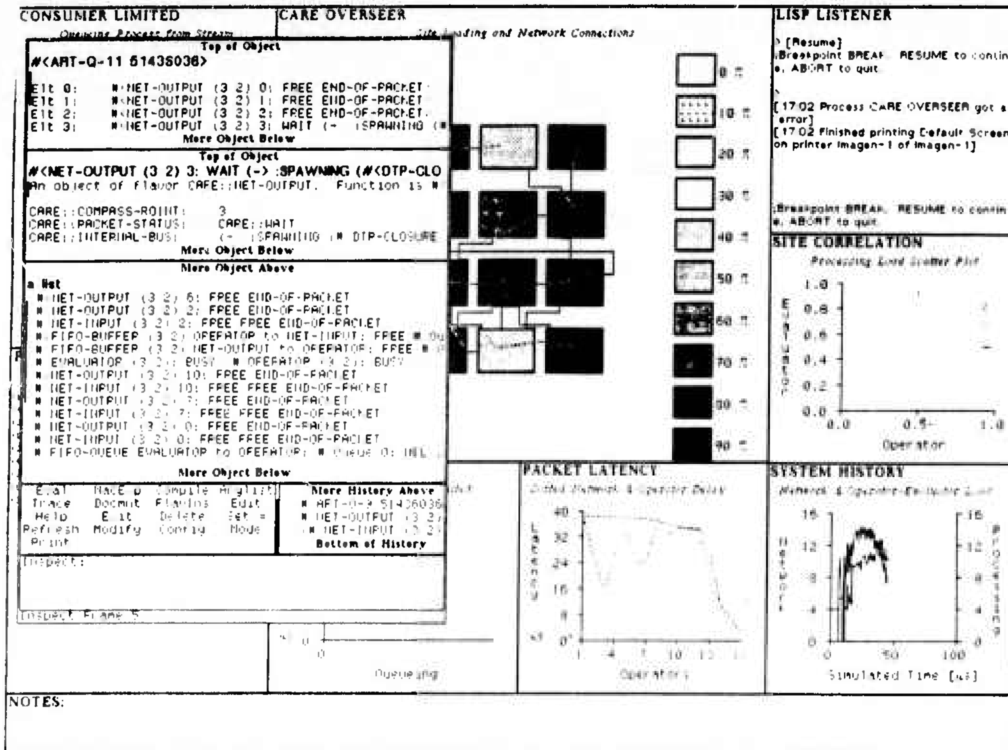


Figure 6-1: Inspecting Simulated Components

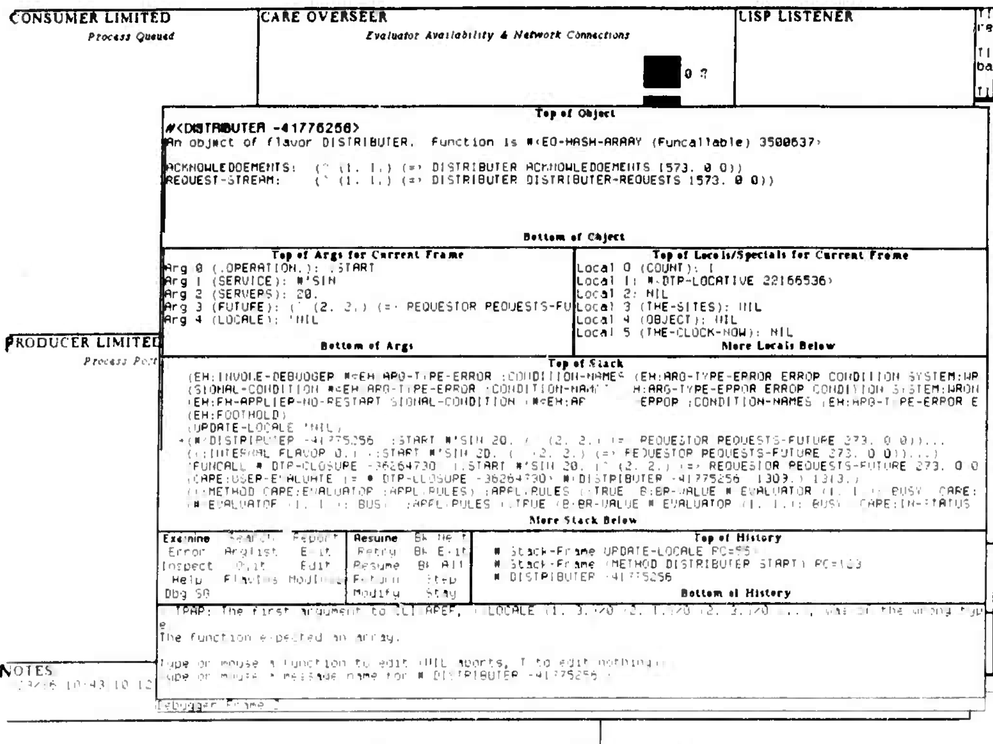


Figure 6-2: Debugging A Simulation

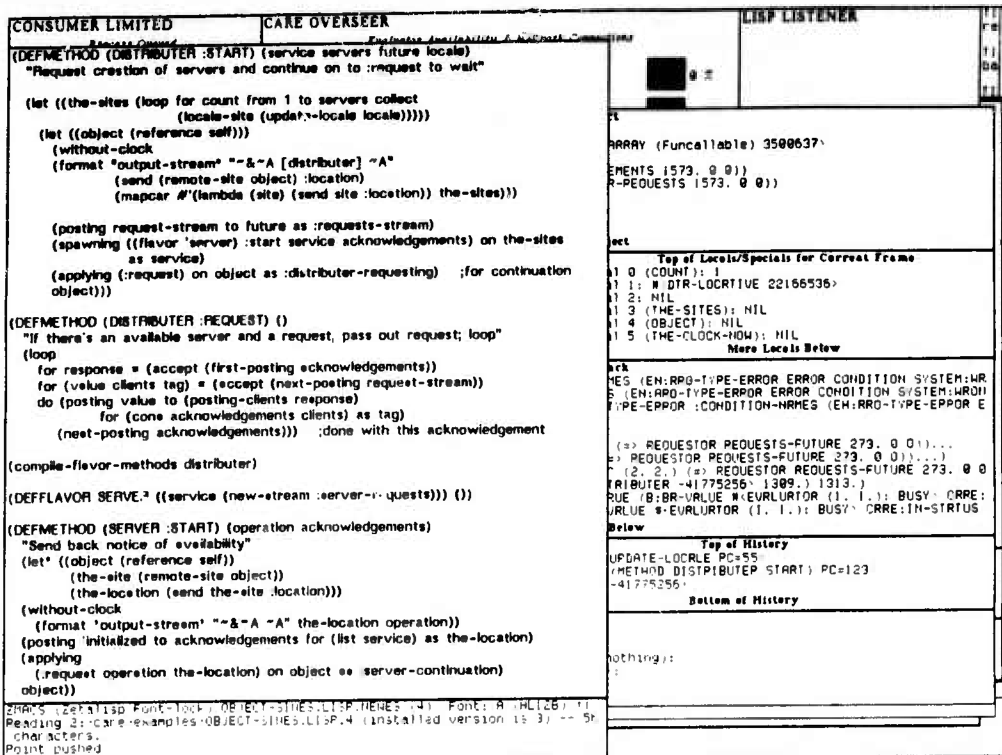


Figure 6-3: Changing Application Code

7. CONCLUSIONS

The goals of simulation flexibility and simulation environment completeness have been dealt with in the ways described throughout this paper. In summary, the system is flexible in that it supports:

- Arbitrary data types and lengths in simulation. The information whose flow and creation is controlled by simulated components may be of arbitrary complexity -- from numbers and keywords to procedure bodies and execution environments.
- Instantaneous effect of definition change at both the application and component modeling level (even during a simulation run).
- A broad range of instrumentation customization. Customizations may involve arbitrary expressions for probe data transformations, many to many probe to panel mappings, information from summary analyses on one panel's data included in another, and control of what state is saved and for how long.
- Separation of probe and component definitions to facilitate their independent modification.
- An application language interface that is easily extended or changed without recasting the information flow control described by the component behaviors.

While there is always room for additional capability[6], SIMPLE/CARE is a usefully complete system. It now includes:

- Supplied components for a network multiprocessor simulation with many of their parameters customizable by menu interactions.
- A hierarchical structure editor that currently provides automatic grid and torus composition operators. (Automated composition of richer topologies, such as hypercubes, has been provided for in the basic design).
- A rule language that supports a synchronous design style without incurring the overhead of (naive) synchronous simulation.
- Method invocation for functional simulation that is integrated into the behavioral simulation rule system and which provides for operations by and on both local and hierarchically related components.
- Method specification design aids provided by the underlying program development environment (for example, method dictionaries and quick access to method sources from the debugging system).
- An evolved set of panel templates providing sorted, scrollable text lines as well as self and fixed scaling, "two and a half" dimensioned, history sensitive displays which may be scatter plots, strip charts, line graphs, intensity maps, and signal animations.

[6] A *histogram panel*, for example, is just now being added to the system

We set off to build a multiprocessor simulation system with performance adequate for the understanding of multiprocessor systems executing significant applications. The SIMPLE/CARE simulation system has been used to study the operation of "expert systems" of respectable size [Brown86]. Depending on instrumentation load, these studies have involved simulation runs from 20 minutes to several hours each. While faster would surely be better, performance has proven adequate to these needs.

Acknowledgements

This work stands on the shoulders of its predecessor, the Palladio system, designed and implemented by Harold Brown and Gordon Foyster. Our functional goals were more restrictive than theirs so we had the luxury of design by simplification. Without their implementation base, it would have been hard to know even where to begin.

Many hands and minds have contributed to the development of SIMPLE/CARE. We are particularly indebted to the work of Russ Nakano who started off to do a simple learning exercise and ended up doing a particularly careful modeling of a intricate signalling protocol.

References

[1] Brown, Harold, Christopher Tong, and Gordon Foyster.
PALLADIO: An Exploratory Design Environment for Integrated Circuits.
IEEE Computer 16, December, 1983.

[2] Harold D. Brown, Eric Schoen, and Bruce A. Delagi.
An Experiment in Knowledge-Based Signal Understanding Using Parallel Architectures.
Technical Report STAN-CS-86-1136 or KSL-86-69, Stanford University, October, 1986.

[3] Greg Byrd, Russell Nakano, and Bruce Delagi.
A Point-to-Point Multicast Communications Protocol.
Technical Report KSL-87-02, Knowledge Systems Laboratory, Stanford University, January, 1987.

[4] Daniel Weinreb and David Moon.
Lisp Machine Manual.
Symbolics, Cambridge, MA, 1981.

Considerations for Multiprocessor Topologies*

Greg Byrd†
Knowledge Systems Laboratory
Stanford University
Stanford, CA 94305

Bruce Delagi
Worksystems Engineering Group
Digital Equipment Corporation
Maynard, MA 01754

Abstract

Choosing a multiprocessor interconnection topology may depend on high-level considerations, such as the intended application domain and the expected number of processors. It certainly depends on low-level implementatin details, such as packaging and communications protocols. We first use rough measures of cost and performance to characterize several topologies. We then examine how implementation details can affect the realizable performance of a topology.

1 Introduction—Design Constraints and Opportunities

The base for development of general purpose multiprocessor systems as for computer systems today generally is given by the design constraints and opportunities established by evolving semiconductor design and manufacturing processes. The VLSI design medium brings a new perspective on cost: switches are cheap; wires are expensive. In modern microprocessors, communication costs dominate those associated with logic. Power and cooling budgets are spent driving wires and overwhelmingly, chip area is dedicated to wiring rather than logic [17]. To an increasing degree, the dominant delays are associated with driving lines rather than the accomplishment of logic functions per se. One implication is that, all other things being equal, smaller, simpler processors can be expected to have shorter operation cycles than larger, more complex designs [18]. They are also likely to be available in a more recent, higher performance base technology.

At the system level, the consequence of relatively expensive communication is that performance is enhanced if the design establishes that whenever a lot of information has to move in a short time, it does not have to move far. Significant locality of high bandwidth links is a goal. Among the highest bandwidth links in a computer system is that connecting the processor and memory. Early computer systems separated these pieces and put a bottleneck between them to accommodate the packaging realities of the time: processors were implemented with electronic means, memory with magnetic, and their power requirements and EMI characteristics were best dealt with separately. There are new realities now: close coupling of processors with local memory is preferred.

With these design constraints in mind, we consider a multicomputer implementation based on a set of processor/memory pairs connected by a communications topology. Many topologies have been proposed [8] and have been compared in terms of theoretical cost and performance measures [16]. We argue, however, that the realizable performance of these topologies are closely linked to details of system packaging.

*This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875.

†Supported by an NSF Graduate Fellowship and by the Stanford Dept. of Electrical Engineering.

2 Interprocessor Connection Topologies

Connection schemes between processing sites can be compared with respect to their cost and performance as a function of the number of sites connected. For a particular connection scheme, if the cost grows no faster than the number of sites and the performance grows at least as fast, that scheme can be described as *scalable*. A rough measure of cost is the number of input-output ports required for connection. A rough measure of performance is the number of links in the topology divided by the largest number of links that must be traversed, and thus occupied to accomplish a transmission, in order to get from one node in the network to another. This indication of the bound on the number of independent, concurrent transmissions we will call the *concurrency* of the network.

For some topologies, the concurrency of a network may *understate* performance as actually experienced in a given application: to the extent that there is locality of reference in transmissions, the number of links actually traversed may be better approximated by a constant than some function of the number of connected sites. Network concurrency may also *overstate* performance of one topology with respect to another: to the extent that the time to traverse links is not the same for all topologies, those that have non-uniform link costs (perhaps due to physical distance considerations applied to the realized lengths of links) will deliver less performance than the concurrency measure suggests. This is because in these cases, logical adjacency due to high dimensionality is merely apparent—embedding the topology in the dimensionality of space available tends to incur just those expenses related to physical distances that the topology was expected to eliminate.

2.1 Topologies With Scalable Concurrency

Several topologies are shown in Table 1 which have scalable concurrency. As the number of sites is increased, the network grows enough to support the consequential additional traffic. In fact, by this measure of performance, the last three of these four topologies scale performance equally well. However, as will be described, there are other considerations to weigh.

Table 1: Scalable Concurrency Topologies. [n = # processors]

Topology	Number of Ports	Longest Path	Concurrency
Completely connected	$O(n^2)$	$O(1)$	$O(n^2)$
Crossbar	$O(n^2)^a$	$O(1)$	$O(n)$
Banyan	$O(n \log n)$	$O(\log n)$	$O(n)$
Boolean k-cube ($n = 2^k$)	$O(n \log n)$	$O(\log n)$	$O(n)$

^{a}The number of links is $O(n)$.

In the crossbar and completely connected topologies, the number of ports, a first approximation to cost, grows quadratically with the number of nodes in the network. Weighing cost and concurrency, then, we might prefer the banyan and boolean k-cube (also known as "hypercube") topologies.

By these measures, there does not seem to be a clear-cut choice between the banyan and the hypercube. A more sophisticated measure of cost would take into account the area required for laying out the topology in a plane [11]. The banyan may have a slight edge in this category[1], but both layouts require relatively long wires, which is undesirable if link transit time dominates switching time.[2]

A major difference between the two topologies is that switching and routing are centralized at the processor in the hypercube, whereas the switching in the banyan is distributed throughout the network. To the extent that storage is required at the switch (as in [3]), it becomes more economical to centralize the switch and utilize the local storage of the processor. For this reason, we prefer the hypercube.

2.2 Topologies With Scalable Cost

There are alternative topologies not as richly connected as those just considered. The topologies in Table 2 all have fixed degree connectivity, so they all have scalable cost as measured by port count. Unfortunately, none of them has scalable concurrency. So, at least among the ten representative topologies discussed, there is no topology that has cost-performance characteristics intrinsically superior to all the others.

Concurrency for the ring and the bus topologies does not increase at all as the number of processors increases. Given no guarantee of transmission source to target locality, these seem unsuitable for systems with a large number of processors (e.g., > 100).

The perfect shuffle and cube-connected cycles (CCC) topologies emulate the $O(\log n)$ latency of the hypercube, but the number of links is linear with the number of processors, so concurrency does not scale. Also, if we measure cost in terms of layout area, the cost of the perfect shuffle ($O(\frac{n^2}{\log^{3/2} n})$) and CCC ($O(\frac{n^2}{\log^{3/2} n})$) [15] do not scale and so will not be considered further.

The tree, grid, and torus topologies all have fixed degree connectivity and have the optimum $O(n)$ area requirement.The tree has a slightly better capacity measure and a lower latency bound. Note, however, that the tree provides no alternate communication paths (useful in network balancing and defect tolerance) and has a bottlenecking root.[3] Connections might be added to provide alternate paths, but, as we will see in the next section, physical link considerations may make the grid or torus a better choice.

Table 2: Scalable Cost Topologies. [n = # processors]

Topology	Number of Ports	Longest Path	Concurrency	Area
Ring	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Global bus	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Perfect shuffle	$O(n)$	$O(\log n)$	$O(\frac{n}{\log n})$	$O(\frac{n^2}{\log^{3/2} n})$
Cube-connected cycles	$O(n)$	$O(\log n)$	$O(\frac{n}{\log n})$	$O(\frac{n^2}{\log^2 n})$
Binary tree	$O(n)$	$O(\log n)$	$O(\frac{n}{\log n})$	$O(n)$
Grid/Torus	$O(n)$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n)$

[1] The area required to lay out a hypercube in a plane is $O(n^2)$ [2], where n is the number of processors. Since "banyan" actually denotes a class of interconnections it is difficult to make a general statement about its layout. However, let us consider a particular banyan network, the omega network [10], which is $\log n$ stages of perfect shuffle connections. The perfect shuffle has area $O(\frac{n^2}{\log^{3/2} n})$ [15], so we would expect $\log n$ perfect shuffles to require area $O(\frac{n^2}{\sqrt{\log n}})$, which is a slightly better bound than for the hypercube. Other types of banyans, with different fan-in, fan-out, and connectivity characteristics might have even smaller bounds.

[2] See Section 3.

[3] We might be able to deal with this by increasing the bandwidth of the links as we proceed toward the root, for example with "fat trees" [12].

3 Link Costs—Examining The Free Lunch

Most studies of topologies assume a constant cost for link traversals as the number of links increases. This is a useful approximation if the time to drive and receive link signals is constant with link length and large compared to signal transit time on the link. However, this is increasingly not a good assumption both as the underlying feature size of the component technology decreases and as we consider larger numbers of sites in a system. Given a fixed circuit feature size, topologies with scalable concurrency, as discussed in Section 2.1 suffer increased link lengths and thus longer signal transit times—with possibly increasing drive times—as the number of processors increases. Alternatively, given a fixed volume of circuits in these topologies and decreasing circuit feature size, the number of processors in the system increases but so does the ratio between link lengths and feature size. Thus relative to the circuit delay times which are dependent on (and decrease with) circuit feature size, the link transit times become increasingly a more important consideration.[4]

Topology has to be viewed as a dependent variable determined principally by the packaging technology of the system. As an example, consider the recursive-H layout for the binary tree (Figure 1) under the assumption that link transit time dominates switching time. Now consider the grid in Figure 2, which can be laid out in the same area. If transit times dominate, then shorter links and more switching sites will likely shorten the point-to-point communications cycle time and improve the realized capacity of the network.[5] Furthermore, additional data paths allow dynamic routing of messages, and additional computing resources make the grid potentially more powerful than the tree.

Though the torus appears to suffer from extremely long wires which "wrap around" the edges, a simple renumbering of the processors in a grid brings each one within two hops of its logical neighbors[6] (see Figure 3). Thus, we can effectively create a torus by changing the routing algorithm of a grid. Alternatively, we could keep the original torus connections and lay out the processors as in Figure 3(h), resulting in links which are at most twice as long as those for a grid. In the remainder of the paper, we will speak of the grid bearing in mind construction of the torus in these terms.

4 A Packaging Example

We are now faced with two topologies: one with scalable performance—the hypercube—and one with scalable cost—the grid. The arguments presented above suggest that, all else being equal, the communication cycle time for the hypercube would be greater than that of the grid, due to its long links. Even so, the average message latency of the hypercube may still be smaller, due to its high connectivity. To get a better understanding of the relative performance of the two systems, we should examine how they might actually be implemented in near-future technology.

In the mid-1990's we would expect a 0.5-μm MOS fabrication process to be available [7]. We will assume that the complexity of our processor is comparable to today's typical 32-bit microprocessor. The

[4] The dependence of communication delays on signalling lengths as circuit feature size decreases depends on assumptions made on the thickness and thus the resistivity of associated interconnects. Uniform scaling leads to relative signalling times that increase quadratically with distance [19]. Detailed analysis of the equations of voltage and current in VLSI wire implementations (including consideration of the non-linear characteristics of signal drivers) demonstrated linear dependences [1] but were done assuming that the interconnect (and field oxide) thicknesses did not decrease at all while all other dimensions scaled with the circuit feature size of the technology [17]. Another approach imagines a hierarchy of interconnect of increasing thicknesses with distance [13] to achieve signalling times that grow only with the logarithm of the distance. Yet another approach accepts resistive links but given control over both minimum and maximum wire lengths and use of high impedance receivers, notes that it is possible to counter dispersive losses with reflective voltage doubling at the receiving end of a point to point link [9]

[5] The assumption made here is that the message routing is relatively independent of the computing activities at a processing site, so there is no penalty associated with being routed at a processing site rather than a switch.

[6] This approach is attributed to B. Zippel.

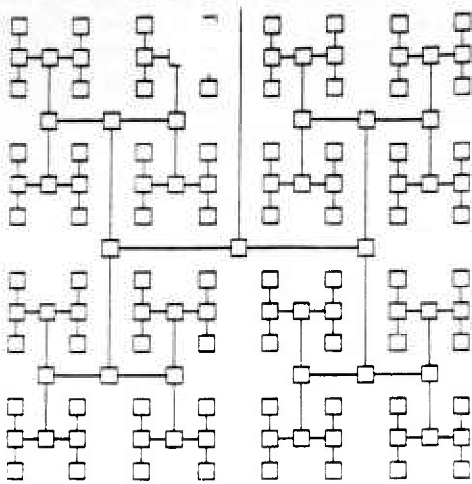


Figure 1: Recursive-H binary tree.

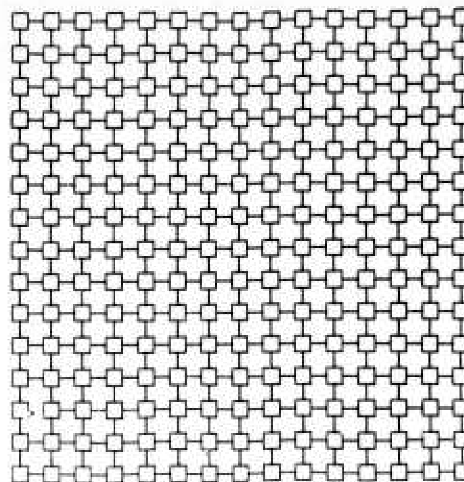
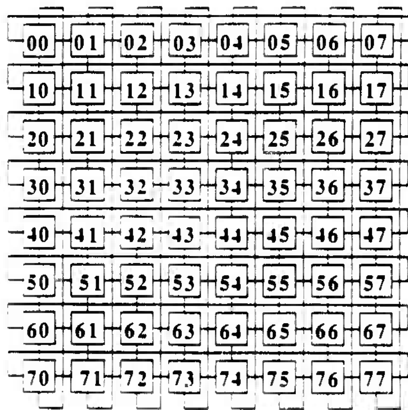
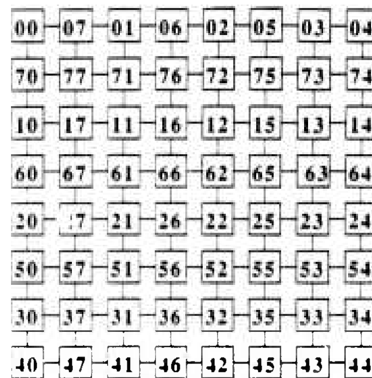


Figure 2: Two-dimensional grid.



(a)



(b)

Figure 3: Torus (a) and renumbered grid (b).

MicroVAX 78032 chip [4], for example, is implemented in 3-μm technology; it measures about 8.5 mm on a side. Using 0.5-μm technology, we could expect a similar processor to require around 1.5 mm on a side. Let us allow 256K bytes (2M bits) of local memory for our processor. Fujitsu's megabit RAM using 1.4-μm technology takes 54.7 mm^2 [6]. If the dimensions of the Fujitsu chip are about 10 mm by 5.5 mm, then a 0.5-μm version would be 3.6 mm by 2.0 mm. Two of these (since we want 2M bits) would be around 3.6 mm by 4 mm. As an approximation, then, each processing element, including a processor, 256K bytes of local memory, and switching and routing circuitry could be expected to fit onto a 5 mm x 5 mm piece of silicon.

Even as devices shrink, die sizes continue to grow. By the mid-90's, the state-of-the-art chips may be as large as 15 mm on a side. Each chip would be expected to have 400–600 I/O pads [14]. Therefore, we could put up to nine processing sites on a single die.

The dice could be flip-mounted on a silicon [5] or ceramic [9] substrate with thin-film transmission lines and integrated capacitors. In [9], the maximum length for 5-μm-thick lines is around 20 cm, so we will assume a 10x10 cm module size, on which we can easily place up to 36 dice. We will assume on the order of 1000 I/O pins per module [5].

Consider first packaging a (32x32) 1024-element octal grid, in which each processor is connected to eight neighbors. With nine processors (arranged as a 3x3 grid) on a die, 32 (bi-directional) communication links must come off the chip through the I/O pads, so no more than 18 pads could be used per channel. A module can carry 324 processors, arranged as an 18x18 grid. The entire system, then, could fit on four modules (with room to spare). The communications links from two sides of the 18x18 grid (105 bidirectional channels) must go off-module. Thus, each channel could use 10 pins—one pin for clock and status information and four for data, in each direction.

Now consider a 1024-element hypercube (a "10-cube"). To allow for more complex wiring and easier packaging, we will assume that each die contains eight processors, and each module will hold 32 dice, for a total of 256 processors per module. (Extra space might be used to provide redundant processors for fault tolerance.) Again, only four modules are required to package all 1024 processors. Each processor has ten

bidirectional links to its logical neighbors. If the eight processors on a die are wired as a 3-cube, then seven channels from each processor must go off-chip. Five of these channels are connected to other processors on the same module, but two must go off the module. With only ~ 1000 I/O pins for 512 bidirectional channels, it appears that a 1-bit combined control/data stream is all that can be supported for the hypercube communications. If we decrease the number of processors per die to four (and possibly add more memory), we can use separate wires for control and data but the wires will be longer.

Note that in both cases the module pin-out is the limiting factor for channel width, rather than the chip pin-out. If more off-module I/O pins are available, things will look better, but there will still be around a 5-to-1 ratio of the number of required off-module channels in the hypercube as compared to the grid. As mentioned before, the average interconnect length for the grid will be much shorter than that for the hypercube. Therefore, the grid offers shorter (i.e., faster) and wider communication paths than the hypercube when implemented in projected near-future technology.

5 Beyond Topology

As the previous example indicates, the electrical and physical characteristics of the circuit packaging in a system may dictate the scheme used to wire the nodes together. In addition, the communications protocol, that is, the actual signalling on the links are an important component of achievable performance. There are many relevant details—for example:

- Dynamic routing, selecting available links as needed, is useful in balancing load and thus allows more of communication resources of the system to be well used throughout a computation.
- Cut-through routing, making a routing decision on the fly as a packet is received, reduces buffer requirements in the system and minimizes latency experienced in network transit.
- Local flow control, signalling transmission delays back to the source based on local blockage information, together with single "word" buffering and transmission validation at each network input and output port allows the source to complete a validated transmission in a time that does not depend on the size of the network.
- Point to point multicast, sending (approximately) the same packet to multiple targets using common resources to the largest degree possible—coupled with dynamic, cut-through routing, flow control, and word level buffering and transmission validation—provides "virtual busses" precisely as and when they are needed.

A point-to-point protocol utilizing these mechanisms is described in [3].

6 Conclusion

Communications performance of practical systems depends first of all on available packaging technology and second on protocol considerations. No topology considered here has both scalable cost and performance, so the topology chosen must be in the context of the number of processors targetted. For a thousand processors or so, given the assumptions on mid-1990's technology discussed earlier, the grid (or torus) seems an appropriate choice. The performance of the grid will depend on the signalling protocol and will be best predicted through application simulations detailed enough to relect design decisions made at that level.

References

[1] G. Bilardi, M. Pracchi, and F. P. Preparata. A critique and an appraisal of VLSI models of computation. In H. T. Kung, B. Sproul, and G. Steele, editors, *VLSI Systems and Computations*, pages 81–88, Computer Science Press, Inc., Rockville, MD, 1981.

[2] G. Brebner. Relating routing graphs and two-dimensional grids. In P. Bertolazzi and F. Luccio, editors, *VLSI: Algorithms and Architectures*, pages 221–231, Elsevier Science Publishers B.V., Amsterdam, 1985.

[3] G. T. Byrd, R. Nakano, and B. A. Delagi. *A Point-to-point Muticast Communications Protocol.* Technical Report KSL-87-02, Knowledge Systems Laboratory, Stanford University, January 1987.

[4] D. W. Dobberpuhl, R. M. Supnik, and R. T. Witek. The MicroVAX 78032 chip, a 32-bit microprocessor. *Digital Technical Journal*, (2):12–23, March 1986.

[5] Capt. B. J. Donlan, J. F. McDonald, R. H. Steinvorth, M. K. Dodhi, G. F. Taylor, and A. S. Bergendahl. The wafer transmission module. *VLSI Systems Design*, 7(1):54–58, 88–90, January 1986.

[6] Electronic News, July 1,1985.

[7] C. K. Lau, et. al. A high performance half-micron gate CMOS process for VLSI. In *Proceedings of the 1985 International Conference on Computer Design: VLSI in Computers*, IEEE, October 1985.

[8] T. Feng. A survey of interconnection networks. *Computer*, 12–27, December 1981.

[9] C. W. Ho, D. A. Chance, C. H. Bajorek, and R. E. Acosta. The thin-film module as a high-performance semiconductor package. *IBM Journal of Research and Development*, 26(3):286–296, May 1982.

[10] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, December 1975.

[11] C. E. Leiserson. *Area-Efficient Graph Layouts (for VLSI).* Technical Report CMU-CS-80-138, Carnegie-Mellon University, August 1980.

[12] C. E. Leiserson. Fat-trees: univeral networks for hardware-efficient supercomputing. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 393–402, IEEE, 1985.

[13] C. Mead and M. Rem. Minimum propagation delays in VLSI. In *Caltech Conference on VLSI*, pages 433–439, January 1981.

[14] D. Nelsen. Personal Communication.

[15] F. P. Preparata and J. Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[16] D. A. Reed and H. D. Schwetman. Cost-Performance bounds for multimicrocomputer networks. *IEEE Transactions on Computers*, C-32(1):83–95, January 1983.

[17] C. L. Seitz. Ensemble architectures for VLSI—a survey and taxonomy. In *1982 Conference on Advanced Research in VLSI*, MIT, January 1982.

[18] C. L. Seitz. Experiments with VLSI ensemble machines. *Journal of VLSI and Computer Science*, 1(3), 1984.

[19] C. L. Seitz. Self-timed VLSI systems. In *Caltech Conference on VLSI*, pages 345–355, January 1979.

RUM: A Layered Architecture for Reasoning with Uncertainty*

Piero P. Bonissone, Steven S. Gans, Keith S. Decker
GE Corporate Research and Development Center
1 River Road, K1-5C32A
Schenectady, New York 12301

ABSTRACT

New reasoning techniques for dealing with uncertainty in Expert Systems have been embedded in RUM, a Reasoning with Uncertainty Module. RUM is an integrated software tool based on a frame system (KEE) that is implemented in an object oriented language. RUM's capabilities are subdivided into three layers: *Representation, Interference*, and *Control*. The Representation layer is based on frame-like data structures that capture the uncertainty information used in the inference layer and the uncertainty meta-information used in the control layer. Linguistic probabilities are used to describe lower and upper bounds of the certainty measure attached to a Well Formed Formula (*wff*). The source and the conditions under which the information was obtained represent the non-numerical meta-information.

The Inference layer provides the uncertainty calculi to perform the intersection detachment, union and pooling of the information. Five uncertainty calculi, based on their underlying Triangular norms (T-norms), are used in this layer.

The Control layer uses the meta-information to select the appropriate calculus for each context and to resolve eventual ignorance or conflict in the information. This feature enables the programmer to declaratively express the local (context dependent) meta-knowledge that will substitute the global assumptions traditionally used in uncertain reasoning. The control layer also provides a context mechanism that allows the system to focus on the relevant portion of the knowledge base, and an uncertain-belief revision system formula (*wffs*) in an acyclic directed deduction graph.

1. Introduction: Reasoning with Uncertainty in Expert Systems

The trend followed by most approaches for reasoning with uncertainty has shown an almost complete disregard for the fundamental issues of automated reasoning, such as the proper *representation* of information and meta-information, the allowable *inference* paradigms suitable for the representation, and the efficient *control* of such inferences in an explicitly programmable form. 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. Most approaches do not even recognize the need for having an explicit control of the inferences.

Specifically, the non-numerical approaches [Cohen 1983a;83b; Doyle 1983], 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 [Genesereth 1982], 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 set 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 conflicting information is detected [Shafer 1976], conflict 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 conflicting part of the information. The non-conflicting portion is then normalized and propagated as if the conflict never existed.

---

* This is a modified version of the paper *RUM: A Layered Approach to Reasoning with Uncertainty* that will appear in the Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87), Milano, Italy, August 1987.

The development of RUM's underlying theory was supported by the Defense Advanced Research Projects Agency (DARPA) under USAF/Rome Air Development Center 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 of the U.S. Government.

The implementation of RUM, was not part of (nor was it funded by) DARPA contract F30602-85-C-0033. This description is included only for the purpose of illustrating the technology integration and transition efforts that GE, within the spirit of the Strategic Computing Initiative, has undertaken outside of its contractural obligations.

This lack of awareness for the fundamental issues of automated reasoning has been the driving force for compiling a list of requirements (desiderata) that each reasoning system handling uncertain information should satisfy. Following the typical structure of automated reasoning techniques, the list of requirements has been organized in three layers: *representation, inference*, and *control*. The extension of this explicit layered separation from *crisp*-reasoning systems to *uncertain*-reasoning systems is a natural step leading to a better integration of the management of uncertainty with the various techniques for automated reasoning.

An in-depth treatment of the layered desiderata can be found in an earlier paper [Bonissone 1986]. In this article we describe the theory, design, and implementation of RUM, a Reasoning with Uncertainty Module whose layered architecture reflects the requirements described in the desiderata. In the next two sections we will summarize RUM's underlying theory and design, with a particular focus on its control layer. In the last section we will discuss the conclusions of this work.

2. RUM's Underlying Theory

Preliminary theoretical results were presented in two previous publications [Bonissone 1985; 86]. This section summarizes some of those results and provides a unified framework for their interpretation and use in RUM's architecture. A philosophical motivation for the RUM's three layer organization can also be found in [Bonissone 1987a].

2.1 Term Sets of Linguistic Probabilities

In expert system applications, users and experts must frequently provide subjective assessments of probability. Due to the difficulty of eliciting precise and consistent numerical certainty values, we have suggested the use of term sets of linguistic probability. 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.

A term set of linguistic probabilities is the set of symbols $L = \{L_1, L_2, ..., L_n\}$. The meaning of each term $L_i \in L$ is represented by a fuzzy number on the [0,1] interval. The fuzzy number's membership distribution, $\mu_{L_i}(x)$, is defined as the mapping:

$$\mu_{L_i}(x): [0,1] \rightarrow [0,1] \quad \text{for all } x \in [0,1].$$

A computationally more efficient way to characterize a fuzzy number is to use a parametric representation of its membership function. This parametric representation [Bonissone 80; 85] 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 $\mu_{L_i}(x)$ is defined as:

$$\mu_{L_i}(x) = \begin{cases} 0 & \text{if } x < (a_i-\alpha_i) \\ (\frac{1}{\alpha_i})(x-a_i+\alpha_i) & \text{if } x \in [(a_i-\alpha_i), a_i] \\ 1 & \text{if } x \in [a_i, b_i] \\ (\frac{1}{\beta_i})(b_i+\beta_i x) & \text{if } x \in [(b_i, (b_i+\beta_i)] \\ 0 & \text{if } x > (b_i+\beta_i) \end{cases}$$

For compactness of notation, we will denote the meaning of the term set element L_i as the list $(a_i, b_i, \alpha_i, \beta_i)$. RUM provides the user with four different term sets that can be used to define the granularity desired in the subjective assessment of probability. The four term sets contain five, seven, nine, and thirteen elements, respectively. The following table illustrates one of the term sets, the nine element *L-nine*:

Index	Symbol	Meaning
1	*impossible*	(0 0 0 0)
2	*extremely_unlikely*	(.01 .02 .01 .05)
3	*very_low_chance*	(.1 .18 .06 .05)
4	*small_chance*	(.22 .36 .05 .06)
5	*it_may*	(.41 .58 .09 .07)
6	*meaningful_chance*	(.63 .80 .05 .06)
7	*most_likely*	(.78 .92 .06 .05)
8	*extremely_likely*	(.98 .99 .05 .01)
9	*certain*	(1 1 0 0)

TABLE 1: The Nine Element Term Set L-nine

RUM's representation layer allows the user to characterize the lower and upper bounds of the certainty of a given fact by using elements of a selected term set.

2.2 T-norms: Definitions and Equivalence Classes

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 [Bonissone 1985]. A T-norm is defined as a mapping $T: [0,1]^2 \rightarrow [0,1]$ which is monotonic, commutative and associative. The boundary conditions of a T-norm (i.e., the evaluation of any T-norm at the extremes of the [0,1]x[0,1] unit square) satisfy the truth tables of the logical AND operator. The T-conorms are defined in terms of the T-norms and a negation operator, by using a generalization of DeMorgan's duality. Thus, for a suitable negation operator, such as $N(a) = 1-a$, the T-conorm $S(a,b)$ is defined as:

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

In a previous paper [Bonissone 1985], six parametrized families of T-norms and dual T-conorms were discussed and analyzed. Of the six parametrized families, one family was selected due to its complete coverage of the T-norm space and its numerical stability. This family, originally defined by Schweizer & Sklar [Schweizer 1963], is denoted by $T_{Sc}(a, b, p)$, where p is the parameter that spans the space of T-norms. More specifically:

$$T_{Sc}(a,b,p) = (a^{p} + b^{p} - 1)^{\frac{1}{p}} \quad \text{if } (a^{p} + b^{p}) \geq 1 \quad \textit{when } p < 0$$

$$T_{Sc}(a,b,p) = 0 \quad \text{if } (a^{p} + b^{p}) < 1 \quad \textit{when } p < 0$$

$$T_{Sc}(a,b,0) = \lim_{p \to 0} t_{Sc}(a,b,p) = ab \quad \textit{when } p < 0$$

$$T_{Sc}(a,b,p) = (a^{p} + b^{p} - 1)^{\frac{1}{p}} \quad \textit{when } p > 0$$

Its corresponding T-conorm, denoted by $S_{Sc}(a,b,p)$ is defined as:

$$S_{Sc}(a,b,p) = 1 - T_{Sc}(1-a, 1-b,p)$$

We have seen that the use of term sets determines the granularity with which the input certaintly is described. This granularity limits the ability to differentiate between two similar calculi; the numerical results obtained by using two calculi whose underlying T-norms are very close in the T-norm space will fall within the same granule in a given term set. Therefore, only a finite, small subset of the infinite number of calculi that can be generated from the parametrized T-norm family 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 [Bonissone 1985] where eleven different calculi of uncertainty, represented by their corresponding T-norms, were analyzed. To generate the eleven T-norms, the parameter p in Schweizer's family was given the following values: -1, -0.8, -0.5, -0.3, 0 (in the limit), 0.5, 1, 2, 5, 8, and ∞ (in the limit).

The experiment showed that five equivalence classes were needed to represent (or reasonably approximate) any T-norm. The corresponding five uncertainty calculi were defined by the common negation operator $N(a)=1-a$ and the DeMorgan pair ($T_{Sc}(a,b,p)$, $S_{Sc}(a,b,p)$) for the following values of p:

$$p = -1 \quad T_1(a,b) = max\ (0,\ a+b-1) \qquad S_1(a,b) = min\ (1,\ a+b)$$

$$p = -0.5 \quad T_{Sc}(a,b,-0.5) = \max(0, a^{0.5} + b^{0.5} - 1)^2$$
$$S_{Sc}(a,b,-0.5) = 1 - \max(0,[(1-a)^{0.5} + (1-b)^{0.5} - 1])^2$$

$$p \to 0 \quad T_2(a,b) = ab \qquad S_2(a,b) = a + b - ab$$

$$p = 1 \quad T_{Sc}(a,b,1) = \max(0, a^{-1} + b^{-1} - 1)^{-1}$$
$$S_{Sc}(a,b,1) = 1 - \max(0,[(1-a)^{-1} + (1-b)^{-1} - 1])^{-1}$$

$$p \to \infty \quad T_3 = min(a,b) \qquad S_3(a,b) = max(a,b)$$

RUM's inference layer provides the user with a selection of the five T-norm based calculi described above. In the inference layer, they are referred to as T_1, $T_{1.5}$, T_2, $T_{2.5}$, T_3, respectively.

3. Design of RUM's Layered Architecture

RUM's architecture is based on three layers: *representation*, *inference*, and *control*. In the first layer (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, linguistic probabilities are used to describe the lower and upper bounds of the certainty measure associated with the Well Formed Formula (*wff*). Various term sets of linguistic probabilities (with fuzzy-valued semantics) provide different granularities of the certainty measure. Non-numerical meta-information, describing the source and the conditions under which the information was obtained, is also represented in this layer.

In the second layer (the inference layer) we define five uncertainty calculi based on their underlying Triangular norms (T-norms). Any operation required by an uncertainty calculus can be expressed in terms of its T-norm and a negation operator. From past experience, it was noted that T-norm based calculi have various computational advantages: they are *truth-functional*, *commutative*, and *associative*. Therefore, if numerical computations to evaluate T-norm based expressions are carried out at run-time, the above properties ensure that any result can be directly computed from the individual value of each argument; that the result is independent from the order of the arguments; and that for more than two arguments, the evaluation of T-norm expressions can be done recursively (alternatively, the evaluation can be decomposed by subdividing the arguments into subgroups, performing each local evaluation independently, and aggregating the partial results).

In the third layer (the control layer) we define the functions required to select the calculus appropriate for each context and to resolve eventual ignorance or conflict in the information. These functions rely on *local* (i.e., context-dependent) knowledge about the information (meta-knowledge). The scope of the calculus selection and ignorance/conflict resolution is limited to the context (knowledge base subset) for which the meta-knowledge is available. Figure 1 illustrates RUM's architecture. The following sections describe RUM's functions attached to each of the three layers.

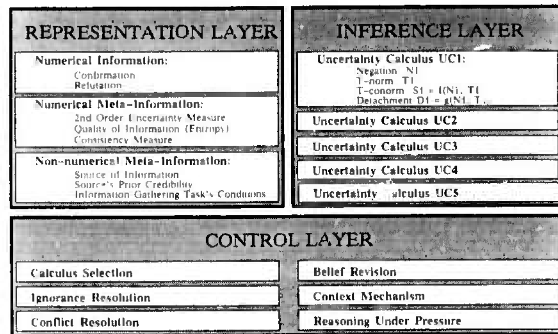


Figure 1. RUM's Three Layer Architecture

3.1 Representation: the Wff System and the Rule Language

The representation layer is based on frame-like data structures that capture the uncertainty information used in the inference layer and the uncertainty meta-information used in the control layer.

3.1.1 RUM's Wff System

RUM's Wff System modifies KEE's representation of a *wff* (well-formed formula). RUM's *wff* is the pair [<*unit*> <*slot*>], which is the description of a variable in the problem domain. For each *wff* a corresponding uncertainty unit is created. The unit contains a list of the values that were considered for the *wff*. For each value the unit maintains its certainty's *lower* and *upper bounds*, an *ignorance measure*, a *consistency measure*, and the *evidence source*.

Figure 2 illustrates an example of an uncertainty unit attached to a *wff*. The *wff* is the variable [*Platform-439 Class-name*]. In the uncertainty unit, under the slot VALUES, we can see the possible values which were considered by the system and their corresponding certainty bounds. The uncertainty unit also maintains a record of the rule instances which were fired to derive such values (for inferred *wffs*, this logical support represents the evidence source).

RUM's Wff System allows the user to express arbitrary uncertainty granularity by providing the flexibility to mix precise and imprecise measures of certainty in defining the input certainty (points, intervals, fuzzy numbers/intervals, linguistic values) and the rule strengths (categorical and plausible IF/IFF). Various term sets of linguistic probabilities with

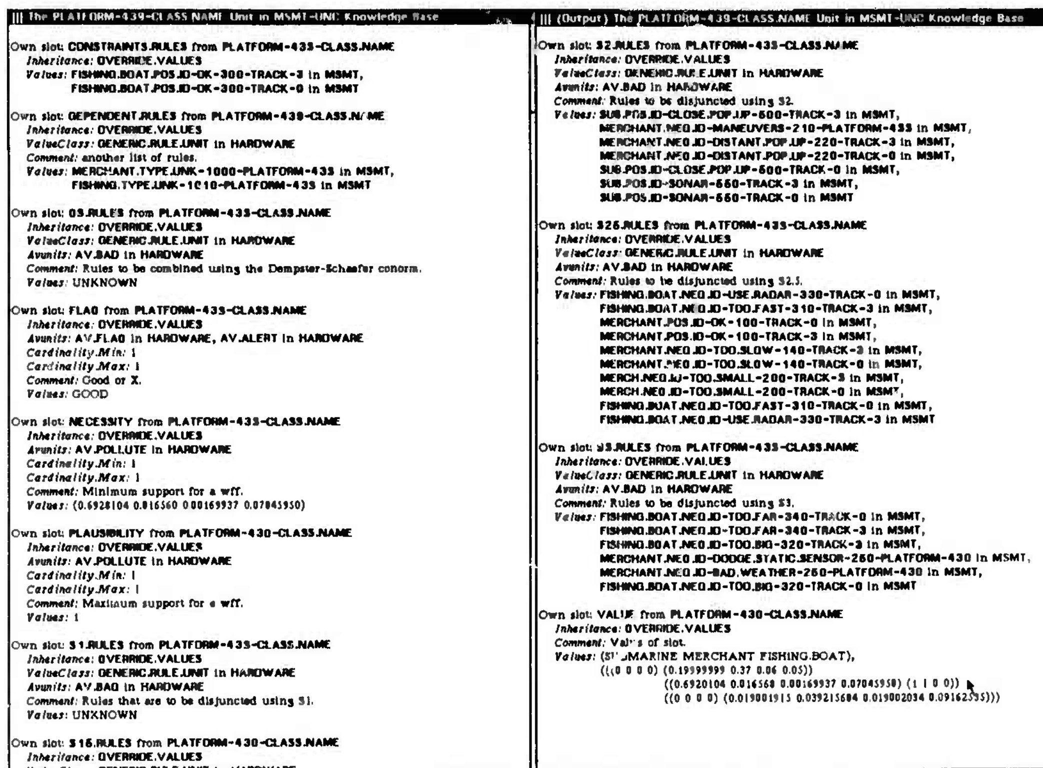


Figure 2. Uncertainty Unit Associated with wff [Platform-439 Class-name]

fuzzy-valued semantics [Beyth-Marom 1982, Bonissone 1985] provide a selection of input granularity. The values of the terms can be used as default values or can be modified by the user.

3.1.2 RUM's Rule System: The Rule Language

RUM's Rule System replaces KEE Rule System-3 capabilities by incorporating uncertainty information in the inference scheme. The uncertain information is described in the uncertainty units of the *wffs*, represented in RUM's Wff System, and in the degrees of sufficiency and necessity attached to each rule.* The degree of sufficiency denotes the extent to which one should believe in the rule conclusion, if the rule premise is satisfied. The degree of necessity indicates the confidence with which one can negate the conclusion, if the premise fails.

A rule is internally represented by a frame with several slots. These slots include the name of the rule; the lists of contexts, premises, and conclusions; the rule's sufficiency and necessity; and the T-norm to be used for aggregation. All slots (except the name, premises, and consequences) have default values. The contexts, premises, and conclusions can comprise values, variables, RUM predicates and arbitrary LISP functions. Rules with unbound variables are instantiated with the necessary environment to produce rule instances. Figure 3 illustrates an example of the instantiation of Rule-550, internally represented as a frame. Rule-550 defines a relationship between the parameters obtained from a sensor report (sonar) and the value *Submarine* for the *wff* [*Platform-Class name*]. The same rule in its English and Lisp versions (the latter being the form in which the rule is originally written) is described in section 3.1 of the second paper included in Part 1.

The T-norm specified with each rule is used to aggregate the certainties of the rule premises and to perform detachment (which computes the certainty of the conclusion given the sufficiency and necessity of the rule). It defaults to T_3, which is the MIN function. The associated T-conorm is used to aggregate the certainties of identical conclusions inferred by multiple rule instances derived from the *same* rule. These are often subsumptive, and the value defaults to S_3, the MAX function. Finally, each separate consequence of a rule has a specified T-conorm that will be used to aggregate the consequence with identical consequences derived from *different* rules. (i.e., multiple assignments of the same value to the wff). The negation operator causes the wff to be assigned the complemented value.*

* It is important to note that the inference symbol $\rightarrow$ in the production rule $A \rightarrow^{s} B$ is interpreted as a (weak) *material implication* operator in multiple-valued logics. The value s is the lower bound of the degree of sufficiency of the implication. This is in contrast with the interpretation of *conditioning*, i.e., $s = P(B|A)$. The symbol $\leftrightarrow$ in the production rule $A \leftrightarrow^{s,n} B$ is interpreted as a (weak) *logical equivalence* operator in multiple-valued logics, in which s and n are the lower bounds of sufficiency and necessity, respectively. This (weak) logical equivalence is an *if-and-only-if* (IFF) rule, which can be decomposed into the two rules $A \rightarrow^{s} B$ and $B \rightarrow^{n} A$ (equivalent to $\neg A \rightarrow^{n} \neg B$). RUM's rules are of the type $C \rightarrow (A \leftrightarrow^{s,n} B)$, where C indicates the context of the rule (see section 2.3.3) and $\rightarrow$ represents the strong material implication.

* If a *wff* has a value A with an II the certainty interval attached to a value A is [L(A), U(A)], its complemented value, $\neg$A, has a certainty interval defined by [1-U(A), 1-L(A)].

```
The SUB.POS.ID-SONAR-550-TRACK-3 Unit in MSM1 Knowledge Base

Own slot: CONSEQUENCES from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: UNIQUE.VALUES
   Avunits: AV.CONSQ.UPDATE in HARDWARE
   Comment: Results.
   Values: ((GET.VALUE TRACK-3 PLATFORM) CLASS.NAME SUBMARINE S2.RULES)

Own slot: CONTEXTS from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: UNIQUE.VALUES
   Avunits: AV.BAD in HARDWARE
   Comment: List of contexts to trigger the rule.
   Values: (IS-IN-CLASS? 'TRACK-3 'SOURCE '(SONAR LOTTA))

Own slot: FLAG from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: OVERRIDE.VALUES
   Avunits: AV.FLAG in HARDWARE, AV.ALERT in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Good or X.
   Values: NA

Own slot: NECESSITY from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: UNIQUE.VALUES
   Avunits: AV.RULE.EVAL in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Minimum proof.
   Values: UNKNOWN

Own slot: PLAUSIBILITY from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: UNIQUE.VALUES
   Avunits: AV.RULE.EVAL in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Maximum proof.
   Values: UNKNOWN

Own slot: PREMISE.SLOTS from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: UNIQUE.VALUES
   Avunits: AV.STILTIFY in HARDWARE
   Comment: Slots which affect this rule.
   Values: TRACK-3-LAST.REPORT in MSMT-UNC, TRACK-3-SOURCE in MSMT-UNC

Own slot: PREMISES from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: UNIQUE.VALUES
   Avunits: AV.BAD in HARDWARE
   Comment: List of premises.
   Values: (IS-VALUE? (GET.VALUE 'TRACK-3 'LAST.REPORT) 'NOISE-EMISSIONS 'LOW),
           (U-LESSP (GET.UNCERTAIN.VALUE (GET.VALUE 'TRACK-3 'LAST.REPORT) 'ELE
           VATION)
                        (FUZZ -20))

Own slot: RULE.NECESSITY from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: OVERRIDE.VALUES
```

```
(Output) The SUB.POS.ID-SONAR-550-TRACK-3 Unit in MSM1 Knowledge Base

Own slot: RULE.NECESSITY from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: OVERRIDE.VALUES
   Avunits: AV.BAD in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Lower bound of Q => P.
   Values: IT.MAY

Own slot: RULE.SUFFICIENCY from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: OVERRIDE.VALUES
   Avunits: AV.BAD in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Lower bound of P => Q.
   Values: EXTREMELY.LIKELY

Own slot: SUB.TASK from GENERIC.RULE.UNIT
   Inheritance: METHOD
   ValueClass: METHOD
   Comment: LISP task performed when the sub threshold is passed
   Values: HARDWARE>GENERIC.RULE.UNIT:SUB.TASK!method

Own slot: SUB.THRESHOLD from GENERIC.RULE.UNIT
   Inheritance: OVERRIDE.VALUES
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Threshold to trigger the sub.task.
   Values: 0

Own slot: SUPER.TASK from GENERIC.RULE.UNIT
   Inheritance: METHOD
   ValueClass: METHOD
   Comment: LISP task to perform should the necessity pass the super.threshold.
   Values: HARDWARE>GENERIC.RULE.UNIT:SUPER.TASK!method

Own slot: SUPER.THRESHOLD from GENERIC.RULE.UNIT
   Inheritance: OVERRIDE.VALUES
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Threshold to trigger the super task.
   Values: 1

Own slot: T.NORM from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: OVERRIDE.VALUES
   ValueClass: T.NORM.FAMILY in HARDWARE
   Avunits: AV.BAD in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: The t-norm.
   Values: T3 in HARDWARE

Own slot: TEMP.CONSQ.LIST from SUB.POS.ID-SONAR-550-TRACK-3
   Inheritance: OVERRIDE.VALUES
```

Figure 3. Internal Representation of an Instance of Rule 550

3.2 Interference: Triangular norms (T-norms) Based Calculi

The inference layer is built on a set of five Triangular norms (T-norms) based calculi The T-norms' associativity and truth functionality entail problem decomposition and relatively inexpensive belief revision. The theory of T-norms has been covered in previous articles [Bonissone 1985; 1986].

3.2.1 Operations in a T-norm Based Calculus

For each calculus, four operations are defined in RUM's Rule System: *premise evaluation, conclusion detachment, conclusion aggregation*, and *source consensus*. Each operation in a calculus can be completely defined by a Triangular norm $T(.,.)$, and a negation operator $N(.)$ (just as in classical logic, any boolean expression can be rewritten in terms of an intersection and complementation operator). The four operations are defined as follows:

Premise evaluation: The premise evaluation operation determines the degree to which all the clauses in the rule premise have been satisfied by the matching *wffs*. Let b_i and B_i indicate the lower and upper bounds of the certainty of condition i in the premise of a given rule. Then the premise certainty range [b,B] is defined as:

$$[b,B] = [\ T(b_1, b_2, \ldots, b_m),\ T(B_1, B_2, \ldots, B_m)\ ]$$

Conclusion Detachment: The conclusion detachment operation indicates the certainty with which the conclusion can be asserted, given the strength and appropriateness of the rule. Let s and n be the lower bounds of the degree of *sufficiency* and *necessity*, respectively, of the given rule, and let $[b,B]$ be the computed premise certainty range. Then the range $[c,C]$, indicating the lower and upper bound for the certainty of the conclusion inferred by such rule, is defined as:

$$[c,C] = [\ T(s,\ b),\ N(\ T(n,\ N(B)))\ ]$$

The degrees of sufficiency and necessity respectively indicate the amount of certainty with which the rule premise implies its conclusion and viceversa. The sufficiency degree is used with *modus ponens* to provide a lower bound of the conclusion. The necessity degree is used with *modus tollens* to obtain a lower bound for the complement of the conclusion (which can be transformed into an upper bound for the conclusion itself).

Conclusion aggregation: The conclusion aggregation operation determines the consolidated degree to which the conclusion is believed if supported by more than one path in the rule deduction graph, i.e., by more than one rule instance. It is also possible to have various groups of deductive paths, i.e. various sets of rule instances, all supporting the same conclusion. Each group of deductive paths can have a distinct conclusion aggregation operator associated with it. Let the ranges $[c_i, C_i]$ indicate the certainty lower and upper bounds of the *same* conclusion inferred by various rules instances belonging to the same group. Then, for each group of deductive paths, the range $[d,D]$ of the aggregated conclusion is defined as:

$$[d,D] = [N(T(N(c_1),\ N(c_2,\ \ldots\ ,N(c_m)),\ T(N(C_1),\ \ldots\ ,N(C_m)))\ ]$$

RUM distinguishes between rule instances generated from the same rule and rule instances derived from different rules. The first type or rule instances is aggregated first, to take into account the usually large amount of redundancy that such rule instances entail. The second set of rule instances is subsequently aggregated taking into account the knowledge about the presence or lack of positive/negative correlation that characterizes the various rules.

Source Consensus: The source consensus operation reflects the fusion of the certainty measures of the same evidence A provided by different sources. The evidence can be an *observed* fact, or a *deduced* fact. In the former case, the fusion occurs before the evidence is used as an input in the deduction process. In the latter case, the fusion occurs after the evidence has been aggregated by each group of deductive paths. The source consensus operation reduces the ignorance about the certainty of A, by producing an interval that is always smaller or equal to the smallest interval provided by any of the information source. If there is an inconsistency among some of the sources, the resulting certainty intervals will be disjoint, thus introducing a conflict in the aggregated result. Let $[L_1(A), U_1(A)]$, $[L_2(A), U_2(A)]$, ..., $[L_n(A), U_n(A)]$ be the certainty lower and upper bounds of the same conclusion provided by different sources of information. Then, the result $[L_{tot}(A), U_{tot}(A)]$, obtained from *fusing* all the assertions about A, is given by taking the intersection of the certainty intervals:

$$[L_{tot}(A), U_{tot}(A)] = [\ Max_i\ L_i(A),\ Min_i\ U_i(A)]$$

3.3 Control: Calculus selection, Uncertain-Belief Revision, Context Mechanism

3.3.1 Calculi Selection

As it was discussed in the previous section, RUM's Rule System uses a set of five T-norm based calculi. The calculus used by each rule instance is inherited from its rule subclass (the rule before the instantiation). The calculus can be modified through KEE's user interface or programmatically (i.e., by an active value). Class inheritance can also be used to modify the degree of sufficiency and necessity of all the rule members of the same class.

The calculi selection consists of two assignments. The first assignment indicates the T-norm with which the premise evaluation and the conclusion detachment will be computed. Such an assignment is made for each rule, and, through inheritance, is passed to all rule instances derived from the same rule.

The second assignment indicates the T-conorm (represented by its dual T-norm) with which the conclusion aggregation will be computed. This assignment is made for each subset of rule instances generated from *different* rules and asserting the same conclusion.

3.3.1.1 Rationale for Calculi Selection

The T-norm characteristics will determine the selection choices. For the first assignment, the T-norm assigned to each rule for the premise evaluation and the conclusion detachment will be a function of the decision maker's *attitude toward risk*. The ordering of the T-norms, which is identical to the ordering of parameter p in the Schweizer & Sklar family of T-norms, reflects the ordering from a conservative attitude ($p = -1$ or t_1) to a non-conservative one ($p \to \infty$ or T_3). From the definition of the calculi operations, we can see that T_1 will generate the smallest premise evaluation and the weakest conclusion detachment (i.e., the widest uncertainty interval attached to the rule's conclusion). T-norms generated by larger values of p will exhibit less drastic behaviors and will produce nested intervals with their detachment operations. T_3 will generate the largest premise evaluation and the strongest conclusion detachment (the smallest certainty interval).

For the second assignment, the T-norm assigned to the subsets of rule instances (derived from different rules and asserting the same conclusion) will be a function of the *lack or presence of positive/negative correlation* among the rules in each subset. The ordering of the T-norms reflects the transition from the case of extreme negative correlation, i.e., mutual exclusiveness (T_1), through the case of uncorrelation (T_2), to the case of extreme positive correlation, i.e., subsumption (T_3).

Currently, all calculi assignments are explicitly made and modified through the user interface, to exercise the implemented accessing functions. In the next development phase of RUM control layer, the calculi assignments will be made by a set of selection rules expressing the meta-knowledge about the context. These rules will select the T-norms that better reflect the knowledge engineer's desired attitude toward risk and the perceived amount of correlation among the rules used in such a context.

3.3.2 Uncertain-Belief Revision

A daemon-based implementation of the belief revision of the uncertain information is available in the control layer of RUM's Rule System. For any conclusion made by a rule, the belief revision mechanism monitors the changes in the certainty measures of the *wffs* that constitute the conclusion's support or the changes in the calculus used to compute the conclusion certainty measure. Validity flags are inexpensively propagated through the rule deduction graph. Five types of flag values are used:

Good Guarantees the validity of the cached certainty measure detached by the rule instance and aggregated into the associated wff.

Bad (level i) Indicates that the cached certainty measure detached by the rule instance is no longer reliable, since the support of some of the wff's in the premise of this rule instance has changed. The *i*th level indicates the correct order of recomputation.

Inconsistent Indicates that the cached certainty measure associated with the wff is conflicting. The inconsistency can be removed by executing a locally defined procedure (differential diagnosis type of experiment, recency of information, split in possible words with subsets of the original sources, etc.)

Not Applicable Indicates that the context of the rule instance is no longer active and the rule instance con-

tribution to the aggregated certainty measure of the wff should be ignored.

Ignorant Indicates that the cached certainty measure detached by the rule instance is too vague to be useful. The default behavior is to ignore the rule instance contribution to the aggregated certainty measure of the wff. Locally defined procedure could be used to remove the ignorance if so specified.

3.3.2.1 An Example of Using the Uncertain-Belief Revision

To provide the reader with a better understanding of the uncertain-belief revision, we will make the following graphical analogy: the *wffs* of the reasoning system correspond to nodes in an acyclic deductive graph; the inference rules in the system correspond to the inference gates that connect the nodes in the graph. There are two types of *wffs*: the observations or assumptions, corresponding to the nodes at the frontier of the graph, and the inferred conclusions, corresponding to the *intermediate* nodes in the graph. The first type of node does not have any logical support (its evidence source is the observer or the assumption's maker). The second type of node has a logical support represented by the set of rule instances that made that inference. For this second type of nodes, this logical support is the evidence source.

Figure 4 illustrates a a portion of an acyclic deductive graph. In the graph we can observe the following five rules:

R1: C → (A,B → J) suffic. = s_1 necess. = n_1 calcul. = T_2 aggreg. = S_2

R2: C → (D → J) suffic. = s_2 necess. = n_2 calcul. = T_2 aggreg. = S_2

R3: (E → J) suffic. = s_3 necess. = n_3 calcul. = T_3 aggreg. = S_3

R4: H → (E,F,G → J) suffic. = s_4 necess. = n_4 calcul. = T_3 aggreg. = S_3

R5: (J,I → J) suffic. = s_4 necess. = n_4 calcul. = T_3 aggreg. = S_2

Two more rules, R6 and R7, are partially shown in the same figure.*

* The following notation is used in the rule description and in the figure:

"." indicates intersection, (input of the same gate)

" →" indicates (strong) material implication, (control line on the side of the gate)

"⟷" indicates (weak) logical equivalence, i.e., if-and-only-if rule (gate)

"s_i" indicates the lower bound of sufficiency of rule i

"n_i" indicates the lower bound of necessity of rule i

"$[T_i]$" indicates the calculus (T-norm) used by the rule to perform premise aggregation and conclusion detachment.

"S_j" Suffix j takes one of the following values: {j = 1, 1.5, 2, 2.5, 3} indicates the calculus (T-conorm) used to perform the conclusion aggregation.

"L_j, $U3_j$" Suffix j takes one of the following values: {j = 1, 1.5, 2, 2.5, 3} indicate the lower and upper bounds of the conclusion detached from rule i

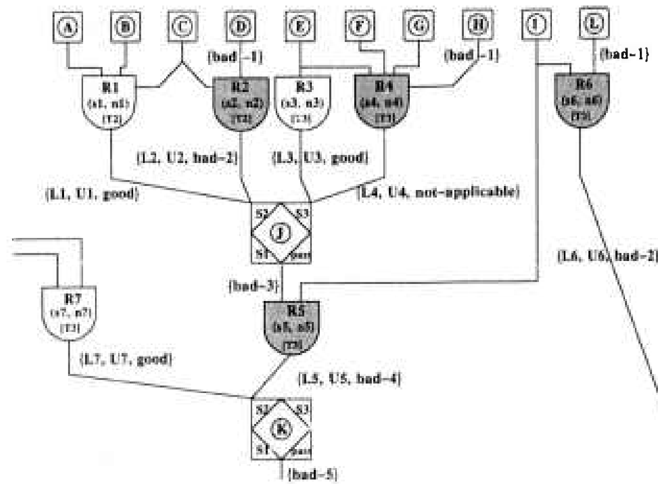


Figure 4. Portion of an Acyclic Deductive Graph

In Figure 4, C and H represent two context descriptions that enable/disable the activation of rules R1, R2, R4. The other two rules (R3 and R5) are always potentially active (regardless of context). The figure shows the case in which fact D has just changed. This change causes the propagation of a bad-validity flag that affects the conclusion of rules R2 and R5 (J and K, respectively). The numbers attached to the bad flag indicate the order in which a recomputation of the certainty measures must be performed. Fact H has also changed and its new value no longer satisfies the context description of rule R4, thus causing the not-applicable flag to be attached to the detachment of R4. Fact L has also changed, affecting the validity of Rule R6's detachment.

3.3.2.2 Reasoning under Pressure

The belief revision system offers both backward and forward processing. A lazy evaluation, running in *depth-first, backward mode*, recomputes the certainty measures of the modified *wff* that are required to answer a given query. This mode (called reasoning under pressure) is used when the system or the user decide that they are dealing with time-critical tasks. In the case illustrated in the previous figure, if the value of *wff* K were requested, the systems would perform the following sequence of tasks: fetch the new certainty values of D (lower and upper bounds); recompute the detachment of rule R2; use T-conorm S_2 to evaluate the OR node (with R1 and R2's detachments); ignore R4's detachment, treating R3's detachment as the only input to the OR node associated with T-conorm S_3; fuse the two OR nodes, defining the new certainty values of *wff* J; recompute the detachment of rule R5; use T-conorm S_2 to evaluate the OR node (with R5 and R7's detachments), obtaining the new certainty values of *wff* K.

When time is not critical, the system can use a *breadth-first, forward mode* processing to recompute the certainty measures of the modified *wffs*, attempting to restore the integrity of the rule deduction graph. In the case illustrated in the previous figure, this implies an update of fact L and rule R6 (both of which were not considered by the backward mode, since they did not play any role in determining the value of the proposed query, e.g. *wff* K).

The structure of the graph can also change, as new rule instances are created or deleted, due to changes in the facts' values, (as opposite to facts' certainty values). The deduction graph is updated and bad flags are propagated throughout the network

3.3.3 Rule Firing Control via Context Activation

A user-definable threshold can be attached to each rule context, either by local definition or by inheritance from a rule class. A rule context is defined as a conjunction of conditions that must be satisfied before the rule can be considered for premise evaluation. Each condition is described by a predicate on object-level *wffs* (facts in problem domain), or control-level *wffs* (markers asserted by meta-rules). The semantics of a context C attached to an inference rule (establishing the weak logical equivalence between A and B) is given by the following expression:

$$\mathbf{C} \rightarrow (\mathbf{A} \leftrightarrow^{s,n} \mathbf{B})$$

where s and n indicate the lower bounds of the degree of sufficiency and necessity that the rule provides; $\rightarrow$ represents the strong material implication; $\leftrightarrow$ denotes the weak logical equivalence.

The context mechanism provides the following features:

1. By activating/deactivating subsets of the KB, it limits the number of rules that will be considered relevant at any given time, thus increasing the overall system efficiency.
2. By only considering the rules relevant to a given situation, it allows the knowledge engineer to effectively use the necessary conditions in the rule's premise. It is now possible to distinguish between the failure of a necessary test (described in the premise) and the failure of the rule's applicability (traditionally described by other clauses in the same premise and now explicitly represented in the context).
3. By using predicates on the control-level *wffs*, it provides the required programmability for defining flexible control strategies, such as causing sequences of rules to be co-executed, firing default rules, ordering and handling ti[illegible] dependent information, etc.
4. By using hierarchical contexts, it can be used as an organizing principle for the knowledge acquisition task.

4. Remarks and Conclusions

RUM's layered architecture properly addresses the requirements defined by the desiderata [Bonissone 1986] for *uncertain*-reasoning systems. The representation layer captures the uncertain information about the *wffs* (lower and upper bounds) used by the calculi in the inference layer to determine the uncertainty of the conclusions. The representation layer also captures the uncertain meta-information (evidence source or logical support, measures of ignorance and conflict) used by the belief revision system and other mechanisms in the control layer.

The inference layer provides the knowledge engineer with a rich selection of well-understood calculi to properly represent existing correlations among rules. Numerical computations performed in this layer are efficiently implemented by using a four parameter representation for the uncertainty bounds, supported by a set of closed form formulae that implement the truth functional uncertainty calculi [Bonissone 1985].

The control layer provides the explicit selection and modification of uncertainty calculi. Its context activation mechanism allows the reasoning system to focus on the relevant subsets of the changing inference base (the acyclic deductive graph). The uncertain-belief revision maintains the integrity of those relevant subsets, reflecting the changes of the information.

RUM's development environment provides the traceability of *wffs* and rules that is required for proper KB development and refinment. An example of such a KB develeopment is presented in the next paper, which describes the experiments used to validate RUM as a reasoning tool applied to a naval situation assessment task.

5. References

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

[Bonissone 1980] Bonissone, P.P., "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 M.M. Gupta, E. Sanchez, Eds.) *Approximate Reasoning in Decision Analysis*,, pp. 329-339, North Holland Publishing Co., New York, 1982.

[Bonissone 1985] Bonissone, P.P. & Decker, K.S., "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. Also in L.Kanal & J.Lemmer (Eds.) *Uncertainty in Artificial Intelligence*, pp. 217-247, North-Holland, 1986.

[Bonissone 1986] Bonissone, P.P., "Summarizing and Propagating Uncertain Information with Triangular Norms", in Lee S. Baumann, (Ed.) *Proceedings of the Expert Systems Workshop*, pp 62-71, Defense Advanced Research Projects Agency, Science Applications International Corporation, Pacific Grove, California, April 1986. To appear in the *International Journal of Approximate Reasoning*, North-Holland Publishing Company, 1987.

[Bonissone 1987a] Bonissone, P.P., "Plausible Reasoning: Coping with Uncertainty in Expert Systems", in S. Shapiro (Ed.) *The Encyclopedia of Artificial Intelligence*, John Wiley and Sons, to appear in April 1987.

[Cohen 1983a] P.R. Cohen, M.R. Grinberg, A Theory of Heuristics Reasoning about Uncertainty, *The AI Magazine*, Vol. 4, No. 2, pp. 17-23, (1983).

[Cohen 1983b] P.R. Cohen, M.R. Grinberg, "A Framework for Heuristics Reasoning about Uncertainty", *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83)*, pp. 355-357, Karlsruhe, West Germany, 1983.

[Doyle 1983] J. Doyle, "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).

[Genesereth 1982] M.R. Genesereth, "An Overview of MRS for AI Experts", Stanford Heuristic Programming Project Memo HPP-82-27, Dept. of Computer Science, Stanford University, (1982).

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

[Shafer 1976] G. Shafer, *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, NJ, (1976).

Progress in Reasoning with Incomplete and Uncertain Information*

Part I: Reasoning with Uncertainty
Part II: Analogical Reasoning
Part III: Reasoning with Incomplete Information

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

BONISSONE@GE-CRD.ARPA, PORTER@GE-CRD.ARPA,
BROWNAL@GE-CRD.ARPA

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 *Reasoning with Uncertainty*, summarizes the papers *RUM: a Layered Architecture for Reasoning with Uncertainty*, [Bonissone, Gans, and Decker, 1987] and *Using T-norm Based Uncertainty Calculi in a Naval Situation Assessment Application* [Bonissone 1987]. This first paper describes an integrated software tool that implements the three layer architecture concept described in our previous progress report. This software tool is based on KEE^{TM} (Knowledge Engineering Environment), an expert system shell implemented in an object oriented language.** The second paper illustrates an application of RUM in solving a multi/sensor multi target problem developed in LOTTA, an object-based simulation environment.

Part II, contains entitled Reasoning by Analogy summarizes the paper *A Mathematical Theory for Diagnosis Based on the MONAD Concept* [Porter87]. This paper describes the model-based knowledge representation and search strategy used to form problem models in the MONAD system.

Part III, entitled *Reasoning with Incomplete Information* summarizes Three papers *An Algebraic Foundation for Truth Maintenance* [Brown, Benanav, and Gaucas, 1987], *Logics of Justified Beliefs* [Brown, 1987], and *A Role for Assumption-based and Non-monotonic Justifications in Automating Strategic Threat Analysis* [Gaucas and Brown, 1987]. The first paper presents a reason maintenance system in which assumption-based justifications (ATMS) and non-monotonic justifications can be directly and transparently described. The second paper provides formal semantics to truth maintenance by offering a mathematical logic — equipped with an underlying model theory — that is used to characterize well known models of truth maintenance. The third paper describes an experiment in using an assumption-based and non-monotonic reasoning capability in support of strategic analysis.

* This work was partially supported by the Defense Advanced Research Projects Agency (DARPA) under USAF/Rome Air Development Center 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.

** The implementation of RUM, Reasoning with Uncertainty Module, was not part of (nor was it funded by) DARPA contract F30602-85-C-0033. This description is included only for the purpose of illustrating the technology integration and transition efforts that GE, within the spirit of the Strategic Computing Initiative, has undertaken outside of its contractual obligations.

Part I: Reasoning with Uncertainty

TABLE OF CONTENTS

RUM: a Layered Architecture for Reasoning with Uncertainty

1. Introduction
2. RUM's Underlying Theory
   2.1 Term Sets of Linguistic Probabilities
   2.2 T-norms: Definition and Equivalence Classes
3. Design of RUM 's Layered Architecture
   3.1 RUM Representation Layer: the Wff System and the Rule Language
      3.1.1 RUM's Wff System
      3.1.2 RUM's Rule System
   3.2 RUM Inference Layer: Triangular Norms (T-Norms) Based Calculi
      3.2.1 T-norm based Calculi Operations
   3.3 RUM Control Layer: Calculus Selection, Uncertain Belief Revision, Context Mechanism
      3.3.1 Calculi Selection
         3.3.1.1 Rationale for Calculi Selection
      3.3.2 Uncertain-Belief Revision
         3.3.2.1 An Example of Using the Uncertain Belief
         3.3.2.2 Reasoning under Pressure
      3.3.3 Rule Firing via Context Activation
4. Conclusions
5. References

Using T-norm Based Uncertainty Calculi in a Naval Situation Assessment Application

1. Introduction
2. An Object Based Simulation Environment
3. The Information Fusion/Situation Assessment Problem
   3.1 Example of RUM rules
   3.2 Notes on the Calculi Selection for Rule 500 and 550
4. The Experiment
5. Remarks and Conclusions
6. References

Part II: Analogical Reasoning

TABLE OF CONTENTS

A Mathematical Theory for Diagnosis Based on the MONAD Concept

1. Introduction
2. The Diagnosis Process
   2.1 Reiter's Approaches
   2.2 The Source of the Problem
3. Diagnosis Based on Variational Analysis
   3.1 Relation Method
4. The Full Adder Example
5. An Abstract Algebra for Logic
   5.1 The Approach to Diagnosis
   5.2 Diagnostic Equations
6. Applying Specific Failure Models
7. An Abstract Example
   7.1 Diagnosis by Dependency
   7.2 Diagnosis Using an Abstract Model
8. Diagnosis with Incomplete Information
9. Conclusions
10. References

Part III: Reasoning with Incomplete Information

TABLE OF CONTENTS

An Algebraic Foundation for Truth Maintenance

1. Introduction
2. Lattice Equational Systems
3. The Existence of Solutions
4. The Structure of Systems and Solutions
5. Conclusions
6. References

Logics of Justified Beliefs

1. Introduction
2. Non-Monotonic Truth Maintenance
   2.1 Syntax
   2.2 Semantics
3. Assumption-Based Truth Maintenance
   3.1 ATMS
   3.2 ANRMS
4. Conclusions
5. References

A Role for Assumption-based and Non-monotonic Justifications in Automating Strategic Threat Analysis

1. Introduction
2. Clarkson's View of Strategic Analysis
3. Representing Threat Situations
   3.1 A Threat Scenario
   3.2 A Formal Model for Situation Assessment
   3.3 An Encoding of a Threat Scenario
4. Reasoning About Threat Situations
   4.1 A Situation Assessment
   4.2 A Problem Solving Architecture for Situation Assessment
   4.3 An Instance of a Problem Solver
   4.4 A Problem Solver's Reasoning Sequence
5. Conclusions
6. References

REFERENCES

[Bonissone et al., 1987] P.P. Bonissone, S.S. Gans, K.S. Decker, "RUM: a Layered Architecture for Reasoning with Uncertainty", to appear in the *Proceedings of the 1987 International Joint Conference of Artificial Intelligence (IJCAI 87)*, Milano, Italy, August 1987.

[Bonissone 1987] P.P. Bonissone, "Using T-norm Based Uncertainty Calculi in a Naval Situation Assessment Application", manuscript submitted for publications in the *Proceedings of the the Third AAAI Workshop on Uncertainty in Artificial Intelligence*, Seattle, Washington, July 1987.

[Brown, et al., 1987] A.L. Brown, D. Benanav, and D.E. Gaucas, "An Algebraic Foundation for Truth Maintenance", to appear in the *Proceedings of the 1987 International Joint Conference of Artificial Intelligence (IJCAI 87)*, Milano, Italy, August 1987.

[Brown 1987] A.L. Brown, "Logics of Justified Beliefs", working paper, Artificial Intelligence Branch, General Electric Corporate Research and Development, Schenectady, New York, 1987.

[Gaucas and Brown, 1987] D.E. Gaucas and A.L. Brown, "A Role for Assumption-based and Nonmonotonic Justifications in Automating Strategic Threat Analysis", working paper, Artificial Intelligence Branch, General Electric Corporate Research and Development, Schenectady, New York, 1987.

[Porter 1987] G.B. Porter, "A Mathematical Theory for Diagnosis Based on the MONAD Concept" working paper, Artificial Intelligence Branch, General Electric Corporate Research and Development, Schenectady, New York, 1987.

An Algebraic Foundation for Truth Maintenance

Allen L. Brown, Jr., Dale E. Gaucas, and Dan Benanav
GE Corporate Research and Development Center
P.O. Box 8
Schenectady, New York 12301

Abstract

We have recast the problem of truth maintenance in a setting of algebraic equations over Boolean lattices. If a method of labeling propositions to justify them according to some reasoning agent's constraints of belief happens to conform to the postulates of Boolean lattices, the labeling system can be reformulated as an algebraic equation solving system. All truth maintenance systems known to us can be so reformulated. This note summarizes our investigations into the existence and structure of solutions of these algebraic systems. Our central result is a unique factorization theorem for lattice equational systems and their solutions. Our theoretical results are interpreted to compare various styles of truth maintenance and to reveal certain computational difficulties implicit in the algebraic structure of truth maintenance.

I. Introduction

Lattice-theoretic truth maintenance is a single theoretical framework that subsumes various notions of truth maintenance, including the assumption-based justifications reported by de Kleer [de Kleer, 1984, de Kleer, 1986a, de Kleer, 1986b, de Kleer, 1986c] and the nonmonotonic justifications reported by Doyle [Doyle, 1979a, Doyle, 1979b, Doyle, 1978] and Goodwin [Goodwin, 1982, Goodwin, 1985, Goodwin, 1984, Goodwin, 1987]. Our complete body of work on lattice-theoretic truth maintenance includes

- An analysis of the algebraic structure of truth maintenance
- An investigation of the abstract and concrete computational complexity of truth maintenance
- A formal account of the embedding of other forms of truth maintenance in the lattice-theoretic paradigm

In this note we focus on the first aspect, because of its intrinsic interest, and because this aspect is a precursor to the others. Our express aim here is to present the lattice-theoretic account of truth maintenance, cite the more important *algebraic* results *vis à vis* this account, and interpret these results so as to cast a qualitative light on various computational considerations of truth maintenance. Readers interested in other aspects of our theoretical work or our practical experience with an implementation embodying this theory are referred to [Benanav *et al.*, 1986].

The initial motivation for this work was the desire to unify in a single abstraction the truth maintenance paradigm of Doyle and Goodwin, and that of de Kleer. The systems of these investigators can be viewed as constraint propagation mechanisms. Given a disjunctive set 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. Doyle's and Goodwin's TMS's, on the other hand, tell 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 nonmonotonic in nature.* Our original intuition was that it should be possible to account *simultaneously* for multiple sets of premises *and* nonmonotonic deductive constraints.**,***

This intuition arose from the striking similarity observed between the computations of truth maintenance systems and the computations of global flow analysis that underly modern optimizing compilers [Aho and Ullman, 1977, Hecht, 1977, Schaeffer, 1973, Waite and Goos, 1984]. 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 pro-

*A monotonic deductive constraint obliges a rational agent to believe its consequent, given that it currently believes all of its antecedents. A *nonmonotonic* deductive constraint obliges a rational agent to believe its consequent given that it believes all of its monotonic antecedents and none of its nonmonotonic antecedents.

**The intellectual challenge of unifying these two approaches to truth maintenance is sufficient motivation for proceeding. Nonetheless, we note that de Kleer [de Kleer, 1986b], and Morris and Nado [Morris and Nado, 1986] are practically motivated to augment their assumption-based truth maintenance systems to support some form of nonmonotonic justification. In our approach nonmonotonicity will be "built-in" rather than "added-on". Although we will not do so here, it can be shown that our conceptually parsimonious approach is at a computational advantage relative to the attempts of de Kleer, and Morris and Nado.

***We have recently been made aware of the work of McDermott [McDermott, 1983] whose perspective on truth maintenance has much in common with our own. Indeed, his concrete solution to what we will eventually define as *even equational systems* appears to be identical to ours, though arrived at from a quite different point of departure. Our investigation is broader in both the scope of equational systems investigated, and in the characterization of those systems' structures and solution spaces.

grams) obliged to 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 truth 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 nonmonotonic constraints) the solution methods used in global flow analysis are inappropriate. A rather different solution method has been developed.

II. Lattice Equational Systems

Let $\mathcal{B}$ be a Boolean lattice equipped with the usual meet, join, and complementation operators; a partial order, $<$; and maximum and minimum elements, $\top$ and $\bot$, respectively.* A complete account of such structures can be found in any of [Balbes and Dwinger, 1974, Birkhoff, 1967, Skornjakov, 1977]. Elements of $\mathcal{B}$ will be called *situations*, and will be denoted by A and B. A and B (possibly subscripted) are *lattice expressions* in $\mathcal{B}$. Moreover, if A and B are expressions in $\mathcal{B}$ then so are $A \vee B$, $A \wedge B$, $\overline{A}$ and $\overline{B}$. 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. Each lattice expression in $\mathcal{B}$ and unknown is a *lattice form* in $\mathcal{B}$. Moreover, if X and Y are forms in $\mathcal{B}$ then so are $X \vee Y$, $X \wedge Y$, $\overline{X}$ and $\overline{Y}$. Individual (fixed) lattice forms in $\mathcal{B}$ will be denoted by X and Y, possibly subscripted. Every fact or proposition has an associated unknown. Note that a proposition and its negation have distinct associated unknowns. Indeed, an unknown corresponds exactly to a *node* as that term is used by Doyle, Goodwin, and de Kleer. A *lattice equation over* $\mathcal{B}$ is a relation of the form $X = Y$ where X is a lattice unknown and Y is a lattice form. A *lattice equational system over* $\mathcal{B}$, Σ, is any collection of lattice equations over $\mathcal{B}$ 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.

Σ will be sub- or superscripted when it is useful to distinguish among various equational systems. Unless the context is ambiguous, 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 entails belief in others. If Σ is a lattice equational system such that the right-hand side of each equality is of the form $\bigvee_i \bigwedge_j X_{ij}$ where each X_{ij} is an element of $\mathcal{B}$ or an unknown (possibly complemented), then Σ is said to be in ***disjunctive normal form.**** Since we can transform any form into disjunctive normal form, we will usually treat forms over $\mathcal{B}$ and lattice equational systems as if they were in disjunctive normal form.

A ***solution*** to a lattice equational system, Σ, is a function, Γ, from the lattice unknowns into $\mathcal{B}$ such that if for each equation in the system, each unknown s in the equation is replaced by $\Gamma(s)$ the equation holds in $\mathcal{B}$. Moreover, Γ takes any unknown, s, not on the left-hand-side of some equation in Σ into $\bot$, and in that regard the system Σ implicitly has the equation $s = \bot$. We will interpret lattice equations as constraints. A solution, then, is a ***labeling*** of propositions with situations. In particular, the situations are those in which a reasoning agent is obliged to believe the correspondingly labeled proposition given acceptance of the constraints imposed by the system. We will often subscript 'Γ' with the name of the system of which it is a solution. 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 ***nonmonotonic*** or ***monotonic antecedent*** of d depending on whether or not it is complemented. The sets of monotonic and nonmonotonic antecedents of d are respectively denoted $\alpha(d)$ and $\bar{\alpha}(d)$. A justification, d, is ***valid*** with respect to a situation, A, and a solution, Γ, of an equational system Σ if and only if,

$$A \leq \bigwedge_{s \in \alpha(d)} \Gamma(s) \wedge \bigwedge_{s \in \bar{\alpha}(d)} \overline{\Gamma(s)}$$

We will write Valid(A,d,Γ) to indicate that d is valid with respect to A and solution Γ. A solution, Γ, is *well-founded with respect to a lattice equational system*, Σ, *at lattice unknown*, s, if and only if either $\Gamma(s) = \bot$, or $\Gamma(s) = \bigvee_i A_i$, and for each A_i, there is a partially ordered set, $\langle \mathcal{P}_{A_i}, \prec_{A_i} \rangle$, such that $\mathcal{P}_{A_i}$ is a set of justifications from Σ and

1. There is a justification, d in $\mathcal{P}_{A_i}$, whose consequent is s
2. For every justification d, in $\mathcal{P}_{A_i}$, Valid(A_i,d,Γ)
3. Every unknown, s', that is a monotonic antecedent of some d in $\mathcal{P}_{A_i}$ is also the consequent of some justification d' in $\mathcal{P}_{A_i}$ and $d' \prec_{A_i} d$

---

*In this report we assume $\mathcal{B}$ to be a recursive set, its operators to be total recursive functions, and its partial order to be a recursive relation.

*We use disjunctive normal form for notational convenience. While its existence is required in establishing some of the formal results that we cite, it plays no essential role in lattice-theoretic truth maintenance computations.

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 our arguments to be noncircular, we impose an additional condition, well-foundedness, to guarantee that state of affairs.

Let us first consider some uninterpreted equational systems, all taken to be over the Boolean lattice, $\mathcal{B}$, having at least two distinct elements: The system Σ_1

$$s = s$$

has one well-founded solution, $\{s = \bot\}$. On the other hand, any of $\{\{s = A\} | A \neq \bot \text{ and } A \in \mathcal{B}\}$ are also solutions, though not well-founded. The system Σ_2

$$s = \overline{s}$$

the classical "odd loop" of Doyle's TMS, has no solutions, well-founded or otherwise. The system Σ_3

$$\begin{aligned} s_1 &= s_2 \\ s_2 &= s_1 \end{aligned}$$

has one well-founded solution, $\{s_1 = \bot, s_2 = \bot\}$. The system Σ_4

$$\begin{aligned} s_1 &= \overline{s}_2 \\ s_2 &= \overline{s}_1 \end{aligned}$$

has well-founded solutions $\{\{s_1 = A, s_2 = \overline{A}\} | A \in \mathcal{B}\}$. The system Σ_5

$$\begin{aligned} s_1 &= \overline{s}_1 \vee s_2 \\ s_2 &= \overline{s}_2 \vee s_1 \end{aligned}$$

has a single solution $\{s_1 = \top, s_2 = \top\}$, and it is not well-founded. Finally, Σ_6

$$\begin{aligned} s_1 &= \overline{s}_2 \wedge \overline{s}_3 \\ s_2 &= \overline{s}_1 \wedge \overline{s}_3 \\ s_3 &= \overline{s}_1 \wedge \overline{s}_2 \end{aligned}$$

has well-founded solutions $\{\{s_1 = \bot, s_2 = A, s_3 = \overline{A}\} | A \in \mathcal{B}\} \cup \{\{s_1 = A, s_2 = \bot, s_3 = \overline{A}\} | A \in \mathcal{B}\} \cup \{\{s_1 = A, s_2 = \overline{A}, s_3 = \bot\} | A \in \mathcal{B}\}$. If $\mathcal{B} = \{\top, \bot\}$ and interpreting $\top$ as "IN" and $\bot$ as "OUT", it should be apparent to readers familiar with the TMS's of Doyle and Goodwin how lattice equational systems correspond to their TMS nodes and justifications.

The correspondence with de Kleer's ATMS is a little harder to convey, and we shall attempt only an approximation here.* We shall do this by actually interpreting a lattice equational system with respect to a toy application. Imagine a simple series-connected circuit consisting of a voltage source, V, of 5 volts connected to resistor R_1 at node n_1, which in turn is connected to resistor R_2 at node n_2, which is connected to ground. The application is a program that diagnoses ground faults in electrical circuits. In its truth maintenance database it has the following system of equations, Σ_7:

$$\begin{aligned} s_1 &= A \\ s_2 &= B_1 \\ s_3 &= B_2 \\ s_4 &= s_1 \wedge s_2 \wedge s_3 \end{aligned}$$

The situations A, B_1, and B_2 respectively correspond to the assumption that the voltage source, V, and resistors, R_1 and R_2, are working. s_1 corresponds to the proposition that voltage at node n_1 is held at 5 volts. s_2 corresponds to the conjunctive proposition that the current into the resistor and node n_1 is the same as the current out of the resistor R_1 at node n_2 and that the voltage drop across the resistor is the product of its resistance and the current through. s_3 corresponds to the conjunctive proposition that the current into the resistor R_2 at node n_2 is the same as the current out of the resistor at ground and that the voltage drop across the resistor is the product of its resistance and the current through. Finally, s_4 corresponds to the proposition that the voltage at node n_2 is the product of 5 volts and the resistance of R_2 divided by the sum of the resistances of R_1 and R_2. The equations can now be interpreted as saying that the propositions associated with s_1, s_2, and s_3 hold whenever the corresponding assumptions can be believed. The proposition associated with s_4 is believed whenever the propositions associated with s_1, s_2, and s_3 are believed. A solution to this system will tell us the circumstances under which the various propositions are to be believed. Since the well-founded solution is $\{s_1 = A, s_2 = B_1, s_3 = B_2, s_4 = A \wedge B_1 \wedge B_2\}$, a reasoning agent believes the propositions associated with s_1,s_2,s_3,and s_4 in situations whose meets with (respectively) A, B_1, B_2, and $A \wedge B_1 \wedge B_2$ are not $\bot$.

In the foregoing examples we have made implicit use of the fact that any set of TMS or ATMS justifications has equivalent renderings in the lattice-based formalization. For our last example we consider the classical problem of adding facts to or deleting facts from worlds or states. To begin with, we interpret situations as worlds or states. We have already asserted that every fact or proposition, p, has an associated unknown, say s_p. We will also posit additional unknowns, s_a and s_d, corresponding to the

*Readers interested in the precise details of encoding these other truth maintenance systems in the lattice-theoretic paradigm should consult [Benanav *et al.*, 1986].

beliefs (respectively) that p has been added and that p has been deleted. Consider now the system of equations

$$\begin{aligned} s_a &= A \\ s_d &= B \\ s_p &= s_a \wedge \overline{s}_d. \end{aligned}$$

The well-founded solution of this system is $\{s_a = A, s_d = B, s_p = A \wedge \overline{B}\}$. Our interpretation of this solution is that a reasoning agent believes p just in case he believes himself to be in a world or state whose meet with $A \wedge \overline{B}$ is not $\perp$. In addition, we can use the lattice partial order to encode inheritance among worlds. Notice that the fact p will be added to *any* world, A', such that $A' \leq A \wedge \overline{B}$, and deleted from any world, B', such that $B' \leq B$.

III. The Existence of Solutions

We have seen how lattice-theoretic truth maintenance is connected to some well known models of truth maintenance; we now turn to the challenge of actually solving truth maintenance problems is this new paradigm. We have already seen in Σ_2 that solutions need not exist, but even if they do (as in Σ_5), there may not be well-founded ones. It is well known that general polynomial equations in rational coefficients cannot be solved by applying the operations of addition, multiplication, and rational root extraction to their coefficients [Birkhoff and MacLane, 1965, van der Waerden, 1953]. By analogy we might ask about the solvability of lattice-equational systems by taking meets, joins, and complements of lattice expressions appearing in the equations. Put another way, could it be the case that the equations are not solvable by applying the obvious operations to the available data? To answer this question we must first formalize our notion of the 'available data'.

A *surface element* of a lattice equational system Σ is an element of $\mathcal{B}$ that actually appears in Σ. The Boolean lattice generated by meets, joins, and complements over the surface elements is called the *surface lattice*. An *atom* of the Boolean lattice, $\mathcal{B}$, is any element, $A \in \mathcal{B}$, $A \neq \perp$, such that there is no $B \in \mathcal{B}$ satisfying $A > B > \perp$. A lattice is *atomic* if each of its elements is the join of atoms. If Σ contains only a finite number of equations, the number of surface elements is finite and thus the surface lattice is atomic. The atoms of this lattice will be called *surface atoms*. A *surface solution* is one such that for every lattice unknown, s, that appears in Σ, $\Gamma(s)$ is in the surface lattice. Consider again the system, Σ_4. Note that it has many possible well-founded solutions (depending on the Boolean lattice with respect to which the system is being interpreted) of which only two, $\{s_1 = \top, s_2 = \perp\}$ and $\{s_1 = \perp, s_2 = \top\}$ are surface. Our question posed in the last paragraph is answered by the following:

Theorem III.1 *If a finite lattice equational system Σ over $\mathcal{B}$ has a well-founded solution, then, it has a well-founded surface solution.*

Thus we see that if there are any well-founded solutions at all, we are guaranteed that some of them can be computed by taking meets, joins, and complements over the available data.

Thus far we have established a framework within which we can formally describe truth maintenance problems *and* within which solutions can be connected with the available data in the equations. For this framework to be truly useful we must provide a way of finding solutions other than by blindly enumerating candidates and testing them. Suppose we could obtain *a* solution. How do we know that this is the *only* solution? Or even the only *surface* solution? To convey some idea of the challenge of this problem consider the system Σ_8

$$\begin{aligned} s &= s \wedge \bigwedge_{1 \leq k \leq n} \overline{s}_{2k} \\ s_1 &= s \vee \overline{s}_2 \\ s_2 &= \overline{s}_1 \\ &\vdots \\ s_{2n-1} &= s \vee \overline{s}_{2n} \\ s_{2n} &= \overline{s}_{2n-1}. \end{aligned}$$

This system has 2^n well-founded surface solutions. s is always $\perp$ and we are free to choose $\perp$ *or* $\top$ as the value of *each* of the odd-indexed unknowns. The usual method of solving algebraic equational systems is by using substitution together with other "legal" (with respect to the algebraic system in question) transformations to produce a new lattice equational system whose solutions are also solutions of the original. Because of the algebraic nature of the meet and join operators there is no obvious way of effecting such a transformation. Before introducing the more novel transformation that we will need, let us first formalize the notion of substitution that we will be using.

Let Σ_{-s} be Σ less its s equation. Systems will be presented always according to some fixed lexical order. This is possible since each system is obviously a recursive set. Hence it is reasonable to speak of the nth occurrence of the unknown, s, on the right-hand side of an equation in Σ. We define a *local substitution*, $\sigma_{s,n}$, as follows: If the s' equation is the locus of the nth occurrence, then $\sigma_{s,n}(\Sigma)$ is $\Sigma_{-s'}$ together with a new s' equation wherein the nth occurrence of s is replaced by the right-hand side of the s equation in Σ. If there is no n^{th} then $\sigma_{s,n}(\Sigma) = \Sigma$. Suppose there are k s's in Σ before the s equation, m in the s equation, and n after the s equation. The (global) *substitution transformation* of Σ under s, $\sigma_s(\Sigma)$, is

$$\sigma_{s,1}(\cdots\sigma_{s,k-1}(\sigma_{s,k}(\sigma_{s,k+m+1}(\cdots\sigma_{s,k+m+n-1}(\sigma_{s,k+m+n}(\Sigma))\cdots)))\cdots)$$

This transformation has the effect of replacing every right-hand side occurrence of s (except those in the s equation) with the right-hand side of the s equation. The following lemma suggests the other transformation necessary for computing solutions.

Lemma III.1 *Let X_1, X_2 and X_3 be expressions having no occurrence of s, then*

$$s = X_1 \vee (X_2 \wedge s) \vee (X_3 \wedge \bar{s})$$
$$\Leftrightarrow \quad (X_1 \vee X_3) \leq s \leq (X_1 \vee X_2)$$
$$\Leftrightarrow \quad (\Gamma(X_1) \vee \Gamma(X_3)) \leq \Gamma(s) \leq (\Gamma(X_1) \vee \Gamma(X_2))$$

where Γ is a solution of a system including the s equation.

Let the s equation be rearranged to have the form: $s = X_1 \vee (X_2 \wedge s) \vee (X_3 \vee \bar{s})$. The distributive nature of the Boolean lattice guarantees that we can always do this. The *minimization transformation* of Σ under s, $\mu_s(\Sigma)$, is Σ_{-s} together with the equation $s = X_1 \vee X_3$. This transformation is semantically equivalent to having substituted $\perp$ for every occurrence of s on the right-hand side of the s equation. Or put another way, we are taking the lower bound of the solution interval defined in the previous lemma.

We ask then whether or not minimization and substitution can be used to produce a solution. The answer will always be in the affirmative for an important class* of equational systems. We may apply these transformations to the original system of equations in such a way as to yield a new system free of unknowns on the right-hand side. The resulting system constitutes a solution for the original system of equations. Before discussing the exact method of applying these transformations, however, we offer the following apparently technical but actually qualitatively important result about minimization and substitution.

Lemma III.2 *A well-founded solution, Γ, of a lattice equational system, Σ, is a well-founded solution of $\sigma_s(\Sigma)$ unless σ_s involves a local substitution for a complemented occurrence of s. If $\mu_s(\Sigma) = \Sigma'$, then a well-founded solution Γ' of Σ' is a well-founded solution of Σ.*

A close examination of the proof would reveal that the substitution operation has the property that it preserves "solution-ness" but may lose well-foundedness. On the other hand, minimization preserves well-foundedness, in the sense that any well-founded solution to the original system that *persists* in being a solution to the transformed system is still well-founded. Consider first the system Σ_4. Applying the transformation σ_{s_2} yields the system Σ_4'

$$s_1 = s_1$$
$$s_2 = \bar{s}_1$$

which has the same surface solutions as Σ_4, but only the second of them is well-founded. Applying μ_{s_1} to Σ_4' yields

$$s_1 = \perp$$
$$s_2 = \bar{s}_1$$

which has only one solution, $\{s_1 = \perp, s_2 = \top\}$.

A *process* for Σ is any functional composition of minimizations and substitutions. μ_s, $\sigma_{s,n}$ and σ_s are all processes for any lattice unknown, s. If π is a process, so are $\mu_s \circ \pi$, $\sigma_{s,n} \circ \pi$ and $\sigma_s \circ \pi$. A *terminal* process for Σ, denoted τ, is a process such that for every process π, $\tau \circ \pi(\Sigma) = \tau(\Sigma)$.

A *path of length n* from s_0 to s_n is a sequence of triples of the form

$$\langle X_1, Y_1, s_1\rangle, \langle X_2, Y_2, s_2\rangle, \ldots, \langle X_n, Y_n, s_n\rangle$$

where $X_i \in \{s_{i-1}, \bar{s}_{i-1}\}$, X_i is an antecedent of the Y_i disjunct of the s_i equation in Σ, and $1 \leq i \leq n$. X_i is a complemented (uncomplemented) unknown if it is a complemented (uncomplemented) conjunct of Y_i. Unknown s is connected to unknown s' if there is a path of any length from s to s'. 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.

Theorem III.2 *Every even lattice equational system, Σ, has a terminal process, τ, such that $\Gamma_{\tau(\Sigma)}$ is a well-founded solution of Σ.*

An immediate consequence of the previous theorem is an algorithm that is analogous to Gaussian elimination [Birkhoff and MacLane, 1965, Gantmacher, 1959] that will always produce a well-founded solution for an even lattice equational system:

1. Let x be a LIFO queue of the equations in Σ; let y be an empty LIFO queue of equations in Σ; let z be an equation in Σ
2. Until x is empty, x becomes $\sigma_s(\mu_s(x))$ (the order of equations remaining invariant with respect to their left-hand sides) where s is the unknown on the left-hand side of the first equation in the queue, dequeue x to z, enqueue z to y
3. Until y is empty, y becomes $\sigma_s(y)$ (the order of equations remaining invariant with respect to their left-hand sides) where s is the unknown on the left-hand side of the first equation in the queue, dequeue y to z, enqueue z to x
4. End

This algorithm terminates with x being a queue of equations with constant right-hand sides (a solution). Step 2 is the analogue of forward elimination; step 3 corresponds to back substitution. Each (unspecified) order in which unknowns are removed from the queue, x, is an *elimination sequence*. Every such sequence produces a well-founded solution (for an even system) and it may be the *only* sequence that produces that particular solution. In re-examining Σ_4 above we implicitly applied this algorithm, first eliminating s_1 and thereby generating the first of the two well-founded surface solutions. Had we done the other elimination first, we would have obtained the second well-founded surface solution. Can we generate all of the surface solutions by varying the order of elimination? Unfortunately the answer is no as can be seen by examining the following system, Σ_9:

$$s_1 = A \wedge \bar{s}_2$$

*The well-known truth maintenance systems in the literature only guarantee well-founded solutions for even equational systems.

$$s_2 = (A \wedge \overline{s}_1) \vee \overline{s}_3$$
$$s_3 = \overline{s}_2$$

only three of whose four well-founded solutions can be produced by varying the order of elimination. As suggested by the statement of the theorem, there are odd lattice equational systems some of whose elimination sequences do not produce solutions. Such a system is Σ_{10}:

$$s_1 = \overline{s}_2$$
$$s_2 = \overline{s}_1$$
$$s_3 = s_1 \wedge \overline{s}_3.$$

Addressing either of the aforementioned deficiencies requires the separability results of the next section.

IV. The Structure of Systems and Solutions

In this section we discuss some separability results for lattice equational systems. These results are of two classes: topological and algebraic. They are important because

- They provide the machinery from which all surface solutions to all equational systems may be generated
- They provide the basic perspective for analyzing the abstract complexity of the truth maintenance problem
- They suggest concrete "divide and conquer" algorithms for solving the truth maintenance problem by
  - supporting "lazy evaluation" of the solution *vis à vis* any given unknown
  - supporting incremental update of solutions as justifications are added or removed
  - enabling parallel solution methods

Let the equations of Σ be partitioned into equivalence classes in such a way that the s_1 and s_2 equations of Σ will be in the same equivalence class Σ', a subsystem of Σ, just in case s_1 is connected to s_2 in Σ and s_2 is connected to s_1 in Σ. Each equivalence class Σ' is a *strongly connected subsystem* of Σ. A partial order, $\leq_\Sigma$, can be defined on the strongly connected subsystems of an equational system such that for subsystems Σ' and Σ'', $\Sigma' \leq_\Sigma \Sigma''$ if and only if there is an s' equation of Σ' and an s'' equation of Σ'' such that s' is connected to s''. Σ'' is *minimal*, if and only if there exists no Σ' such that $\Sigma' \leq_\Sigma \Sigma''$. Given a partitioning of a lattice equational system Σ into strongly connected subsystems, there always exists at least one minimal strongly connected subsystem of Σ.

Proposition IV.1 *Every lattice equational system can be partitioned into a partial order of strongly connected subsystems. Moreover, each of these subsystems can be treated as if the unknowns whose corresponding equations are in other strongly connected subsystems were lattice elements (i.e., constants).*

Since we can solve each of the partitions separately, treating the unknowns whose corresponding equations are *not* in the strongly connected subsystem being solved as if they were expressions from the surface lattice (*i.e.*, constants), we can pursue a strategy of lazy evaluation and incremental update. Not only can the system be partitioned very efficiently [Aho *et al.*, 1974], but there are also efficient methods of updating this partially ordered partition as justifications are added (and deleted). As things change in the truth maintenance database, we compute new partitions and only *re*-solve for unknowns whose equations are in the same strongly connected subsystem as the changed equation, and (optionally) for unknowns in greater subsystems. For example, consider Σ_{10}. This system partitions into two strongly connected subsystems. The s_1 and s_2 equations form the first and lesser (in $\leq_{\Sigma_{10}}$) subsystem; the second strongly connected subsystem consists of the s_3 equation. We can solve the second subsystem, treating it as if s_1 were a constant. Note that the s_3 equation has a solution only in the case that $s_1 = s_3 = \bot$. This "forces" the solution of the first subsystem to be $\{s_1 = \bot, s_2 = \top\}$.

A lattice equational system, Σ, is *reduced* if and only if for every pair of unknowns s and s' whose equations are in the same strongly connected subsystem of Σ such that s is an antecedent of s' in Σ, s is a nonmonotonic antecedent of s'.

Theorem IV.1 *For every lattice equational system, Σ, there is a process, π, such that $\Sigma' = \pi(\Sigma)$ is reduced, and Γ is a well-founded solution of Σ if and only if it is a well-founded solution of Σ'.*

The utility of the previous theorem becomes clearer when we combine the computation of strongly connected subsystems of a system with reduction and minimization. Since we need to do only *substitutions* through uncomplemented occurrences of unknowns in the original system in order to reduce it, the resulting system has exactly the same well-founded solutions as the original. Reconsider now Σ_8. If we minimize with respect to s and reduce again we get the reduced system:

$$\begin{aligned} s &= \bot \\ s_1 &= \overline{s}_2 \\ s_2 &= \overline{s}_1 \\ \vdots &\ \ \vdots\ \ \vdots \\ s_{2n-1} &= \overline{s}_{2n} \\ s_{2n} &= \overline{s}_{2n-1}. \end{aligned}$$

The system has now been separated into $n+1$ strongly connected subsystems, each of which is *disconnected* from the others, hence independently solvable. We see now how the 2^n solutions arise, since we have n repetitions of the system Σ_4.

If we combine the previous two results, and manipulate the proof of theorem III.2 we obtain

Theorem IV.2 *Let Σ be a lattice equational system whose surface lattice is $\{\top, \bot\}$. Every well-founded surface*

solution of Σ is produced by** some **sequence of minimizations and global substitutions.

Thus if we restrict the lattice over which the equations are taken to correspond exactly to the TMS's of Doyle and Goodwin, each well-founded solution will be produced by some elimination sequence. Unfortunately, among the large number of elimination sequences, many (in the case of odd systems) do not produce solutions. Since we and others [McAllester] have independently shown the NP-completeness (in the size of the equational system) of solving this restricted class of lattice equational systems, there can be no "easy" characterization of the circumstances under which a particular elimination sequence will produce a well-founded solution.

We turn now to the question of how to find all the well-founded surface solutions for arbitrary systems. Let $\Gamma_1 \vee \Gamma_2$, $\Gamma_1 \wedge \Gamma_2$, and $A \wedge \Gamma$ denote functions such that $(\Gamma_1 \vee \Gamma_2)(s) = \Gamma_1(s) \vee \Gamma_2(s)$, $(\Gamma_1 \wedge \Gamma_2)(s) = \Gamma_1(s) \wedge \Gamma_2(s)$, and $(A \wedge \Gamma)(s) = A \wedge \Gamma(s)$. A *Goodwin projection* of a lattice element B with respect to an atom A, denoted $\gamma_A(B)$, is defined by

$$\gamma_A(B) = \begin{cases} \top & \text{if } A \leq B \\ \bot & \text{otherwise.} \end{cases}$$

We extend the notion of a Goodwin projection to expressions over lattice elements by

$$\begin{aligned} \gamma_A(B_1 \wedge B_2) &= \gamma_A(B_1) \wedge \gamma_A(B_2) \\ \gamma_A(B_1 \vee B_2) &= \gamma_A(B_1) \vee \gamma_A(B_2) \\ \gamma_A(\overline{B}) &= \overline{\gamma_A(B)} \end{aligned}$$

and to lattice equations by applying the projection to each constant term in each equation.

Theorem IV.3 *Γ_Σ is a well-founded surface solution of Σ if and only if there exists a subset, $\{A_i | 1 \leq i \leq N\}$, of the atoms of the surface lattice of Σ, and corresponding Goodwin projections, $\{\gamma_{A_i} | 1 \leq i \leq N\}$, such that*

$$\Gamma_\Sigma = \bigvee_{i=1}^{N} A_i \wedge \Gamma_{\gamma_{A_i}(\Sigma)}$$

where the $\Gamma_{\gamma_{A_i}(\Sigma)}$ are well-founded surface solutions.

This theorem guarantees a unique prime (where the primes are the surface atoms) factorization of lattice equational systems and their solutions. Notice that each Goodwin projection of a system Σ results in a system whose surface lattice is $\{\top, \bot\}$. We "know" how to solve these by theorem IV.2. Hence factoring followed by finding all of the (successful) elimination sequences produces all of the surface solutions. In particular, all four of the well-founded surface solutions of Σ_9 can be produced by taking the Goodwin projections with respect to the surface atoms, A and $\overline{A}$, solving the two resulting systems, "multiplying" the resulting solutions by A and $\overline{A}$ respectively, and "adding" the results. Since each projected system has two solutions, the overall system has four. Elsewhere [Benanav *et al.*] we have shown the problem of solving general lattice equational systems to be NP-*hard* in the size of the system. We see now how this might arise: Using the algebraic results that we have cited can produce solutions at the cost of composing two potentially exponential processes, the projection by surface atoms and the finding of minimization and substitution sequences that actually produce a well-founded solution.

V. Conclusions

In the foregoing we have introduced a general model of truth maintenance couched in a lattice-theoretic framework. All of the truth maintenance systems familiar to us in the literature can be construed as solving systems of lattice equations. Indeed, those systems can be properly embedded in our lattice-theoretic formalism. We introduced the fundamental transformations of substitution and minimization and showed how they could be used to produce solutions of even lattice equational systems. We have cited a number of theoretical results about the algebraic structure of truth maintenance systems and interpreted these results in terms of concrete examples. We have used these examples to illustrate how a given formal algebraic result either reveals some intrinsic difficulty, or how it can be used to computational advantage. Finally, we sketched how our separation results can be used to generate all of the surface solutions of an arbitrary lattice equational system. The principal technical contributions of those aspects of our work on lattice-theoretic truth maintenance that we have presented in this paper are:

- The formalization of truth maintenance in a way that properly includes nonmonotonic justifications and assumption-based justifications
- The presentation of a point of view from which one can algebraically analyze the structure of truth maintenance problems and the construction of solutions to those problems
- The motivation of the algebraic results with computational and phenomenological interpretations

Though not the topic of this paper, it is also from this same lattice-theoretic point of view that we have carried out the analysis of the abstract and computational complexity of truth maintenance.

References

[Aho and Ullman, 1977] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[Aho *et al.*, 1974] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[Balbes and Dwinger, 1974] Raymond Balbes and Philip Dwinger. *Distributive Lattices*. University of Missouri Press, Columbia, Missouri, 1974.

[Benanav *et al.*, 1986] Dan Benanav, Allen L. Brown, Jr., and Dale E. Gaucas. Reason maintenance from a lattice-theoretic point of view. In Lee S. Baumann, editor, *Proceedings of the Expert Systems Workshop*, pages 83–87, Defense Advanced Research Projects Agency, Science Applications International Corporation, Pacific Grove, California, April 1986.

[Benanav *et al.*] Dan Benanav, Allen L. Brown, Jr., and Dale E. Gaucas. *A Lattice-Theoretic Framework for Reason Maintenance*. Technical Report, General Electric Corporate Research and Development Center, Schenectady, New York, forthcoming.

[Birkhoff, 1967] Garrett Birkhoff. *Lattice Theory*. Volume 25 of *American Mathematical Society Colloquium Publications*, American Mathematical Society, Providence, Rhode Island, third edition, 1967.

[Birkhoff and MacLane, 1965] Garrett Birkhoff and Saunders MacLane. *A Survey of Modern Algebra*. MacMillan, New York, third edition, 1965.

[de Kleer, 1984] Johan de Kleer. Choices without backtracking. In *Proc. 4th Nat. Conf. on Artificial Intelligence*, pages 79–85, Austin, 1984.

[de Kleer, 1986a] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[de Kleer, 1986b] Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, 28:163–196, 1986.

[de Kleer, 1986c] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28:197–224, 1986.

[Doyle, 1978] Jon Doyle. *Truth Maintenance Systems for Problem Solving*. Technical Report AI-TR-419, Massachussetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, January 1978.

[Doyle, 1979a] Jon Doyle. A glimpse of truth maintenance. In *Proc. 6th Int. Joint Conf. on Artificial Intelligence*, pages 232–237, 1979.

[Doyle, 1979b] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Gantmacher, 1959] F.R. Gantmacher. *Matrix Theory*. Volume 1, Chelsea, New York, 1959.

[Gaucas and Brown] Dale E. Gaucas and Allen L. Brown, Jr. *A Role for Assumption-based and Non-monotonic Justifications in Automating Strategic Threat Analysis*. These proceedings.

[Goodwin, 1982] James W. Goodwin. *An Improved Algorithm for Non-Monotonic Dependency Update*. Technical Report LITH-MAT-R-82-23, Linköping University, Linköping, Sweden, August 1982.

[Goodwin, 1984] 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.

[Goodwin, 1985] 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.

[Goodwin, 1987] James W. Goodwin. *A Theory and System for Non-Monotonic Reasoning*. PhD thesis, Linköping University, Linköping, Sweden, 1987.

[Hecht, 1977] Matthew S. Hecht. *Data Flow Analysis of Computer Programs*. American Elsevier, New York, 1977.

[McAllester] David A. McAllester. Personal communication.

[McDermott, 1983] Drew McDermott. Contexts and data dependencies: a synthesis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3):237–246, May 1983.

[Morris and Nado, 1986] Paul H. Morris and Robert A. Nado. Representing actions with an assumption-based truth maintenance system. In *Proc. 5th Nat. Conf. on Artificial Intelligence*, pages 13–17, Philadelphia, 1986.

[Schaeffer, 1973] Marvin Schaeffer. *A Mathematical Theory of Global Program Optimization*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[Skornjakov, 1977] L.A. Skornjakov. *Elements of Lattice Theory*. Hindustan Publishing Corporation, Dehli, India, 1977. Translated from the Russian by V. Kumar.

[van der Waerden, 1953] B.L. van der Waerden. *Modern Algebra*. Ungar, New York, 1953.

[Waite and Goos, 1984] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, New York, 1984.

Logics of Justified Belief

Allen L. Brown, Jr.
GE Corporate Research and Development Center
P.O. Box 8
Schenectady, New York 12301

Abstract

We give a formal semantics to truth maintenance by offering here a mathematical logic—equipped with an underlying model theory—that is used to characterize quite precisely some well known models of truth maintenance. Our usage of 'precise' is doubly intended in that we give meaning to truth maintenance in terms of a formal logic, *and* that each characterizing logic corresponds to a particular truth maintenance system and *vice versa*.

I. Introduction

The history of mathematics is replete with formal systems consisting of symbols and operational transformations on same, wherein the utility of the formal systems had been explored and exploited long before completely satisfactory (mathematical) semantical accounts of the systems were provided. Examples of this are the integral calculus, predicate calculus and the lambda calculus whose corresponding mathematical semantics are respectively Lebesgue measure, Tarskian semantics, and Scott semantics. Truth maintenance systems are a more recent instance of this phenomenon where operational utility has been realized in advance of mathematical justification. Although there are various logical accounts of nonmonotonic reasoning (see [Perlis, 1984] for a complete survey) that have been equipped with suitable formal semantics (including our own attempt in [Brown, 1985]), none of these accounts captures truth maintenance with satisfactory precision. Th questions we propose to answer here are:

1. With respect to what logic might the formulae labeled as "IN" by truth maintenance systems be counted as theorems?
2. What exactly is the logical status of formulae labeled as "OUT"?

In the following we will develop logics and associated semantics that correspond to the the TMS's of Doyle [Doyle, 1979a, Doyle, 1979b, Doyle, 1978] and Goodwin [Goodwin, 1982, Goodwin, 1985, Goodwin 1984, Goodwin, 1987], the ATMS of de Kleer [de Kleer, 1986a, de Kleer, 1984, de Kleer, 1986b, de Kleer, 1986c], and our own ANRMS [Brown *et al.*, 1987, Benanav *et al.*, Forthcoming, Gaucas and Brown, 1987]. We will first provide a logic and model theory for the TMS's of Doyle and Goodwin. We will then reduce the logical characterization of other TMS's to the Doyle/Goodwin case. As we pointed out earlier, our principle task is to make logical sense of "IN" and "OUT". We do this this by formalizing the propositional attitude of *belief* for the propositions of an underlying logical language. We call these beliefs ***justified*** in that they are the consequents of syntactically well-formed arguments. We distinguish them from the ***true beliefs*** [Gettier, 1967, Griffiths, 1967, Malcolm, 1967, Prichard, 1967] ordinarily of interest to philosophers in that we are disinterested in the logical soundness of the arguments in question (just as is the case for a truth maintenance system). The logics we construct will give a syntactic characterization to justifications and beliefs. When consistent, these logics will count propositions as believed just in case the corresponding truth maintenance system would have labeled them "IN".

II. Nonmonotonic Truth Maintenance

A. Syntax

Let $\mathcal{L}$ be a first-order language equipped with functions, predicates, connectives, quantifiers, and perhaps even modalities.* $\mathcal{L}$ has the usual formation rules for first-order languages. The details of $\mathcal{L}$ will not concern us very much here. p, q, r (possibly subscripted) range over formulae of $\mathcal{L}$. We define the language $\mathcal{L}_1$ as follows:

1. If p is a formula of $\mathcal{L}$, then $\mathcal{B}[p]$ and $\neg\mathcal{B}[p]$ are formulae of $\mathcal{L}_1$. Formulae of this form are called *elementary* (respectively *positive* and *negative*) *beliefs* with *core* p. The set of beliefs (positive and negative) will be denoted $\mathcal{B}[\mathcal{L}]$. Similarly, if $A \subseteq \mathcal{L}$, $\mathcal{B}[A]$ is the set of positive and negative beliefs whose cores are in A.
2. If $p, q_1, \ldots, q_m, r_1, \ldots, r_n$ are formulae of $\mathcal{L}$, then $\mathcal{J}[p]$, $\mathcal{J}[p|q_1, \ldots, q_m]$, $\mathcal{J}[p|q_1, \ldots, q_m|r_1, \ldots, r_n]$, and $\mathcal{J}[p||r_1, \ldots, r_n]$, are all in $\mathcal{L}_1$. Formulae of the latter form are called *justifications*. p is the *consequent* of the justifications, while $q_1, \ldots, q_m$ and $r_1, \ldots, r_n$ are respectively the *monotonic* and *nonmonotonic antecedents*

*We will freely use the connectives and quantifiers of first-order logic as part of our ordinary mathematical discourse. Since no formulae of $\mathcal{L}$ are ever actually mentioned, we trust that this will cause no confusion.

of the justifications where they are mentioned. j will be a variable that ranges over justifications. Justifications without nonmonotonic antecedents are termed *monotonic* while those with are termed *nonmonotonic*. $\alpha(j)$, and $\overline{\alpha}(j)$, and $\kappa(j)$ are respectively the monotonic, and nonmonotic antecedents, and the consequent of the justification, j.

3. No other formulae are in $\mathcal{L}_1$.

Since $\mathcal{L}$ (and consequently $\mathcal{L}_1$) is presumed to be recursive, we may also presume the existence of a total lexical ordering on the formulae of $\mathcal{L}_1$.

A *TMS theory*, $\mathcal{T}$, is any *finite* set of justifications. A TMS theory having nonmonotonic justifications is nonmonotonic. Let A be the subset of $\mathcal{L}$ containing exactly those formulae that appear as antecedents or consequents of justifications in $\mathcal{T}$. Let Y be the set

$$\{\mathcal{T} \cup X | X \subseteq \mathcal{B}[A] \wedge (\forall p \in A)\mathcal{B}[p] \in X \vee \neg\mathcal{B}[p] \in X\}$$

Let $\mathcal{F}(\mathcal{S})$ be a subset of $\mathcal{L}_1$ such that

1. $\mathcal{S} \subseteq \mathcal{F}(\mathcal{S})$;
2. $\mathcal{B}[p]$ is in $\mathcal{F}(\mathcal{S})$ whenever there is a justification in $\mathcal{S}$ of which p is the consequent, for each of whose monotonic antecedents, q, $\mathcal{B}[q]$ is in $\mathcal{S}$, and for each of whose nonmonotonic antecedents, r, $\neg\mathcal{B}[r]$ in $\mathcal{S}$;
3. $\neg\mathcal{B}[p]$ is in $\mathcal{F}(\mathcal{S})$ if for every justification in $\mathcal{S}$ of which p is the consequent, then for at least one of its monotonic antecedents, q, $\neg\mathcal{B}[q]$ is in $\mathcal{S}$, or for at least one of its nonmonotonic antecedents, r, $\mathcal{B}[r]$ in $\mathcal{S}$;
4. no other formulae are in $\mathcal{F}(\mathcal{S})$.

A *justification closure operator*, $\mathcal{C}_\mathcal{T}$, for a TMS theory, $\mathcal{T}$, is a function on elements of Y defined by

$$\mathcal{C}_\mathcal{T}(\mathcal{S}) = \mathcal{F}(\mathcal{T} \cup \mathcal{S}).$$

We are typically interested in the least fixed points of $\mathcal{C}_\mathcal{T}$. We will abuse our notation by occasionally referring to a particular fixed point as $\mathcal{C}_\mathcal{T}$. Also, we will refer to a fixed point of a theory, $\mathcal{T}$, meaning a fixed point of the justification closure operator for that theory. A set of formulae, $\mathcal{S}$, will be termed *inconsistent* if it contains both a belief, $\mathcal{B}[p]$, and its negation, $\neg\mathcal{B}[p]$. Notice that it will typically be the case that a fixed point will be indifferent to most beliefs (*i.e.* it will contain neither $\mathcal{B}[p]$ nor $\neg\mathcal{B}[p]$). In such cases we are free to add either a positive (exclusive) or negative belief and still have a consistent fixed point of $\mathcal{T}$, though no longer least. Since truth maintenance systems by and large profess disbelief in any formula (of $\mathcal{L}$) for which there is no argument, we will augment a least fixed point with any negative belief for which there is no corresponding positive belief in the fixed point. Such an augmentation is a *justification completion* of a TMS theory.

For fixed points to be interesting they must exist:

Proposition II.1 *Every TMS theory has a least fixed point.*

Proof. Let $\mathcal{T}$ be a TMS theory. Clearly $\mathcal{T} \cup \mathcal{B}[A]$ is in Y and is a finite fixed point of $\mathcal{C}_\mathcal{T}$. $\mathcal{T} \cup \mathcal{B}[A]$ being finite, it contains some least subset $\mathcal{S}$ that is an element of Y and is a fixed point of $\mathcal{C}_\mathcal{T}$. Hence $\mathcal{S}$ is a least fixed point of $\mathcal{C}_\mathcal{T}$.□

Since TMS labelings are obviously nonmonotonic in that the addition of new justifications can cause formulae formerly labeled as "IN" to be relabeled "OUT", the corresponding logical theory ought to have this property as well:

Proposition II.2 *There exist TMS theories $\mathcal{T}_1$ and $\mathcal{T}_2$ such that there is no least fixed point of $\mathcal{T}_1 \cup \mathcal{T}_2$ containing any least fixed point of $\mathcal{T}_1$.*

Proof. Consider the TMS theories $\{\mathcal{J}[p||p]\}$ and $\{\mathcal{J}[p]\}$. The first theory has the least fixed point $\{\mathcal{J}[p||p], \mathcal{B}[p], \neg\mathcal{B}[p]\}$, and the second has the least fixed point $\{\mathcal{J}[p], \mathcal{B}[p]\}$, while their union has the least fixed point $\{\mathcal{J}[p], \mathcal{J}[p||p], \mathcal{B}[p]\}$. Since all of these least fixed points are unique for their respective theories, the proposition follows.□

A partial order, a subset of $\mathcal{D}^2$, is *graded* if there is a function from $\mathcal{D}$ into the non-negative integers such that

1. every $d \in \mathcal{D}$ has a grade;
2. the grade of $d \in \mathcal{D}$ is 0 whenever there is no $d' \in \mathcal{D}$ such that d' is less than d in the partial order;
3. the grade of each $d \in \mathcal{D}$ is larger than that of every $d' \in \mathcal{D}$ smaller than d in the partial order.

The grading, δ, for a partial order over the domain, $\mathcal{D}$ will be termed *standard* if it satisfies $\delta(d) = 1 + \max\{\delta(d') | d' < d \wedge \neg(\exists d'')d' < d'' < d\}$. A unique standard grading always exists for a graded partial order. Henceforth we will assume 'standard' whenever we mention 'grading'. A fixed point, $\mathcal{S}$, of a theory, $\mathcal{T}$, is *well-founded* if there is a graded partial order, $<_\mathcal{S}$, on positive beliefs (of S) such that for every positive belief, $\mathcal{B}[p] \in \mathcal{S}$, there is a justification, $J[p|q_1, \ldots, q_m|r_1, \ldots, r_n]$, such that $\mathcal{B}[q_1], \ldots, \mathcal{B}[q_m] \in \mathcal{S}$, $\neg\mathcal{B}[r_1], \ldots, \neg\mathcal{B}[r_n] \in \mathcal{S}$, and $q_1, \ldots, q_m <_\mathcal{S} p$.

We complete this section with some additional proof-theoretic results for TMS theories that will serve us later in our investigation.

Proposition II.3 *Let $\mathcal{S}$ be a well-founded least fixed point of the justification closure operator, $\mathcal{C}_\mathcal{T}$ of the TMS theory $\mathcal{T}$. Let $<_\mathcal{S}$ be a graded partial order for $\mathcal{S}$. There exists a least partial order contained in $<_\mathcal{S}$ under which $\mathcal{S}$ remains well-founded.*

Proof. Suppose $\mathcal{S}$, $\mathcal{T}$ and $<_\mathcal{S}$ are as in the statement of the proposition. By the definition of well-founded, for every $\mathcal{B}[p] \in \mathcal{S}$ there is a justification, $j \in \mathcal{T}$ such that $\kappa(j) = p$ and for each $q \in \alpha(j)$, $\mathcal{B}[q] <_\mathcal{S} p$ and $\mathcal{B}[q] \in \mathcal{S}$. Our first task is to extend the partial order, $<_\mathcal{S}$, to include justifications in $\mathcal{T}$. Let $<_1$ be the least transitive partial ordering on beliefs and justifications such that: $q <_1 j$ if and only if $j \in \mathcal{T}$, $\mathcal{B}[\kappa(j)] \in \mathcal{S}$, $q \in \alpha(j) <_\mathcal{S} \kappa(j)$, $(\forall r \in \alpha(j))\mathcal{B}[r] \in \mathcal{S} \wedge r <_\mathcal{S} \kappa(j)$, $(\forall r \in \overline{\alpha}(j))\neg\mathcal{B}[r] \in \mathcal{S}$. $j <_1 p$ if and only if $j \in \mathcal{T}$, $p = \kappa(j)$, $\mathcal{B}[p] \in \mathcal{S}$, $(\forall q \in \alpha(j))\mathcal{B}[q] \in \mathcal{S} \wedge q <_\mathcal{S} p$, and $(\forall r \in \overline{\alpha}(j))\neg\mathcal{B}[r] \in \mathcal{S}$. Clearly

if $<_{\mathcal{S}}$ is graded, so is $<_1$. Let $<_2$ be a suborder of $<_1$ such that $p <_2 j$ if and only if $p <_1 j$, and $j <_2 p$ if and only if j is the lexically smallest justification of least grade (in $<_1$) having p as a consequent. Clearly any given p is immediately preceded (in the $<_2$ order) by at most one justification. Finally, $x <_3 y$ if and only if $x <_2 y$ and $y <_2 p$ where $\mathcal{B}[p] \in \mathcal{S}$. $<_3$ is a graded partial order having the property that it is a suborder of $<_{\mathcal{S}}$ and every positive belief of $\mathcal{S}$ *not* in $\mathcal{T}$ is preceded (in $<_3$) by exactly one justification from $\mathcal{T}$. $<_3$ is clearly minimal.□

A TMS theory, $\mathcal{T}$, together with $\mathcal{C}_{\mathcal{T}}$ is a logic of justified belief. As mentioned earlier, we speak of justified beliefs rather than true beliefs. For a belief to be justified we merely require it to be grounded in a well-founded argument (the partial order $<_{\mathcal{T}}$ together with a suitable set of justifications). Thus it is possible for both $\mathcal{B}[p]$ and $\mathcal{B}[\neg p]$ to be justified in a *consistent* TMS theory even though this pair of beliefs would not be held by a *rational* agent. This contrasts with logics of true belief wherein the cores of positive (negative) beliefs are typically (non-)theorems is some underlying $\mathcal{L}$-theory. In general we shall be interested in least fixed points of the justification operators of particular TMS theories, where those fixed points are well-founded under the associated partial order. Consider the TMS theories

$$
\begin{aligned}
\mathcal{T}_1 &= \{\mathcal{J}[p||p]\}, \\
\mathcal{T}_2 &= \{\mathcal{J}[p|q], \mathcal{J}[q|p]\}, \\
\mathcal{T}_3 &= \{\mathcal{J}[p||q], \mathcal{J}[q||p]\}, \\
\mathcal{T}_4 &= \{\mathcal{J}[p||p], \mathcal{J}[p|q], \mathcal{J}[q|p], \mathcal{J}[q||q]\}.
\end{aligned}
$$

$\mathcal{T}_1$ has a single least fixed point, $\mathcal{T}_1 \cup \{\mathcal{B}[p], \neg\mathcal{B}[p]\}$, and it is inconsistent. $\mathcal{T}_2$ has two consistent least fixed points, $\mathcal{T}_2 \cup \{\neg\mathcal{B}[p], \neg\mathcal{B}[q]\}$ and $\mathcal{T}_2 \cup \{\mathcal{B}[p], \mathcal{B}[q]\}$ of which the first is well-founded. $\mathcal{T}_3$ has two least fixed points, $\mathcal{T}_3 \cup \{\mathcal{B}[p], \neg\mathcal{B}[q]\}$ and $\mathcal{T}_3 \cup \{\mathcal{B}[q], \neg\mathcal{B}[p]\}$, and each of them is consistent and well-founded. $\mathcal{T}_4$ has a single least fixed point, $\mathcal{T}_4 \cup \{\mathcal{J}[q|p], \mathcal{J}[q||q], \mathcal{B}[q], \mathcal{B}[p]\}$ and it is consistent and *not* well-founded.

Proposition II.4 *A justification completion of a monotonic TMS theory is consistent.*

Proof: That there *is* a justification completion has already been guaranteed. Without loss of generality, we may replace 'completion' with 'closure' in the statement of the proposition. Suppose that there are elementary beliefs $\mathcal{B}[p]$ and $\neg\mathcal{B}[p]$ in the justification closure of the monotonic theory, $\mathcal{T}$. Suppose further that all of the justifications in $\mathcal{T}$ having p as a consequent have no antecedents. The definition of $\mathcal{F}$ guarantees that no least fixed point of $\mathcal{C}_{\mathcal{T}}$ can contain both $\mathcal{B}[p]$ and $\neg\mathcal{B}[p]$. Hence for every contradictory pair $\{\mathcal{B}[p], \neg\mathcal{B}[p]\}$ there must be at least one justification having p as a consequent *and* a non-empty set of antecedents. Furthermore, the definition of $\mathcal{F}$ also guarantees that for each justification with consequent p there is an antecedent q such that $\mathcal{B}[p]$ and $\neg\mathcal{B}[p]$ are in the justification closure of $\mathcal{T}$. It is easily verified that if we remove the negative belief of every contradictory pair in the justification closure, the resulting (consistent) set of beliefs is *still* a fixed point of $\mathcal{C}_{\mathcal{T}}$. This last contradicts the claim that we started with a justification closure, and the proposition follows.□

We will say that q is *directly connected to* p in a theory $\mathcal{T}$ if there is a justification in $\mathcal{T}$ of which q is an antecedent and p is the consequent. The 'connected to' relation is then the the least transitive relation containing the 'directly connected to' relation. p and q are *strongly* connected in $\mathcal{T}$ if p is connected to q in $\mathcal{T}$ and q is connected to p in $\mathcal{T}$. Finally, a *strongly connected component* of $\mathcal{T}$ is a subset of $\mathcal{L}$ such that every pair of elements in the subset is strongly connected. A maximal strongly connected component in $\mathcal{T}$ is one that is contained in no larger strongly connected component. Henceforth we will only consider maximal strongly connected components.

Proposition II.5 *Every monotonic TMS theory has a consistent, well-founded justification completion.*

Proof: That a TMS theory, $\mathcal{T}$, has a consistent justification completion is guaranteed by the previous proposition. Again we may restrict our attention to the justification closure. In order for the justification closure of such a theory *not* to be well-founded, one can readily verify that there must exist a maximal strongly connected component, $\mathcal{S}$, of $\mathcal{T}$ such that for every $p \in \mathcal{S}$:

1. $\mathcal{B}[p]$ is in the justification closure,
2. every justification in $\mathcal{T}$ has at least one antecedent in $\mathcal{S}$ if it has consequent p and for all of its antecedents, r, $\mathcal{B}[r]$ is in the justification closure.

Now observe that if every positive belief in the justification closure with core in $\mathcal{S}$ is replaced by the corresponding negative belief, the resulting set of formulae will still be a least fixed point of $\mathcal{C}_{\mathcal{T}}$. If we apply this replacement recipe to every $\mathcal{S}$ fitting the description that we gave above, we will be left with a well-founded, least fixed point of $\mathcal{C}_{\mathcal{T}}$.□

Proposition II.6 *Every monotonic TMS theory has a unique consistent well-founded justification completion.*

Proof: That a monotonic TMS theory, $\mathcal{T}$, has a consistent well-founded justification completion is already established by the propositions above. Our aim here is to establish uniqueness. Every theory $\mathcal{T}$ can be uniquely partitioned into strongly connected components. Moreover, we will say that one such component is below another just in case there is a p in the first connected to some q in the second. This relation obviously induces a unique partial order on strongly connected components in $\mathcal{T}$. Let $\mathcal{T}$ be the theory having the strongly connected component of smallest size such that $\mathcal{T}$ has two distinct well-founded justification completions, $\mathcal{S}_1$ and $\mathcal{S}_2$. Clearly some strongly connected component, Z, of least height in $\mathcal{T}$ must be such that the subsets of beliefs from each of the completions, $\mathcal{S}_1$ and $\mathcal{S}_2$, whose cores are exactly the elements of Z will also be distinct. We may presume that Z has nothing below it, for if it did, there would be a smaller theory $\mathcal{T}'$

that also had distinct well-founded justification completions. This new theory would be obtained from $\mathcal{T}$ by first deleting any justifications connecting strongly connected components below Z or justifications whose antecedents are in strongly connected components below Z. We would *add* a justification with consequent p and having no antecedents whenever there was a justification, j, in $\mathcal{T}$ such that $(\forall q \in \alpha(j))\mathcal{B}[q] \in \mathcal{S}_1$. There exists a $p \in Z$ such that $\mathcal{B}[p]$ is in one of those justification completions *and* $j = \mathcal{J}[p]$ is in $\mathcal{T}$. But this means that the theory $\mathcal{T}$ less any justification $j' \neq j$ with consequent p must also have the same two distinct justifications as $\mathcal{T}$. (Note that at least one such j' exists in order for $\mathcal{T}$ to have a single strongly connected component of more than one element.) This contradicts the assumption that $\mathcal{T}$ has the strongly connected component of smallest size while also having two distinct well-founded justification completions.□

The justification completion is meant to capture the constraint propagation processes implicit in the truth maintenance systems of Doyle and Goodwin. Justification completions of TMS theories, in contrast to the *deductive* closures of $\mathcal{L}$-theories, are meant to capture that which has been ***proven*** in contrast to that which is ***provable***. The correspondence between the syntactic notion of justification given above and the homonymous notion in the TMS's of Doyle and Goodwin will be apparent to readers familiar with those investigators' systems. Our aim here is for the justifications in $\mathcal{T}$, having no antecedents, to correspond to the *premisses* of a typical truth maintenance system. The elementary positive and negative beliefs in the justification completion are meant to correspond to the formulae labeled (respectively) "IN" and "OUT" by a TMS. Readers can readily verify that the TMS's of Doyle and Goodwin would label a node as "IN" with respect to a set of justifications only if the proposition associated with that node were the core of a positive belief in the justification completion of the corresponding TMS theory. Having given a syntactic characterization to truth maintenance by defining a formal logic of justified belief, we turn now to supplying a suitable semantics for that logic.

B. Semantics

In this section we will equip TMS logics with a possible world semantics [Bradley and Swartz, 1981, Chellas, 1980, Hughes and Cresswell, 1984, Hughes and Cresswell, 1968]. This semantics is slightly unusual in that there are two accessibility relations, one to give meaning to justifications (or *validity* in the terminology of Goodwin) and one to give meaning to well-foundedness: furthermore, the accessibility relation corresponding to justifications relates arbitrary (finite, ordered tuples of worlds in contrast to the usual ordered pairs of worlds. In order to capture the nonmonotonicity of TMS theories, we base our semantics on the idea of *minimal* models, a notion introduced by McCarthy [McCarthy, 1980] and Davis [Davis, 1980], further pursued by Bossu and Siegel [Bossu and Siegel, 1985], and ultimately explored and exploited by Shoham [Shoham, 1986]. Finally, it will develop that the computation carried out by a truth maintenance system will correspond to the construction of an appropriate model should such a structure exist.

An *interpretation*, $\mathcal{M}$, is a structure $\langle W, \pi, \mu, \prec\rangle$. We will subscript the various elements of a structure as required to avoid ambiguity of reference. $W = \{w_p | p \in \mathcal{L}\}$ is a set of *moments* of truth. u, v, w (possibly subscripted) will denote moments. $\prec \subseteq W^2$. π is of type $\pi: \mathcal{L} \to 2^W$. $\mathcal{M}$ satisfies $p \in \mathcal{L}$ at moment w, denoted $\mathcal{M}, w \models p$, just in case $(\pi(p))(w) = 1$. We impose an additional restriction on π that $w_p \prec w \Rightarrow (\pi(p))(w) = (\pi(p))(w_p)$. μ is of type $\mu: (\mathcal{L}_1 - \mathcal{B}[\mathcal{L}]) \to \bigcup_{m,n} 2^{W \times W^m \times W^n}$. In particular,

$$\mu(\mathcal{J}[p|q_1, \ldots, q_m|r_1, \ldots, r_n]) \in 2^{W \times W^m \times W^n}.$$

$\mathcal{M}$ satisfies $\mathcal{J}[p|q_1, \ldots, q_m|r_1, \ldots, r_n]$, denoted

$$\mathcal{M} \models \mathcal{J}[p|q_1, \ldots, q_m|r_1, \ldots, r_n],$$

just in case

1. $(\mu(\mathcal{J}[p|q_1, \ldots, q_m|r_1, \ldots, r_n]))(w_p, w_{q_1}, \ldots, w_{q_m}, w_{r_1}, \ldots, w_{r_n}) = 1$;
2. and if
   (a) $(\mu(\mathcal{J}[p|q_1, \ldots, q_m|r_1, \ldots, r_n]))(u, v_1, \ldots, v_m, w_1, \ldots, w_n) = 1$,
   (b) $\mathcal{M}, u_1 \models q_1, \ldots, \mathcal{M}, u_m \models q_m$, $\mathcal{M}, v_1 \not\models r_1, \ldots, \mathcal{M}, v_m \not\models r_n$,

   then $\mathcal{M}, u \models p$.

$\mathcal{M} \models \mathcal{B}[p]$ just in case $\mathcal{M}, w_p \models p$. An interpretation, $\mathcal{M}$, is said to be a *model* of a set of formulae $\mathcal{S} \subset \mathcal{L}_1$, denoted $\mathcal{M} \models \mathcal{S}$, if and only if it satisfies every formula in $\mathcal{S}$.

Let $\trianglelefteq$ be the least irreflexive, asymmetric, transitive relation on models of $\mathcal{T}$, a TMS theory, such that if $\mathcal{M}_1$ and $\mathcal{M}_2$ are models of $\mathcal{T}$ then the following criteria are met:

1. if $(\forall j)\mathcal{M}_1 \models j \Rightarrow \mathcal{M}_2 \models j$, then $\mathcal{M}_1 \trianglelefteq \mathcal{M}_2$;
2. if $\mathcal{M}_1$ and $\mathcal{M}_2$ are unordered with respect to the previous criterion and $(\forall p)\mathcal{M}_1 \models \mathcal{B}[p] \Rightarrow \mathcal{M}_2 \models \mathcal{B}[p]$, then $\mathcal{M}_1 \trianglelefteq \mathcal{M}_2$;
3. if $\mathcal{M}_1$ and $\mathcal{M}_2$ are unordered with respect to the previous criterion and

$$\begin{aligned}((\forall p)\mathcal{M}_1 \models \mathcal{B}[p] \Rightarrow ((\exists j)p = \kappa(j) \\ \wedge (\forall q \in \alpha(j))\mathcal{M}_1 \models \mathcal{B}[q] \wedge q \prec_{\mathcal{M}_1} p \\ \wedge (\forall r \in \overline{\alpha}(j))\mathcal{M}_1 \not\models \mathcal{B}[r])) \\ \wedge \\ (\neg(\forall p)\mathcal{M}_2 \models \mathcal{B}[p] \Rightarrow ((\exists j)p = \kappa(j) \\ \wedge (\forall q \in \alpha(j))\mathcal{M}_2 \models \mathcal{B}[q] \wedge q \prec_{\mathcal{M}_2} p \\ \wedge (\forall r \in \overline{\alpha}(j))\mathcal{M}_2 \not\models \mathcal{B}[r]))\end{aligned}$$

   then $\mathcal{M}_1 \trianglelefteq \mathcal{M}_2$;
4. if $\mathcal{M}_1$ and $\mathcal{M}_2$ are unordered with respect to the previous criteria and $\prec_{\mathcal{M}_1}$ is a graded partial order while $\prec_{\mathcal{M}_2}$ is not, then $\mathcal{M}_1 \trianglelefteq \mathcal{M}_2$;

5. if $\mathcal{M}_1$ and $\mathcal{M}_2$ are unordered with respect to the previous criteria and $\prec_{\mathcal{M}_1}$ is a subrelation of $\prec_{\mathcal{M}_2}$, then $\mathcal{M}_1 \trianglelefteq \mathcal{M}_2$;
6. if $\mathcal{M}_1$ and $\mathcal{M}_2$ are unordered with respect to the previous criteria and $(\forall p, w)(\pi_{\mathcal{M}_1}(p))(w) = 1 \Rightarrow (\pi_{\mathcal{M}_2}(p))(w) = 1$ or

$$(\forall j, u, v_1, \ldots, v_m, w_1, \ldots, w_n)$$
$$(\mu_{\mathcal{M}_1}(j))(u, v_1, \ldots, v_m, w_1, \ldots, w_n) = 1$$
$$\Rightarrow (\mu_{\mathcal{M}_2}(j))(u, v_1, \ldots, v_m, w_1, \ldots, w_n) = 1$$

then $\mathcal{M}_1 \trianglelefteq \mathcal{M}_2$.

A *TMS model* of $\mathcal{S}$ is a minimal model in the partial order $\triangleleft$. Intuitively, a TMS model says that

- as few justifications as possible are satisfied,
- as few elementary positive beliefs as possible are satisfied,
- as many consequent positive beliefs as possible are ordered with respect to the monotonic antecedents of some justification,
- at most one argument justifying any given elementary belief is offered in an "explanation", $\prec$.

Theorem II.1 *A TMS theory, $\mathcal{T}$, has a consistent (well-founded) justification completion, $\mathcal{S}$, if and only if it has a TMS model, $\mathcal{M}$, (with graded partial order, $\prec_{\mathcal{M}}$) such that* $(\forall x \in \mathcal{L}_1)\mathcal{M} \models x \Leftrightarrow x \in \mathcal{S}$.

Proof.

$\Rightarrow$: Suppose $\mathcal{T}$ and $\mathcal{S}$ are as in the statement of the theorem. By proposition II.3 we may assume without loss of generality that if we have a graded partial order for $\mathcal{S}$, it is minimal. We construct a TMS model as follows:

$$(\pi_{\mathcal{M}}(p))(w) = \begin{cases} 1 \text{ if } w = w_p \text{ and } \mathcal{B}[p] \in \mathcal{S}, \\ 0 \text{ otherwise.} \end{cases}$$

If $j = \mathcal{J}[p|q_1, \ldots, q_m|r_1, \ldots, r_n]$ then

$$(\mu_{\mathcal{M}}(j))(u, v_1, \ldots, v_m, w_1, \ldots, w_n) = \begin{cases} 1 \text{ if } j \in \mathcal{T} \text{ and} \\ \quad u = w_p, \\ \quad v_1 = w_{q_1}, \ldots, v_m = w_{q_m}, \\ \quad w_1 = w_{r_1}, \ldots, w_1 = w_{r_n}, \\ 0 \text{ otherwise.} \end{cases}$$

If there is a graded partial order, $<_{\mathcal{S}}$, then $w_p \prec_{\mathcal{M}} w_q \Leftrightarrow p <_{\mathcal{S}} q$. The fact that $\mathcal{M}$ is a minimal TMS model follows immediately.

$\Leftarrow$: Let $\mathcal{M}$ be a minimal model of $\mathcal{T}$ such that $(\forall x \in \mathcal{L}_1)\mathcal{M} \models x \Leftrightarrow x \in \mathcal{S}$. The consistency of $\mathcal{S}$ follows immediately from the existence of $\mathcal{M}$ and the definition of satisfiability. Any graded partial order $\prec_{\mathcal{M}}$ on $W_{\mathcal{M}}$ induces a similar partial order on the positive beliefs of $\mathcal{S}$.□

Corollary II.1 *If $\mathcal{M}_1$ and $\mathcal{M}_2$ are distinct TMS models of a justification completion of a TMS theory, they differ only in their ordering relations.*

Proof. It is immediate from the last theorem that if $\mathcal{S}$ is the justification completion in question, that $(\forall x \in \mathcal{L}_1)\mathcal{M}_1, \mathcal{M}_2 \models x \Leftrightarrow x \in \mathcal{S}$. Since $\mathcal{M}_1$ and $\mathcal{M}$ are minimal $\pi_{\mathcal{M}_1} = \pi_{\mathcal{M}_2}$ and $\mu_{\mathcal{M}_1} = \mu_{\mathcal{M}_2}$. Thus, the two models can only differ on their partial orders.□

III. Assumption-Based Truth Maintenance

The TMS's of Doyle and Goodwin are termed *justification-based.* An alternative model of truth maintenance is the *assumption-based* approach introduced by de Kleer. We address assumption-based truth maintenance by reduction to an equivalent justification-based model.

A. ATMS

Let $\mathcal{A} = \{A_1, \ldots, A_N\}$ be a finite set of *assumptions.* A label is any subset of the set of assumptions. The language, $\mathcal{L}$, will be as before. The language $\mathcal{L}_2$ is $\mathcal{L}_1$ excluding nonmonotonic justifications. An ATMS theory is a set,

$$\Theta = \{\mathcal{T}_X | \mathcal{T}_X \text{ is a monotonic TMS theory and } \mathcal{A} \supseteq Y \supseteq X \Rightarrow \mathcal{T}_X \subseteq \mathcal{T}_Y\}.$$

The *mutual justification completion* of Θ is *the* set

$$\{\mathcal{S}_X | \mathcal{S}_X \text{ is the justification completion of } \mathcal{T}_X \in \Theta\}.$$

Θ together with the relevant justification closure operators constitutes a logic of justified belief. Each of the consistent justification completions, $\mathcal{S}_X$, exists since the corresponding $\mathcal{T}_X$ is over a monotonic TMS theory. Interpreting a set of assumptions as an "environment", in de Kleer's sense of the word, an element of a mutual justification completion tells us what formulae are "IN" the corresponding environment. The ATMS model of an ATMS theory then is merely the set of TMS models:

$$\{\mathcal{M}_X | \mathcal{M}_X \text{ is a TMS model of } \mathcal{T}_X \in \Theta\}.$$

Immediate from the various propositions and theorem II.1 we have

Corollary III.1 *An ATMS theory has a unique consistent mutual justification completion. Moreover, the theory has a unique ATMS model such that* $(\forall x \in \mathcal{L}_2)\mathcal{M}_X \models x \Leftrightarrow x \in \mathcal{S}_X$.

To see how an ATMS theory arises, consider the set of nodes and justifications from de Kleer [de Kleer, 1986a, pages 150–151]:

$$\gamma_{x+y=1}: \langle x + y = 1, \{\{A, B\}, \{B, C, D\}\}, \{\ldots\}\rangle,$$
$$\gamma_{x=1}: \langle x = 1, \{\{A, C\}, \{D, E\}\}, \{\ldots\}\rangle,$$
$$\gamma_{x+y=1}, \gamma_{x=1} \Rightarrow \gamma_{y=0}.$$

The underlying logical language, $\mathcal{L}$, seems to be the language of algebraic equations. The set of assumptions is $\{A, B, C, D, E\}$. Let Θ be defined as above with each contained TMS theory being as small as possible and

$$\begin{aligned} \mathcal{T}_{\{A,B\}} &= \{\mathcal{J}[x+y=1]\}, \\ &= \mathcal{T}_{\{B,C,D\}}, \\ \mathcal{T}_{\{A,C\}} &= \{\mathcal{J}[x=1]\}, \\ &= \mathcal{T}_{\{D,E\}}, \\ \mathcal{T}_{\{\}} &= \{\mathcal{J}[y=0, x+y=1, x=1]\}. \end{aligned}$$

The reader can readily verify that $\mathcal{B}[y=0]$ will be in the justification completion of every $\mathcal{T}_X \in \Theta$ such that $\{A, B, C, D, E\} \supseteq X$ and either $X \supseteq \{A, B, C\}$, $X \supseteq \{A, B, D, E\}$, or $X \supseteq \{B, C, D, E\}$, exactly the desired result.*

B. ANRMS

In [Brown *et al.*, 1987, Gaucas and Brown, 1987] we introduced a new model of truth maintenance based on solving equations over Boolean lattices that subsumes both TMS and ATMS styles of truth maintenance. Indeed, in the cited references we show how to embed to the Doyle/Goodwin and de Kleer styles of truth maintenance system in ANRMS. Also, we give an algebraic characterization of the reduction of an ANRMS to a collection of TMS's which exactly mirrors the reduction of an ANRMS theory (below) to a collection of TMS theories. The Assumption-based Nonmonotonic Reasoning System (ANRMS) has an associated logic of justified belief analogous to that we have associated with the ATMS. Let $\mathcal{B}$ be a boolean lattice with the usual operations of meet, join, and complement (denoted $\sqcap$, $\sqcup$ and $^-$), distinguished constants, top ($\top$) and bottom ($\perp$), and a partial order ($\sqsubseteq$). An ANRMS theory is a set,

$$\Sigma = \{\mathcal{T}_X | \mathcal{T}_X \text{ is a TMS theory} \wedge \top \sqsupseteq Y \sqsupseteq X \Rightarrow \mathcal{T}_X \subseteq \mathcal{T}_Y\}.$$

The mutual justification completions and models of Σ are defined analogously to those for Θ above. Since a model for an ANRMS theory is just the collection of models associated with a collection of TMS theories, theorem II.1 generalizes directly.

IV. Conclusions

We have defined a collection of logical theories and associated them with various models of truth maintenance. We have identified the truth maintenance concepts of premiss, assumption, justification, node, "IN" and "OUT" with certain syntactic constructs in those logics. We have characterized the proof theories of those logics in terms of a justification closure operator and its fixed points. Each instance of a given logic is uniquely identified with set of premisses and justifications in a corresponding model of truth maintenance (and *vice versa*). We have given a semantical account of these logics in terms of minimal models. As it turns out, the labeling process carried out by a truth maintenance system corresponds to the construction of a minimal (collection of) model(s) (each) with a graded partial order should one (they) exist.

*Notice that we need take no explicit heed of de Kleer's *nogood* mechanism as this is entirely an apparatus for avoiding unnecessary computation. That is, it is deemed unnecessary to compute the justification closure of a TMS theory, $\mathcal{T}_X$, if for some $Y \subseteq X$ the justification closure of $\mathcal{T}_Y$ has both $\mathcal{B}[p]$ and $\mathcal{B}[\neg p]$ for some $p \in \mathcal{L}$.

References

[Benanav *et al.*, Forthcoming] Dan Benanav, Allen L. Brown, Jr., and Dale E. Gaucas. *A Lattice-Theoretic Framework for Reason Maintenance.* Technical Report, General Electric Corporate Research and Development Center, Schenectady, New York, Forthcoming.

[Bossu and Siegel, 1985] Genevieve Bossu and Pierre Siegel. Saturation, non-monotonic reasoning, and the closed world assumption. *Artificial Intelligence*, 25(1):13–64, January 1985.

[Bradley and Swartz, 1981] Raymond Bradley and Norman Swartz. *Possible Worlds: an Introduction to Logic and Its Philosophy.* Hackett, Indianapolis, 1981.

[Brown, 1985] Allen L. Brown, Jr. Modal propositional semantics for reason maintenance systems. In *Proc. 9th Int. Joint Conf. on Artificial Intelligence*, pages 178–184, Los Angeles, 1985.

[Brown *et al.*, 1987] Allen L. Brown, Jr., Dale E. Gaucas, and Dan Benanav. An algebraic foundation for truth maintenance. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1987.

[Chellas, 1980] Brian F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, London, 1980.

[Davis, 1980] M. Davis. The mathematics of non-monotonic reasoning. *Artificial Intelligence*, 13:73–80, 1980.

[de Kleer, 1984] Johan de Kleer. Choices without backtracking. In *Proc. 4th Nat. Conf. on Artificial Intelligence*, pages 79–85, Austin, 1984.

[de Kleer, 1986a] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[de Kleer, 1986b] Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, 28:163–196, 1986.

[de Kleer, 1986c] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28:197–224, 1986.

[Doyle, 1978] Jon Doyle. *Truth Maintenance Systems for Problem Solving.* Technical Report AI-TR-419, Massachussetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, January 1978.

[Doyle, 1979a] Jon Doyle. A glimpse of truth maintenance. In *Proc. 6th Int. Joint Conf. on Artificial Intelligence*, pages 232–237, 1979.

[Doyle, 1979b] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Gaucas and Brown, 1987] Dale E. Gaucas and Allen L. Brown, Jr. A role for assumption-based and non-monotonic justifications in automating strategic threat analysis. In *Proceedings of the Expert Systems Workshop*, Defense Advanced Research Projects Agency, St. Louis, April 1987.

[Gettier, 1967] Edmund L. Gettier. Is justified true belief knowledge? In A. Phillips Griffiths, editor, *Knowledge and Belief*, chapter X, Oxford University Press, 1967.

[Goodwin, 1982] James W. Goodwin. *An Improved Algorithm for Non-Monotonic Dependency Update*. Technical Report LITH-MAT-R-82-23, Linkoeping University, Linkoeping, Sweden, August 1982.

[Goodwin, 1984] James W. Goodwin. A theory and system for non-monotonic reasoning. In *Proc. of the Workshop on Non-Monotonic Reasoning*, pages 103–104, New Paltz, October 1984.

[Goodwin, 1985] 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.

[Goodwin, 1987] James W. Goodwin. *WATSON: A Dependency Directed Inference System*. PhD thesis, Linkoeping University, Linkoeping, Sweden, 1987.

[Griffiths, 1967] A. Phillips Griffiths. On belief. In A. Phillips Griffiths, editor, *Knowledge and Belief*, chapter IX, Oxford University Press, 1967.

[Hughes and Cresswell, 1968] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic*. Methuen, London, 1968.

[Hughes and Cresswell, 1984] G.E. Hughes and M.J. Cresswell. *A Companion to Modal Logic*. Methuen, London, 1984.

[Malcolm, 1967] Norman Malcolm. Knowledge and belief. In A. Phillips Griffiths, editor, *Knowledge and Belief*, chapter V, Oxford University Press, 1967.

[McCarthy, 1980] John McCarthy. Circumscription: a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[Perlis, 1984] Donald Perlis. Bibliography of literature of non-monotonic reasoning. In *Proc. of the Workshop on Non-Monotonic Reasoning*, pages 396–401, New Paltz, New York, October 1984.

[Prichard, 1967] H.A. Prichard. Knowing and believing. In A. Phillips Griffiths, editor, *Knowledge and Belief*, chapter IV, Oxford University Press, 1967.

[Shoham, 1986] Yoav Shoham. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence*. PhD thesis, Yale University, New Haven, Connecticut, 1986.

Using T-norm Based Uncertainty Calculi in a Naval Situation Assessment Application

Piero P. Bonissone
GE Corporate Research and Development Center
1 River Road, K1-5C32A, Schenectady, New York 12301
Phone: 518-387-5155 Arpanet: Bonissone@GE-CRD

ABSTRACT

RUM, the Reasoning with Uncertainty Module described in the previous paper, has been tested and validated in a sequence of experiments in both naval and aerial situation assessment (SA). The purpose of these experiments is to exercise and evaluate RUM's reasoning capabilities in correlating sensor reports and tracks, locating and classifying platforms, and identifying intents and threats. An example of naval situation assessment is illustrated.

The testbed environment for developing these experiments has been provided by LOTTA, a symbolic simulator implemented in Zetalisp Flavors. This simulator maintains time-varying situations in a multi-player antagonistic game where players must make decisions in light of uncertain and incomplete data. RUM has been used to assist one of the LOTTA players to perform the SA task.

1. Introduction

In the previous paper we have described RUM, the Reasoning with Uncertainty Module whose layered architecture reflects the typical structure of automated reasoning techniques [Bonissone 1986;87a].

In this paper we will illustrate the naval situation assessment problem which was used to validate RUM. This application is based on an architecture designed to simulate various military scenarios involving Multi-Sensors/Multi-Targets (MS/MT) and to perform situation assessment (SA) related tasks. The MS/MT architecture, illustrated in Figure 1, is composed of two major blocks: a reasoning system and a simulation environment.

The first block of the MS/MT architecture, the reasoning system, is based on RUM and has already been described in the preceding paper. The second block of the MS/MT architecture, the simulation environment, is described in section 2. In section 3, we provide some definitions of the tasks required to perform situation assessment. The last two sections contain an analysis of the MS/MT experiment and some preliminary conclusions on this work.

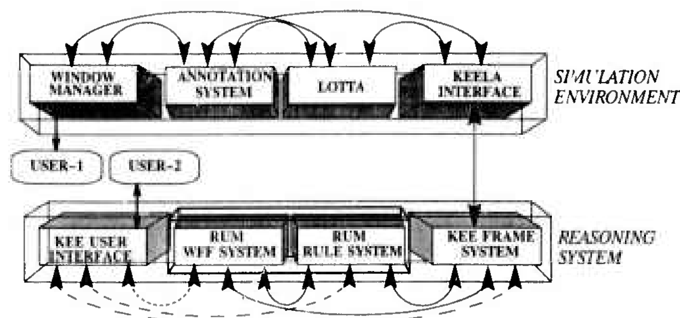


Figure 1. Architecture for Multi-Sensors/Multi-Targets (MS/MT)

2. An Object Based Simulation Environment

The second block of the MS/MT architecture is the simulation environment. This environment is centered around LOTTA, an object-oriented symbolic battle management simulator that maintains time-varying situations in a multi-player antagonistic game [Bonissone 1987b]. The development environment based on LOTTA constitutes a testbed for validating new techniques in reasoning with uncertainty and for performing information fusion functions [Sweet 1986]. The development environment is composed of four basic modules. the *window manager*, the *annotation system*,

The window manager is a map-like window-oriented user interface. It controls the menu driven interaction of the human player with LOTTA and handles multiple windows per player.

The annotation system is an intelligent database for LOTTA. It is composed of a feature extraction system and a feature watcher. The feature extraction system allows both simple and complex time-varying features to be calculated and stored (along with the features or parameters that they depend on and the methods to update them over time). Every feature, internally or externally computed, has multiple views (numerical and graphical representations to allow either people or computer programs to use them in decision-making or explanation tasks). The feature watcher maintains the dependency directed information that characterizes the dynamic support of the features. The watcher will guide the "lazy" recomputations of those features whose support has changed since the last computation.

* This is a modified version of the paper *Using T-norm Based Uncertainty Calculi in a Naval Situation Assessment Application* that will appear in the Proceedings of the Third AAAI Workshop on Uncertainty in Artificial Intelligence, Seattle, Washington, July 1987.

This work was partially supported by the Defense Advanced Research Projects Agency (DARPA) under USAF/Rome Air Development Center 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.

Figure 2 shows a split screen in which two players, Blue and Orange, using the Window Manager and the Annotation System can observe the location of their own units, the limits of the territorial waters (jagged line) and the shipping lane that is vertically crossing the screen. One of the players (Blue) can also see the coverage provided by his surface radar. The coverage is represented by a changing gray level distribution that represents the probability of detecting any other unit in that range.

LOTTA is the simulator that executes commands and maintains internal states. Each simulation unit is a Flavor object that is a node in a Flavors graph. Message passing is the uniform communication paradigm for sending commands and modifying the internal states of the objects. A simulation cycle corresponds to a real-time variable that is common to all the players. The simulation cycle is divided into 12 phases: GAME-SYNCH, SENSOR (initialization, send, receive, ECM), MOVEMENT, SENSOR (initialization, send, receive, ECM), COMBAT (CIDS, offensive).

KEELA (KEE to LottA interface) links LOTTA with KEE, the expert system shell that provides the capabilities to write and execute the rules describing the policies of the player. A new rule system has been implemented in KEE, to allow the representation, use, and control of the different uncertainty calculi. KEELA is based on FLUTE (FLavor to Units Translation Environment). FLUTE transforms a Flavors graph, such as the LOTTA objects graph, into a graph of KEE Units, with their corresponding slots and facets. This generates a "vocabulary" (names and structures) of the objects in LOTTA. This transformation enables the programmer to use the KEE browser to generate a pictorial representation of the graph, providing an aid for debugging and documentation. Utility functions for display/explanation link KEE to LOTTA's Feature and Window Managers.

3. The Information Fusion/Situation Assessment Problem

The Information Fusion (IF)/Situation Assessment (SA) requires a variety of tasks in which uncertainty pervades both the input data and the knowledge bases. Beside its intrinsic uncertainty, usually the information dealt in each task is also incomplete, time-varying, and, sometimes, erroneous. Thus, the SA problem represents a strong challenge for most automated reasoning systems, since it requires an integration of the uncertainty management with a truth maintenance system (belief revision system) to maintain the integrity of the inference base (or of its relevant subset). The SA problem also requires the reasoning system to detect useless and contradicting information, rejecting the former and resolving the latter.

There is no uniformly agreed definition of what a situation assessment problem entails. The following definitions have been compiled and summarized from a variety of sources [Levitt 1984], [Clarkson 1981] to succinctly describe the SA problem. Given a platform (aircraft, ship, tank) in a potentially hostile environment, the process of performing Situation Assessment consists of the following tasks:

1. Sensor data must be collected from various sources and described as reports.

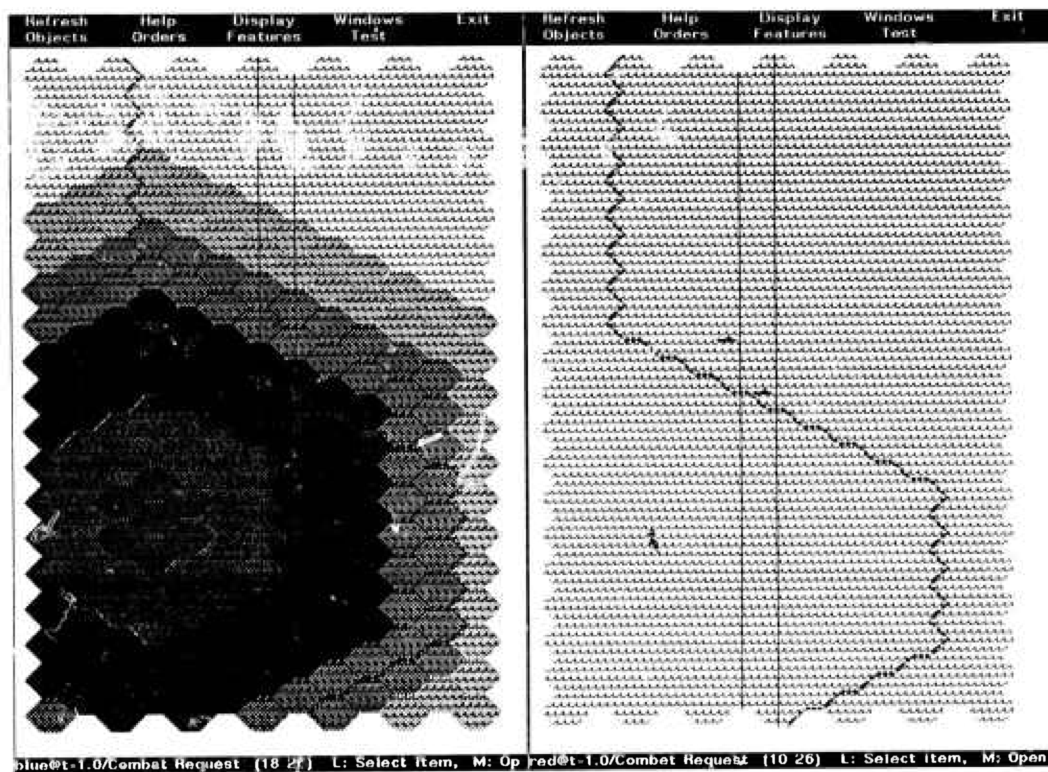


Figure 2. Window Manager and Annotation System Based Display

2. Time-stamped sensor reports must be consolidated into tracks (each track is the trace of an object followed by a given sensor).
3. Tracks associated to the same object must be fused into a platform.
4. The detected platform must be classified and identified (by class and type).
5. Node organization (formation of the identified platforms), use of special equipment, and maneuvering must be recognized.
6. Using the knowledge of the opponent's doctrines and rules of engagement, the recognized formation and observed use of special equipment must be explained by a probable intent, which is then translated into a threat assessment (retrospective SA).
7. This analysis is then projected into the future to evaluate plausible plans and to determine likely interesting developments of the current situation (prospective SA).

The first four tasks constitute what is generally known as Information Fusion [Dillard 1978; 83] and define the scope of the first MS/MT experiment.

3.1 Example of RUM rules

The RUM knowledge base (KB) used in MS/MT application is composed of approximately forty rules, each of which can be instantiated by new sensor reports, new tracks, or new platforms. A representative sample of such a KB is provided by the following two rules.

English Version of Rule-500 (identifying submarines):

Assuming that a radar was used to generate a sensor report (that with other reports generated by the same sensor has been attached to a track associated with a platform), if the first time that the platform was detected (in the track's first report), the platform was located at a distance of at most twenty miles from our radar (i.e., it was a close-distance radar pop-up) then it is most likely that the platform is a submarine. Otherwise, there is a small chance that it is not a submarine.

RUM's Version of the same rule:

```
(add-template 'sub.pos.id-
   close.pop.up-500                      ; Name
   'msmt                                 ; KB
   '((u-lessp (get.uncertain.value
      (get.value ?track 'first.report)
      'range) (fuzz 20)))                ; Premise-list
   (((get.value ?track 'platform)
      class.name submarine s2.rules))    ; Consequence-list
   '((?track first.report))              ; List of wffs in premise
   '(?track)                             ; List of units in premise
   '((is-in-class? (get.value ?report
      'track) 'source '(radar lotta))    ; Context
   '(most.likely small.chance)           ; Sufficiency and necessity
   't3                                   ; Aggregation T-norm
   '(submarine track.templates))         ; Rule class &
                                           instantiation templ.
```

English Version of Rule-550 (identifying submarines):

Assuming that a sonar was used to generate a sensor report (that with other reports generated by the same sensor has been attached to a track associated with a platform), if the detected platform has a low noise emission, and is located at a depth of at least twenty meters, then it is extremely likely that it is a submarine. Otherwise, it may not be a submarine.

RUM's Version of the same rule:

```
(add-template 'sub.pos.id-sonar-550      ; Name
   'msmt                                 ; KB
   '((is-value? ?report 'noise-emissions
      'low)                              ; Premise-list
   (u-lessp (get.uncertain.value ?report
      'elevation) (fuzz-20)))
   '((?report elevation))                ; List of wffs in premise
   '(?report)                            ; List of units in premise
   '((is-in-class? (get.value ?report
      'track) 'source '(sonar lotta)))   ; Context
   '(extremely.likely it.may)            ; sufficiency and necessity
   't3                                   ; Aggregation T-norm
   '(submarine report.templates))        ; Rule class &
                                           instantiation templ.
```

3.2 Notes on the Calculi Selection for Rule 500 and 550

The T-norm used to detach the conclusion of rule 500 and 550 is T_3. This is due to the fact that we want to obtain the smallest certainty interval associated with the detached conclusion. The T-conorm used to aggregate the certainties of the detachments of both rules is S_2. This assignment indicates a lack of correlation among the two rules, which is substantiated by the fact that independent sources of information (radar and sonar) are used in the context of the two rules.

4. The Experiment

In the experiment, a modified version of the naval situation assessment scenario used by NOSC to test STAMMER and STAMMER2 [McCall 1979, Bechtel 1979, Ferranti 1981] was created. In this modified scenario, a missile cruiser of the type CGN36 operating with a surface radar (SPS 10) and a passive sensor (GPS-3) faced two platforms (selected from a set of possible platform classes such as cruisers, destroyers, frigates, patrol hydrofoils, submarines, merchant ships, and fishing boats). One of the two platforms was using an active sensor (navigational radar), while the second platform was not using any sensor.

The cruiser's task was to track, correlate, and classify each detected object. Both passive and active sensors on the cruiser were run twice (during one LOTTA cycle), generating sensor reports which were translated through the KEELA interface into observed *wffs*. The reports were then grouped into tracks for each sensor. A total of three tracks were generated: two tracks were produced by the cruiser's active sensor and one by its passive sensor. Plausible correlations were made among the tracks to correctly group them into the two detected platforms. Figure 3 graphically illustrates a portion of the knowledge base where the report, track, and platform information is stored. In the same figure it is possible to ob-

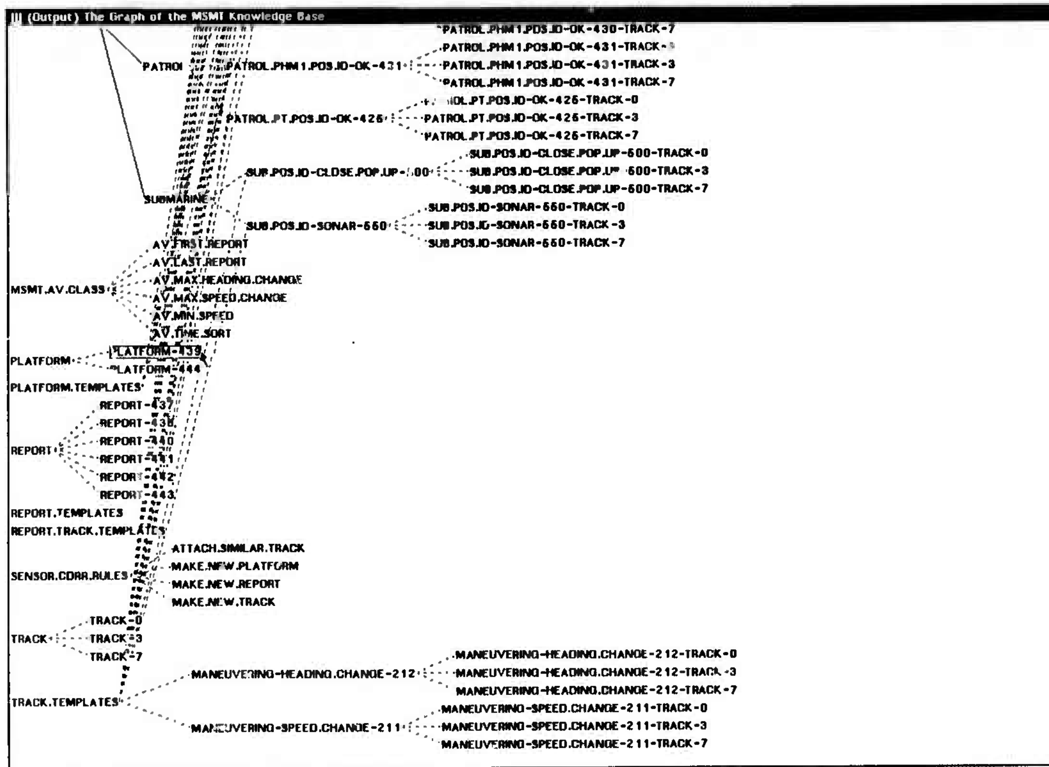


Figure 3. Subgraph of the MSMT Knowledge Base

serve the rule instantiation (by track) of the two rules (500 and 550) described in section 3.1.

Figure 4 shows the sensor report information generated by running a sensor once. The parameters obtained from this particular passive sensor (GPS-3) were the heading, position, range, speed, and time at which the platform was detected. This information was attached to a track (TRACK-0) which maintained subsequent sensor reports generated by the same sensor (GPS-3) and associated with the same platform (PLATFORM-439). Figure 5 describes the track information.

Another track (TRACK-3) was generated by using a second sensor (SPS-10). The information from both tracks was attached to the same platform. The query posed to RUM was to deduce the class value of such platform from the tracks information. Using the RUM knowledge base and the backward chaining mode, various attributes of the platform were inferred or observed The platform was correctly identified as a merchant ship. This conclusion was made by a set of rules based on the fact that the platform was reasonably close to a shipping lane, it was traveling at a a typical merchant's speed (in the 9-14 miles/hour range), it was not maneuvering, nor was it trying to dodge the cruise's surface radar. Figure 6 shows the attributes of Platform-439, which were required to determine the platform's class value, as well as the class value itself.

Figure 7 shows the uncertainty information and meta-information associated with the value assignments to the variable [*Platform-439 Class-name*]. In the slot VALUES, we can see the platform classes which were considered by the system and their corresponding certainty bounds:

Merchant [.69 1], *Submarine* [0 .2], *Fishing Boat* [0 .02]

The best value in terms of its certainty is clearly the one which identifies Platform-439 as a *Merchant.* Its certainty's lower bound indicates a reasonably large amount of positive (confirming) evidence. Its upper bound indicates the absence of any negative (refuting) evidence. The class *Submarine* obtained no confirming evidence and a large amount of negative evidence. The refuting evidence was provided by rule 500, which from the failure to observe a *close-distance radar pop-up* determined that there was only a small chance for the platform to be a submarine. The class *Fishing Boat* also had no confirming evidence and an overwhelming amount of negative evidence. This refuting evidence was due to the fact that the platform was too far from the fishing areas, too big, and was using a radar (rules 340, 320, and 330). This information can be obtained from Figure 7, by observing the logical support for each value, and from Figure 8, hy observing the dominant rules for each value.

The same Figure 7 shows the logical support for each of the three values considered for the *wff* [*Platform-439 Class-name*]. Each rule instance, fired to infer a value of the *wff*, has a cached certainty value (lower and upper bounds) and an associated validity flag. This was illustrated in Figure 4 of the previous paper, where each rule instance was schematically identified as a gate in an acyclic graph. The rule instances' cached certainty values are illustrated in Figure 8 and show which rules dominated the certainty assignment for each of the three values.

```
||| (Output) The REPORT-440 Unit in MSMT Knowledge Base
Unit: REPORT-440 in knowledge base MSMT
Created by Leaning Jowler on 4-22-87 11:31:46
Modified by Leaning Jowler on 4-22-87 10:31:51
  Member Of: REPORT

  Created Automagically by Process.Image

Own slot: ELEVATION from REPORT-440
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: UNBOUND

Own slot: HEADING from REPORT-440
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: 180

Own slot: LOTTA.IMAGE from REPORT-440
   Inheritance: VARIABLE.VALUES
   ValueClass: UNIT
   Values: #<IMAGE Hawkeye 8818671> in LOTTA

Own slot: NOISE-EMISSIONS from REPORT-440
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: UNBOUND

Own slot: POSITION from REPORT-440
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: (51 95)

Own slot: RANGE from REPORT-440
   Inheritance: VARIABLE.VALUES
   ValueClass: NUMBER
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: 23.43075

Own slot: SIGNAL-STRENGTH from REPORT-440
   Inheritance: VARIABLE.VALUES
   Cardinality.Max: 1
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: UNBOUND

Own slot: SPEED from REPORT-440
   Inheritance: VARIABLE.VALUES
   ValueClass: NUMBER
```

```
||| The REPORT-440 Unit in MSMT Knowledge Base
Own slot: SPEED from REPORT-440
   Inheritance: VARIABLE.VALUES
   ValueClass: NUMBER
   Cardinality.Max: 1
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: 15

Own slot: TIME from REPORT-440
   Inheritance: VARIABLE.VALUES
   ValueClass: NUMBER
   Cardinality.Max: 1
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: 8.5

Own slot: TRACK from REPORT-440
   Inheritance: VARIABLE.VALUES
   ValueClass: TRACK
   Values: TRACK-0
```

Figure 4. REPORT-440 Attached to TRACK-0

```
||| The TRACK-0 Unit in MSMT Knowledge Base
Unit: TRACK-0 in knowledge base MSMT
Created by Leaning Jowler on 4-22-87 11:31:46
Modified by Leaning Jowler on 4-22-87 10:31:57
  Member Of: TRACK

  Created by Process.Track

Own slot: FIRST.REPORT from TRACK-0
   Inheritance: VARIABLE.VALUES
   ValueClass: REPORT
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Comment: Computed
   Values: REPORT-441

Own slot: LAST.REPORT from TRACK-0
   Inheritance: VARIABLE.VALUES
   ValueClass: REPORT
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Comment: Computed
   Values: REPORT-440

Own slot: LOTTA.TRACK from TRACK-0
   Inheritance: VARIABLE.VALUES
   ValueClass: UNIT
   Values: #<TRACK Hawkeye 8818688> in LOTTA

Own slot: MAX.HEADING.CHANGE from TRACK-0
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Comment: Computed
   Values: 0

Own slot: MAX.SPEED.CHANGE from TRACK-0
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Comment: Computed
   Values: 0

Own slot: MIN.SPEED from TRACK-0
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Comment: Computed
   Values: 15

Own slot: PLATFORM from TRACK-0
   Inheritance: VARIABLE.VALUES
   ValueClass: PLATFORM
   Values: PLATFORM-439
```

```
||| (Output) The TRACK-0 Unit in MSMT Knowledge Base
Own slot: REPORTS from TRACK-0
   Inheritance: VARIABLE.VALUES
   Avunits: AV.TIME.SORT, AV.FIRST.REPORT, AV.LAST.REPORT,
        AV.MAX.HEADING.CHANGE, AV.MAX.SPEED.CHANGE, AV.MIN.SPEED
   Values: (REPORT-441 REPORT-440)

Own slot: SOURCE from TRACK-0
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: GPS-3
```

Figure 5. TRACK-0 Attached to PLATFORM-439

```
||| (Output) The PLATFORM-439 Unit in MSMT Knowledge Base
Unit: PLATFORM-439 in knowledge base MSMT
Created by Leaning Jowler on 4-22-87 11:31:45
Modified by Leaning Jowler on 4-22-87 10:32:36
  Member Of: PLATFORM

  Created by process.platform

Own slot: CLASS.NAME from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Comment: put the name of the class here, since KEE doesn't treat classes like slots.
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: MERCHANT

Own slot: DODGE.KNOWN.SENSORS from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: UNKNOWN

Own slot: DODGE.NEW.SENSORS from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Values: UNKNOWN

Own slot: ERRATIC.RADAR from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Values: UNKNOWN

Own slot: FOLLOW.BAD.WEATHER from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: UNKNOWN

Own slot: MANEUVERS from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: T

Own slot: RADAR from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: UNKNOWN

Own slot: TRACKS from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Values: TRACK-0, TRACK-3

Own slot: TYPE.NAME from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
```

```
||| The PLATFORM-439 Unit in MSMT Knowledge Base
Own slot: TYPE.NAME from PLATFORM-439
   Inheritance: VARIABLE.VALUES
   Avunits: AV.CERT.UNIT in HARDWARE
   Urole: ALL
   Values: UNKNOWN
```

Figure 6. Platform-439 Unit with Associated Attributes

```
||| (Output) The PLATFORM-439-CLASS.NAME Unit in MSMT-UNC Knowledge Base
Own slot: CONSTRAINTS.RULES from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   Values: FISHING.BOAT.POS.ID-OK-300-TRACK-3 in MSMT,
           FISHING.BOAT.POS.ID-OK-300-TRACK-0 in MSMT

Own slot: DEPENDENT.RULES from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   ValueClass: GENERIC.RULE.UNIT in HARDWARE
   Comment: another list of rules.
   Values: MERCHANT.TYPE.UNK-1000-PLATFORM-439 in MSMT,
           FISHING.TYPE.UNK-1010-PLATFORM-439 in MSMT

Own slot: DS.RULES from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   ValueClass: GENERIC.RULE.UNIT in HARDWARE
   Avunits: AV.BAD in HARDWARE
   Comment: Rules to be combined using the Dempster-Schaefer conorm.
   Values: UNKNOWN

Own slot: FLAG from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   Avunits: AV.FLAG in HARDWARE, AV.ALERT in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Good or X.
   Values: GOOD

Own slot: NECESSITY from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   Avunits: AV.POLLUTE in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Minimum support for a wff.
   Values: (0.6928104 0.816568 0.08169937 0.07845958)

Own slot: PLAUSIBILITY from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   Avunits: AV.POLLUTE in HARDWARE
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: Maximum support for a wff.
   Values: 1

Own slot: S1.RULES from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   ValueClass: GENERIC.RULE.UNIT in HARDWARE
   Avunits: AV.BAD in HARDWARE
   Comment: Rules that are to be disjuncted using S1.
   Values: UNKNOWN

Own slot: S15.RULES from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   ValueClass: GENERIC.RULE.UNIT in HARDWARE
```

```
||| The PLATFORM-439-CLASS.NAME Unit in MSMT-UNC Knowledge Base
Own slot: S2.RULES from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   ValueClass: GENERIC.RULE.UNIT in HARDWARE
   Avunits: AV.BAD in HARDWARE
   Comment: Rules to be disjuncted using S2.
   Values: SUB.POS.ID-CLOSE.POP.UP-500-TRACK-3 in MSMT,
           MERCHANT.NEG.ID-MANEUVERS-210-PLATFORM-439 in MSMT,
           MERCHANT.NEG.ID-DISTANT.POP.UP-220-TRACK-3 in MSMT,
           MERCHANT.NEG.ID-DISTANT.POP.UP-220-TRACK-0 in MSMT,
           SUB.POS.ID-CLOSE.POP.UP-500-TRACK-0 in MSMT,
           SUB.POS.ID-SONAR-650-TRACK-3 in MSMT,
           SUB.POS.ID-SONAR-650-TRACK-0 in MSMT

Own slot: S25.RULES from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   ValueClass: GENERIC.RULE.UNIT in HARDWARE
   Avunits: AV.BAD in HARDWARE
   Comment: Rules to be disjuncted using S2.5.
   Values: FISHING.BOAT.NEG.ID-USE.RADAR-330-TRACK-0 in MSMT,
           FISHING.BOAT.NEG.ID-TOO.FAST-310-TRACK-3 in MSMT,
           MERCHANT.POS.ID-OK-100-TRACK-0 in MSMT,
           MERCHANT.POS.ID-OK-100-TRACK-3 in MSMT,
           MERCHANT.NEG.ID-TOO.SLOW-140-TRACK-3 in MSMT,
           MERCHANT.NEG.ID-TOO.SLOW-140-TRACK-0 in MSMT,
           MERCH.NEG.ID-TOO.SMALL-200-TRACK-3 in MSMT,
           MERCH.NEG.ID-TOO.SMALL-200-TRACK-0 in MSMT,
           FISHING.BOAT.NEG.ID-TOO.FAST-310-TRACK-0 in MSMT,
           FISHING.BOAT.NEG.ID-USE.RADAR-330-TRACK-3 in MSMT

Own slot: S3.RULES from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   ValueClass: GENERIC.RULE.UNIT in HARDWARE
   Avunits: AV.BAD in HARDWARE
   Comment: Rules to be disjuncted using S3.
   Values: FISHING.BOAT.NEG.ID-TOO.FAR-340-TRACK-0 in MSMT,
           FISHING.BOAT.NEG.ID-TOO.FAR-340-TRACK-3 in MSMT,
           FISHING.BOAT.NEG.ID-TOO.BIG-320-TRACK-3 in MSMT,
           MERCHANT.NEG.ID-DODGE.STATIC.SENSOR-250-PLATFORM-439 in MSM
           T, MERCHANT.NEG.ID-BAD.WEATHER-260-PLATFORM-439 in MSMT,
           FISHING.BOAT.NEG.ID-TOO.BIG-320-TRACK-0 in MSMT

Own slot: VALUE from PLATFORM-439-CLASS.NAME
   Inheritance: OVERRIDE.VALUES
   Comment: Value of slot.
   Values: (SUBMARINE MERCHANT FISHING.BOAT),
           (((0 0 0 0) (0.19999999 0.37 0.06 0.05))
                ((0.6928104 0.816568 0.08169937 0.07845958) (1 1 0 0))
                ((0 0 0 0) (0.019801915 0.039215684 0.019802034 0.09162535)))
```

Figure 7. Uncertainty Unit Associated with wff [Platform-439 Class-name]

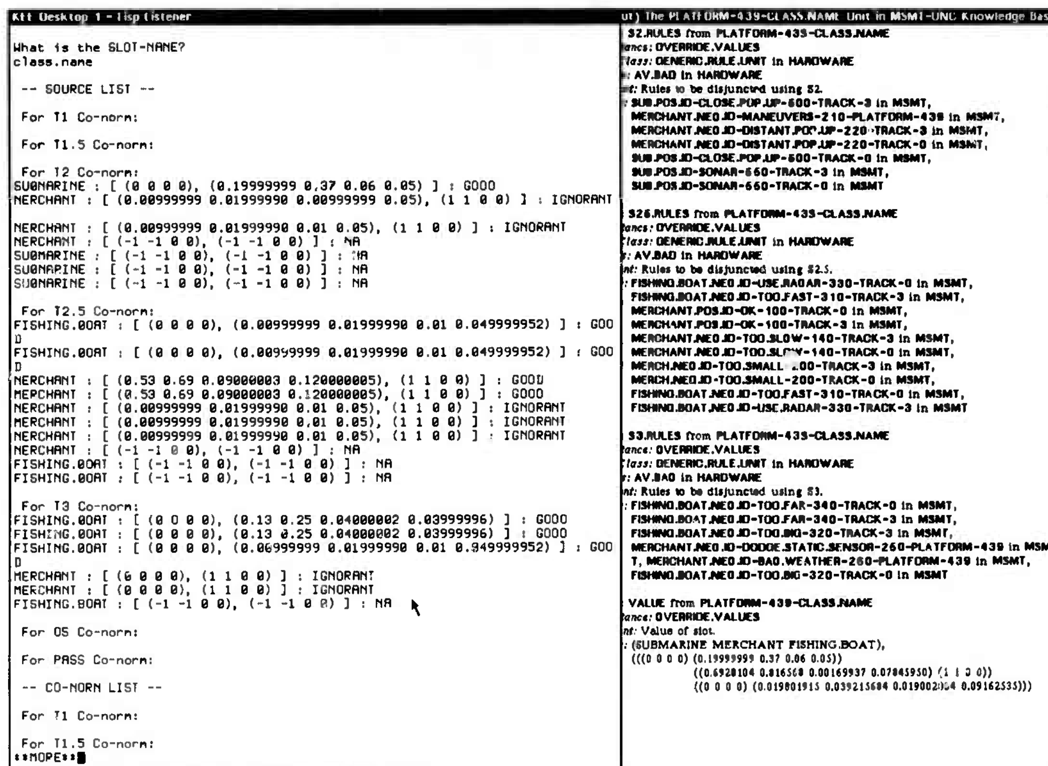


Figure 8. Uncertainty Bounds Detached by the Rule Instances in [Platform-439 Class.name] Logical Support

5. Remarks and Conclusions

RUM's layered architecture properly addresses the requirements imposed by the SA problem. The MS/MT experiment described in this paper, has been used to illustrate RUM's capabilities in an IF/SA application. It is a complete experiment, but certainly not a complex one. A more strenuous and realistic validation of RUM is in progress: currently RUM is successfully being used as the reasoning system of the Situation Assessment module in DARPA's Pilot's Associate Program [Sweet 1986]. In this application, the six tasks (described in section 4) that comprise the retrospective SA problem are addressed by RUM in Scenarios involving up to twenty platforms. This application is also used to derive some of the real-time requirements that will represent the focus of future development work in RUM.

6. References

[Bonissone 1985] P.P. Bonissone & K.S. Decker, "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. Also in L.Kanal & J.Lemmer (Eds.) *Uncertainty in Artificial Intelligence*, pp. 217-247, North-Holland, 1986.

[Bonissone 1986] P.P. Bonissone, "Summarizing and Propagating Uncertain Information with Triangular Norms", in Lee S. Baumann, (Ed.) *Proceedings of the Expert Systems Workshop*, pp 62-71, Defense Advanced Research Projects Agency, Science Applications International Corporation, Pacific Grove, California, April 1986. To appear in the *International Journal of Approximate Reasoning*, North-Holland Publishing Company, 1987.

[Bonissone 1987a] P.P. Bonissone, S. Gans, K.S. Decker, "RUM: A Layered Approach to Reasoning with Uncertainty", manuscript submitted for presentation at the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87),

[Bonissone 1987b] P.P. Bonissone, J.K. Aragones, K.S. Decker, "LOTTA: An Object Based Simulator for Reasoning in Antagonistic Situations", Working Paper, General Electric Corporate Research and Development, Schenectady, New York.

[Clarkson 1981] A. Clarkson, *Toward Effective Strategic Analysis: New Applications of Information Technology*, Westview Press, Boulder, Colorado, 1981.

[Dillard 1978] R.A. Dillard, "New Methodologies for Automated Data Fusion Processing" NOSC Technical Report 364, September 1978.

[Dillard 1983] R.A. Dillard, "Computing Confidences in Tactical Rule-Based Systems by Using Demster-Shafer Theory", NOSC Technical Document 649, September 1983.

[Ferranti 1981] J. P. Ferranti, "Evaluation of the Artificial Intelligence Program STAMMER2 in the Tactical Situation Assessment Problem", M.S. Thesis, Naval Postgraduate School, Monterrey, California, March 1981.

[Bechtel 1979] R.J. Bechtel, P.H. Morris, "STAMMER: System for Tactical Assessment of Multisource Messages, Even Radar", NOSC Technical Document 252, May 1979.

[Levitt 1984] T.S. Levitt, G.J. Courand, M.R. Fehling, R.M. Fung, C.F. Kaun, R.M. Tong, "Intelligence Data Analysis", TR-1056-6, Volume 1, Advanced Decision Systems, Mountain View, CA 94040.

[McCall 1979] D.C. McCall, P.H. Morris, D.F. Kilber, R.J. Bechtel, "STAMMER2 Production System for Tactical Situation Assessment", NOSC Technical Document 298, October 1979.

[Sweet 1986] L.M. Sweet, P.P. Bonissone, A.L. Brown, S. Gans, "Reasoning with Incomplete and Uncertain Information for Improved Situation Assessment in Pilot's Associate", Proceedings of the *12th DARPA Strategic Systems Symposium*, Monterey, California, October 28-30, 1986.

A Role for Assumption-Based and Nonmonotonic Justifications in Automating Strategic Threat Analysis

Dale E. Gaucas and **Allen L. Brown, Jr.**
GE Corporate Research and Development Center
P.O. Box 8
Schenectady, New York 12308

Abstract

This report describes an experiment in using an assumption-based and nonmonotonic reasoning capability in support of strategic analysis as envisioned by Albert Clarkson. In particular, a new representational form for his notion of a threat model is developed and used to recode his North Korean threat scenario. In addition, a methodology for realizing a selected portion of his functional description of threat recognition is proposed. This methodology is demonstrated in a prototype problem solver and a hypothetical threat assessment involving Clarkson's encoded scenario. Finally, the roles of causal and temporal reasoning, uncertainty, and the control of reasoning in automating threat analysis are discussed.

I. Clarkson's View of Strategic Analysis

In his book Albert Clarkson [Clarkson, 1981] presents a concept for computer-based strategic analysis, intended to be a start at addressing some of the problems involved in this activity, *e.g.*, human limitations in reasoning about quantities of data. He provides a specification of strategic analysis in terms of three functional stages: monitoring, threat recognition, and projection. In this process model, significant indicators are recognized by the monitoring stage and associated with models of situations by the threat recognition stage. The models are then used for predictions and forecasts by the projection stage. Clarkson's own concept for realizing strategic analysis entails the design of various forms or schemata for both representing knowledge and hypothesizing threat situations. He claims that they should be flexible in order to support the analyst's creative process. Such forms, together with accompanying analysis routines, are viewed by Clarkson as supporting a system which

> *preserves, tailors and allows manipulation of a large operational situation of prior analysis as a framework for assigning meaning to new information.*

Focusing on the threat recognition stage of analysis, Clarkson's functional view of this activity requires that the analyst propose hypothetical situations to be recognized in advance of their full impact. The aspects of this stage which are relevant to our representation of threat assessment presented in section III. include:

- correlating indicators with preestablished threat models
- correlating other previously detected critical events with preestablished threat models
- identifying key events and activities whose occurrences have not been detected
- identifying further information needs
- comparing all preestablished threat models with which the data has positively correlated.

The preestablished threat models attempt to capture "potential courses of action by various countries, entities and decision makers", and act as filters through which the analyst reviews input data. Clarkson suggests a realization of these models through the implementation of the following high level representational forms, some of which are also useful in monitoring. These forms provide a context for the indicators.

- The PAMNACs form (Projected Alternative Major National Courses of Action) is used for representing a country's definitive national policies and courses of action with respect to key strategic aspects. It must be flexible to "accommodate new analytic perspectives as they arise."
- The DENs (Decision/Event Networks) are extensions of the PAMNACs intended to model how a particular course of action might be implemented, and must be easily changeable.
- The CEFs (Critical Event Filters) associate with a node from the PAMNACs or DENS additional activities which signify that node. These activities can be thought of as events which can be verified by incoming data.
- The ANEMs (Anomalous Event Matrices) assist the analyst in changing his existing models when incoming data does not support the hypothesized situations well.

The threat assessment stage then corresponds to these forms as follows: the indicators are matched against the various models; the correlations are identified; the models

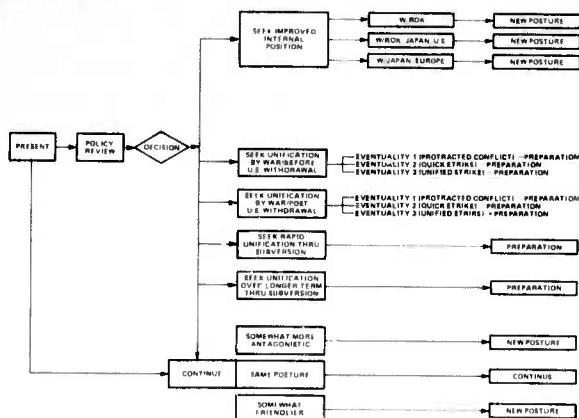


Figure 1: Example PAMNAC.

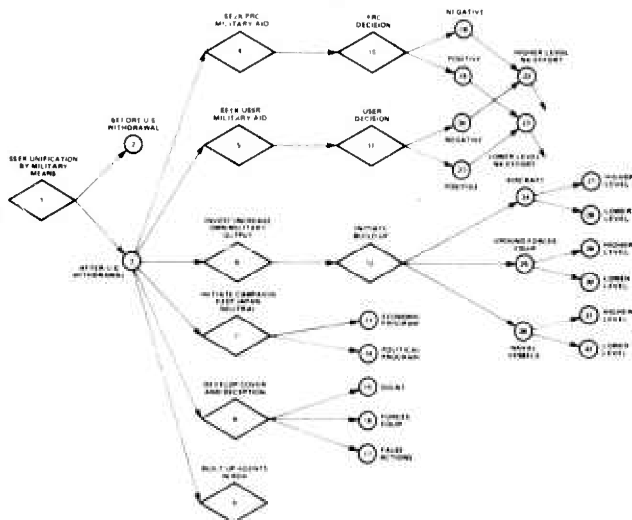


Figure 2: Example DEN.

as a set are compared in terms of their relative levels of activity; novel threat analysis is conducted.

II. Representing Threat Situations

A. A Threat Scenario

Clarkson provides general examples of his representational forms in terms of a hypothetical national security scenario which addresses "the problem of warning of hostile activity directed against U.S. interests and assets by North Korea." During the monitoring stage, indications of a turbulent situation developing in the Republic of Korean (ROK) such as increased student demonstrations might suggest the opportunity for hostile activity by North Korea against ROK. Relevant PAMNACs and DENs (see figures 1, 2, and 3) include nodes representing North Korean national policies and courses of action, for example, with respect to the reunification of the Korean peninsula and with respect to North Korea's international position. This scenario can be reformulated in terms of a new approach to supporting threat assessment, presented in the following sections.

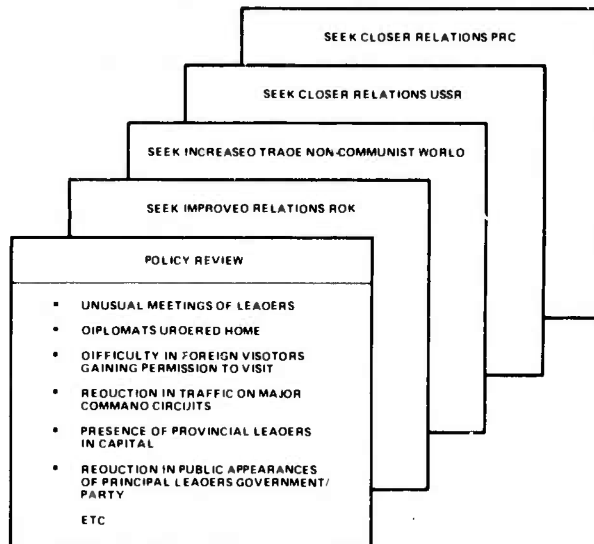


Figure 3: Example CEF.

This approach does not address the novel threat assessment supported by the ANEM forms.

B. A Formal Model for Situation Assessment

In this section, a model for situation assessment is described in an attempt to formalize what the situation assessment problem might be for the strategic threat analysis domain. This abstraction is inspired by the diagnosis problem [de Kleer and Williams, 1986]. In the most extreme case diagnosis is carried forth in an environment where the models of the mechanism to be diagnosed are accurate and complete, and the operational data are reliable and readily available. In diagnosis, the situation assessment problem is to enumerate the models of the mechanism compatible with the observed data. This is done in the context of a structural model (which remains invariant and is part of the mechanism's specification) and a functional model (which may include local deviations from the device's *specified* functional model).

The structural/functional model can be adopted as a general paradigm for situation assessment, with the view that "theory formation" is not a situation assessment problem. The structural model defines the essential causality of the artifact to be analyzed, and observables are interpreted within the confines of that causal model. Some considerations that distinguish one situation assessment problem from another are

1. the extent to which time has a role in the interpretation of structure and function,
2. the accuracy of observations,
3. the cost of observations,
4. the availability of observations,
5. the extent to which observations are direct evidence of function.

In this light, a structural and functional description for one view of strategic situation assessment can be formulated. A structural description corresponds to PAMNACs and DENs and can be viewed as a directed acyclic graph each of whose nodes correspond to *assuming* a causal linkage among particular events. Functional descriptions, on the other hand, correspond to the actual logical (entailment) relations among events. These events include strategic decisions such as ***decide whether or not to ask the Soviet Union for military assistance.*** (Though most of these decisions are under the control of North Korea, some–such as the USSR's deciding whether or not to respond favorably to a request for assistance–are not.)

We implicitly identify each of the assumption nodes with the event that is the *effect* of the corresponding causal linkage. Nodes occurring at the heads of the connecting arcs in the graph are interpreted as occurring after those at the tails in the sense that the corresponding effects of the former precede the corresponding effects of the latter. The situation assessment problem (informally) is to determine which of the causal linkages can be presumed to be in effect. These causal linkages represent the decision making structures that are thought to prevail in North Korea. In support of the reasoning mechanisms discussed in section III., the structural model can be represented by a distributive lattice of entities called "situations." The situations are the 'nodes' mentioned above, with the arcs of the graph corresponding to the ordering of the lattice, but with the opposite sense. Hence the sense of the order in the lattice is the opposite of that usually associated with time. A situation is said to sanction a causal relationship, in that a reasoning agent takes the corresponding nonmonotonic logical relationship as axiomatic only when the sanctioning situation is part of its current assessment. A situation assessment will then be one of a distinguished collection of the possible lattice expressions.

The axioms sanctioned by situations are called causal rules, and are one aspect of the functional description. The second aspect of the functional description corresponds to CEFs. These are represented as logical entailments called indicative rules. Indicative rules provide evidence that certain events have occurred that cannot be directly detected. Indicative rules have monotonic antecedents (indicators) and nonmonotonic antecedents (contraindicators). A special case called confirmation rules have only monotonic antecedents. The set of monotonic antecedents of an indicative rule is called a critical set. Notice that the same formula may appear as a monotonic antecedent of more than one indicative rule.

A situation assessment begins by assuming the situation which corresponds to the belief that none of the causal rules is currently sanctioned. That is, NK's decision making processes are inactive. As data comes in, rules are triggered that cause the current assessment to become a (non-trivial) conjunction of situations (corresponding causal rules that are consistent with and supportive of the current observations). Because of the contraindicators, it is possible for the belief status of indicated events to oscillate, that is, vary between believed and not believed ("IN" and "OUT" [Doyle, 1979]). In the special case that belief is supported by a confirmation rule the indicated event will be permanently adopted. To some extent the reasoning agent's problem is to choose between the definitive evidence given by confirmation rules and the suggestive evidence given by indicative rules having nonmonotonic antecedents.

A crucial element of the situation assessment process is the control of the inferences and observation requests made in order to arrive at the current assessment. The concept for a situation assessment problem solver outlined in section III. supports such control at various levels. Several observations can be made about this approach. For simplification, situation assessments are defined with respect to a fixed theory. Causal and indicative rules are propositional in nature and do not have explicit temporal or certainty aspects. Such oversimplifications result in inadequate expression of temporal relations and general logical relations.

C. A Formal Language for Situation Assessment Models

The situations and causal and indicative rules of situation assessment models can be represented in a formal system of logic called Epilog. Epilog is an epistemic, modal, nonmonotonic extension of propositional logic equipped with a minimal model semantics [Brown, 1987]. Epilog's proof theory is based on ordinary propositional deduction together with lattice theory-based reason maintenance [Benanav *et al.*, 1986, Brown *et al.*, 1987, Brown, 1987] and the rule of necessitation. Informally, the Epilog syntax contains the following entities:

- Propositions
  p, q, s, t
- Clauses
  p, $q \leftarrow p$, $q \leftarrow q_1 \wedge \cdots \wedge q_n$
- Beliefs
  $\Box p$, $\Box \neg q$, $\neg \Box s$, $\neg \Box \neg t$
- Justifications
  $\Box[\neg]p \leftarrow [\neg]\Box[\neg]q_1 \wedge \cdots \wedge [\neg]\Box[\neg]q_n$
- Situations
  $\top$, $\bot$, M, P, Q, S, $\overline{P}$, $P \sqcup Q$, $P \sqcap Q$
- Axioms
  situation $\rightarrow$ *clause*, *situation* $\rightarrow$ *justification*

Theories are sets of axioms whose situations when conjoined are not algebraically equal to $\bot$. The prototype problem solver whose operation is described in section III.B. currently uses the forward and backward chaining machinery of the Intellicorp's KEE^{TM} 3.0 rule system, a propositional prover, and reason maintenance to produce the inferences that would be produced by a "native" implementation of Epilog.

D. An Encoding of a Threat Model

In this section we present a portion of the structural and functional description of the North Korea scenario (with extensions). We base this on the model described in section II.B. There is for each situation, A_i, sanctioning a causal linkage, an event, a_i, that is the effect of that linkage. The linkage itself will be represented as a logical implication. Thus when the situation assessment reasoning agent has A_i as a conjunctive component of assessment, the agent sanctions the notion that (dis-)belief in the (non-)monotonic antecedents entails belief in the consequent or effect of the causal linkage. Using the abbreviations 'NK', 'ROK', 'USSR', and 'PRC' for North Korea, the Republic of Korea, the Soviet Union, and the Peoples' Republic of China respectively, the following propositions correspond to decision events:

- $s1$ - North Korea makes a policy change
- $q1$ - NK seeks unification of the peninsula by military means after U.S. military withdrawal via protracted, all out war
- $q4$ - NK seeks PRC military aid
- $q5$ - NK seeks USSR military aid
- $q6$ - NK invests in increase in own military output
- $q7$ - NK initiates campaign to keep Japan neutral
- $q8$ - NK develops cover and deception
- $q9$ - NK builds up agents in ROK
- $q18$ - PRC decision to help NK negative
- $q19$ - PRC decision to help NK positive
- $q20$ - USSR decision to help NK negative
- $q21$ - USSR decision to help NK positive
- $q22$ - higher level NK effort
- $q23$ - lower level NK effort
- $q12$ - NK initiates build-up
- $q27$ - NK initiates high increase in one or more of aircraft, ground forces equipment, naval vessels
- $q28$ - NK initiates low increase in one or more of aircraft, ground forces equipment, naval vessels
- $q13$ - NK initiates one or more of economic programs or political programs to keep Japan neutral
- $q15$ - NK develops cover and deception through one or more of sigint, forces/equipment, false actions
- $p1$ - NK seeks unification of the peninsula by military means before U.S. military withdrawal via quick strike

The following additional propositions (indicators) correspond to events and are needed for indicative rules:

- $t1$ - unusual meetings of NK leaders
- $t2$ - diplomats ordered home from NK capital
- $t3$ - difficulty in foreign visitors gaining permission to visit NK capital
- $t4$ - reduction in traffic on major command circuits in NK capital
- $t5$ - presence of provincial leaders in NK capital
- $t6$ - reduction in public appearances of principal NK leaders government/party
- $t7$ - NK diplomatic meetings with PRC
- $t8$ - NK diplomatic meetings with USSR
- $t9$ - intelligence reports USSR decision to support NK negative
- $t10$ - death of important NK political figure
- $t11$ - NK economic crisis
- $t12$ - NK internal political conflict
- $t13$ - NK military leaders present in PRC
- $t14$ - NK grants PRC economic favors
- $t15$ - pro PRC propaganda in NK government media
- $t16$ - PRC meetings with ROK
- $t17$ - increased NK diplomatic communication with PRC
- $t18$ - increased NK diplomatic communication with USSR
- $t19$ - intelligence reports high increase in aircraft in NK
- $t20$ - convergence of NK ground units in areas outside their garrisons
- $t21$ - intelligence reports low increase in aircraft in NK

Some of the partial order relations in the lattice of situations are indicated below.

$$
\begin{array}{rcl}
S1 & \sqsupseteq & P1 \sqcup Q1 \\
Q1 & \sqsupseteq & Q4 \sqcup Q5 \sqcup Q6 \sqcup Q7 \sqcup Q8 \sqcup Q9 \\
Q4 & \sqsupseteq & Q18 \sqcup Q19 \\
Q5 & \sqsupseteq & Q20 \sqcup Q21 \\
Q6 & \sqsupseteq & Q12 \\
Q7 & \sqsupseteq & Q13 \\
Q8 & \sqsupseteq & Q15 \\
Q9 & & \\
Q18 & \sqsupseteq & Q22 \\
Q19 & \sqsupseteq & Q23 \\
Q20 & \sqsupseteq & Q22 \\
Q21 & \sqsupseteq & Q23 \\
Q12 & \sqsupseteq & Q27 \sqcup Q28 \\
Q13 & & \\
Q15 & & \\
Q22 & \sqsupseteq & Q27 \\
Q23 & \sqsupseteq & Q28 \\
Q27 & & \\
Q28 & &
\end{array}
$$

Some of the causal rules of the structural model follow.

$$
\begin{array}{lll}
\overline{Pi} & \rightarrow & \Box\neg pi \\
\overline{Qi} & \rightarrow & \Box\neg qi \\
\overline{Si} & \rightarrow & \Box\neg si \\
P1 & \rightarrow & \Box p1 \leftarrow \Box s1 \wedge \neg\Box \text{ab-}p1 \\
Q1 & \rightarrow & \Box q1 \leftarrow \Box s1 \wedge \neg\Box \text{ab-}q1 \\
Q4 & \rightarrow & \Box q4 \leftarrow \Box q1 \wedge \neg\Box \text{ab-}q4 \\
Q5 & \rightarrow & \Box q5 \leftarrow \Box q1 \wedge \neg\Box \text{ab-}q5 \\
Q6 & \rightarrow & \Box q6 \leftarrow \Box q1 \wedge \neg\Box \text{ab-}q6 \\
Q7 & \rightarrow & \Box q7 \leftarrow \Box q1 \wedge \neg\Box \text{ab-}q7 \\
Q8 & \rightarrow & \Box q8 \leftarrow \Box q1 \wedge \neg\Box \text{ab-}q8 \\
Q9 & \rightarrow & \Box q9 \leftarrow \Box q1 \wedge \neg\Box \text{ab-}q9 \\
Q23 & \rightarrow & \Box q23 \leftarrow \Box q5 \wedge \Box q21 \wedge \neg\Box \text{ab-}q23 \\
Q18 & \rightarrow & \Box q18 \leftarrow \Box q4 \wedge \neg\Box \text{ab-}q18 \\
Q19 & \rightarrow & \Box q19 \leftarrow \Box q4 \wedge \neg\Box \text{ab-}q19 \\
 & & \Box \text{ab-}q19 \leftarrow \Box\neg q19 \\
Q20 & \rightarrow & \Box q20 \leftarrow \Box q5 \wedge \neg\Box \text{ab-}q20 \\
Q21 & \rightarrow & \Box q21 \leftarrow \Box q5 \wedge \neg\Box \text{ab-}q21 \\
 & & \Box \text{ab-}q21 \leftarrow \Box\neg q21 \\
Q12 & \rightarrow & \Box q12 \leftarrow \Box q6 \wedge \neg\Box \text{ab-}q12 \\
Q27 & \rightarrow & \Box q27 \leftarrow \Box q12 \wedge \Box q22 \wedge \neg\Box \text{ab-}q27 \\
 & & \Box \text{ab-}q27 \leftarrow \Box q28 \\
Q28 & \rightarrow & \Box q28 \leftarrow \Box q12 \wedge \Box q23 \wedge \neg\Box \text{ab-}q28 \\
 & & \Box \text{ab-}q28 \leftarrow \Box\neg q28 \\
 & & \Box \text{ab-}q28 \leftarrow \Box q27
\end{array}
$$

Some relevant indicative rules include the following.

$$
\begin{array}{rcl}
t17 & \leftarrow & t7 \\
\Box\neg q19 & \leftarrow & \Box t16 \\
t18 & \leftarrow & t8 \\
\Box\neg q21 & \leftarrow & \Box t9 \\
\Box\neg q28 & \leftarrow & \Box t19 \\
\Box s1 & \leftarrow & \Box t1 \wedge \Box t2 \wedge \Box t3 \wedge \Box t4 \wedge \Box t5 \wedge \Box t6 \\
 & & \wedge \neg\Box t10 \wedge \neg\Box t11 \wedge \neg\Box t12 \\
\Box q4 & \leftarrow & \Box t17 \wedge \neg\Box\neg q4 \\
\Box q5 & \leftarrow & \Box t18 \wedge \neg\Box\neg q5 \\
\Box q19 & \leftarrow & \Box q4 \wedge \Box t13 \wedge \neg\Box\neg q19 \\
\Box q19 & \leftarrow & \Box q4 \wedge \Box t14 \wedge \neg\Box\neg q19 \\
\Box q19 & \leftarrow & \Box q4 \wedge \Box t15 \wedge \neg\Box\neg q19 \\
\Box q21 & \leftarrow & \Box q5 \wedge \neg\Box\neg q21
\end{array}
$$

III. Reasoning About Threat Situations

A. A Situation Assessment

The automation of threat assessment alluded to in section II.B. can be demonstrated through a brief reasoning sequence based on Clarkson's North Korean threat scenario. A high level overview of the reasoning sequence follows: Initially, with no information to indicate otherwise, there are no imminent threats from NK. Eventually, the observation of some events including an unusual meeting of leaders in the NK capital and diplomats being ordered home, indicates that NK is making a policy change. Such a policy change may lead to several courses of action on the part of NK. The one preferentially assumed (based on a plausibility ordering of situations) to be in progress is an attempt at unification by military means after US withdrawal, via a protracted, all out war. It is reasonable to believe that NK is seeking USSR military aid if it can be determined that NK is having increased diplomatic communication with USSR (which is indeed observed), and there is no evidence to indicate that the aid request is not being made. It is also reasonable to believe that the USSR will be supportive unless there is evidence to indicate otherwise. From assuming such assistance, it can then be assumed that NK will only need to initiate a low increase in military equipment build-up. The receipt of an intelligence report that the USSR decision was not in support of NK contradicts the belief that they would be supportive. This leads to the revised assumptions that after requesting USSR aid and being turned down, NK will have a higher level of effort and will greatly increase its military forces.

In this example, the information about diplomatic communication with USSR suggests a proper subset of the possible situations presented in the structural model. Deducing that this information discriminates among several candidate situations involves reasoning control, *i.e.*, utilizing selected rules sanctioned by selected situations. For example, from the indication of a policy change, it can be deduced that there are several situations sanctioning causal entailments whose antecedents include a policy change. From any particular one, it may be possible to deduce that certain decision events have already been initiated. From such predictions, it may also be possible to deduce observations to be made that can either substantiate the initiation or eliminate it. In the specific example above, assuming that NK is seeking unification by military means, after US withdrawal, via all out war, with the assistance from the USSR but not the PRC, suggests diplomatic interaction between NK and USSR. Subsequent reasoning leads to the belief in low build-up of NK forces which, upon receipt of new information, is revised to belief in high build-up of NK forces. If it was eventually observed that a low build-up of forces was in progress, the situation assessment sanctions the belief that the PRC was providing assistance. The deductions referred to above are

performed by forward and backward inference using the rules of the structural/functional model, the partial ordering on the situations, and current control strategy. Such a reasoning capability is supported by the architecture suggested in section III.B.

B. A Problem Solving Architecture for Situation Assessment

The situation assessment paradigm envisioned is supported by a problem solving system with the following components: a reason maintenance system (ANRMS), a deductive system (DS) consisting of a propositional theorem prover and a defeasible inference mechanism directly supported by the ANRMS, and a control system (CS). The interfacing among these subsystems is more complicated than simple layering, but to a first approximation one can imagine them to be layered in an ascending hierarchy in the order cited.

As discussed in section II.B., the structural/functional model is encoded as a collection of causal rules capturing the causality between various decision making events, and a collection of indicative rules modelling possible antecedents and logical consequences of such decisions. The situations are represented as ANRMS Boolean lattice elements, and the justifications and beliefs are represented as ANRMS constraints and nodes respectively. The justifications have the property that establishing their monotonic antecedents, while having no evidence for their nonmonotonic antecedents (defeasibly) assures their consequents. The clauses and propositions and represented by logical formulae. Establishing the antecedents of a material implication can be used to confirm the consequent. A situation assessment is an algebraic expression from the Boolean lattice of situations.

An operational situation assessment application (SAA) is a collection of reasoning agents (or processes), each of which has under its control a DS with a corresponding assessment (or state) characterized by a lattice expression. A reasoning agent's DS reasons from certain pieces of knowledge represented by nodes in the ANRMS. In particular, it reasons from those nodes whose reason maintenance labels have disjuncts that are consistent with the reasoning agent's characterizing lattice expression L, *i.e.*, disjuncts which when conjoined with L do not sanction logical falsehood (contradiction). A given reasoning agent continues to reason until a contradiction is produced, that is, when the distinguished reason maintenance node, FALSE, is entailed by that agent's situation assessment. A reasoning agent is termed *active* if its situation assessment does not entail the FALSE node. At that point a new reasoning agent is generated, whose situation assessment does not (currently) entail an inconsistency. At any given time the situation assessment is the (lattice-theoretic) meet of the situation assessment expressions of each of the reasoning agents whose assessments do not currently entail a contradiction. In general, it will be the aspiration of a situation assessment application to have exactly one active reasoning agent whose situation assessment will have exactly one disjunct. Arranging for this to be the case is the central (and essential) control issue of any situation assessment application.

Such a problem solving paradigm differs from de Kleer's General Diagnostic Engine in the following ways:

1. there are two levels of inference: inferences about the problem solving domain (domain deductions) and inferences about the control of the former (control deductions); since both inference structures are logic based, one is said to have deductive control of inference
2. the deductive system for the problem domain has various "points of intercession" at which deductive control can be exercised
3. the inferential stance of both layers is firmly rooted in logic
4. there is an explicit abstraction of the notions of a reasoning agent and a reasoning agent's "mental state"; this abstraction is presented at the interface between the theorem prover responsible for domain deductions and the theorem prover responsible for control deductions.

C. An Instance of a Problem Solver

A realization of the situation assessment problem solving paradigm discussed in III.B. devolves to addressing a collection of control issues. These issues appear to fall into a natural hierarchy and as such impose a natural hierarchical structure on a situation assessment problem solver. To each control issue corresponds a control policy for guiding control, a policy language for describing the policy, and a control methodology for realizing the policy. An example of a policy might be the prescript to assume that either part A is working or that it is not working. A policy language might be the collection of conjunctive lattice expressions containing the assumption A or its complement. An example of a control methodology might be a procedure that chose only deductions entailed by the previously mentioned control policy. There is a list of roughly a half dozen control issues, arranged in a two-tiered hierarchy. The two tiers correspond roughly to deciding when to create a reasoning agent and when to schedule it to run vs. deciding what deductive goal a reasoning agent should attempt to achieve and how to achieve it:

1. reason only from nodes sanctioned by the reasoning agent's current assessment;
2. use only axioms sanctioned by the reasoning agent's current assessment;
3. in general causal rules are forward chained;
4. the disambiguation of situation assessments is to take place in a depth-first fashion;
5. in general, indicative rules are backward chained for the purpose of establishing of the most likely discrim-

inating measurement to disambiguate a situation assessment.

D. A Problem Solver's Reasoning Sequence

This section presents an example reasoning sequence in detail in terms of the outlined problem solver of section III.C. The reasoning sequence of section III.A. is described as a sequence of deductive states, some of which summarize several deductive steps. At each state, the corresponding situation assessment (which necessarily does not entail an inconsistency) is based on the current set of beliefs. The distinguished assumption M represents the assumption that all measurements are correct, and is required to be entailed by all situation assessments.

The assessment process is initiated by the following sequence.

- The reasoning agent's initial assessment is $M \sqcap \overline{S}1$
- Unsolicited observations lead to
  $M \rightarrow \Box t1$
  $M \rightarrow \Box t2$
  $M \rightarrow \Box t3$
  $M \rightarrow \Box t4$
  $M \rightarrow \Box t5$
  $M \rightarrow \Box t6$
- Forward chaining on

$$\Box s1 \leftarrow \Box t1 \wedge \Box t2 \wedge \Box t3 \wedge \Box t4 \wedge \Box t5 \wedge \Box t6 \wedge \neg\Box t10 \wedge \neg\Box t11 \wedge \neg\Box t12$$

  leads to $M \rightarrow \Box s1$
- The reasoning agent's assessment supports the contradiction $M \sqcap \overline{S}1 \rightarrow \Box s1 \wedge \Box\neg s1$

  Initial belief revision and refinement follows.
- The most preferable initial revision is $M \sqcap S1$
- The subsequent refinements are $M \sqcap Q1$, $M \sqcap Q5$, $M \sqcap Q21$, $M \sqcap Q23$, and $M \sqcap Q28$
- The refinement $M \sqcap Q5$ is supported by backchaining from $\Box q5 \leftarrow \Box t18 \wedge \neg\Box\neg q5$ which solicits the observation of $t18$ yielding $M \rightarrow \Box t18$ and $M \sqcap Q5 \rightarrow \Box q5$
- The refinement $M \sqcap Q21$ is supported by backchaining from $\Box q21 \leftarrow \Box q5 \wedge \neg\Box\neg q21$ which yields $M \sqcap Q21 \rightarrow \Box q21$
- The reasoning agent's new best assessment is $M \sqcap Q28$

  New evidence leads to further reasoning.
- Unsolicited observation leads to $M \rightarrow \Box t9$
- Forward chaining from $\Box\neg q21 \leftarrow \Box t9$ leads to $M \rightarrow \Box\neg q21$
- Forward chaining on the causal rules leads to $M \sqcap Q28 \rightarrow \Box q21$
- But this means that the reasoning agent's current assessment supports the contradiction $M \sqcap Q28 \rightarrow \Box q21 \wedge \Box\neg q21$
- Note also that $M \rightarrow \Box\neg q21$ also negates the earlier nonmonotonic support of $\Box q21$

Subsequent belief revision and refinement updates the current assessment.

- $Q5$ being the least (in the lattice) situation above $Q28$ in which the contradiction goes away, the reasoning agent's initial reassessment is $M \sqcap Q5$
- Since the only choices for refinement are $M \sqcap Q20$, $M \sqcap Q22$, and $M \sqcap Q27$, the reasoning agent's final assessment is $M \sqcap Q27$

IV. Conclusions

We have presented a formal model of situation assessment and a logical language and interpreter for encoding threat situations. We have also proposed a problem solving architecture comprised of both the interpreter and a system for controlling its application.

A lattice theoretic reason maintenance system serves as the focus of control for the problem solver in forward and backward chaining. Lattice situations encode both assessments and a weak (but sufficient for the present purpose) model of time. Nonmonotonic justification allows the weak support of certain conclusions which, with new evidence, may be withdrawn.

The structural/functional model is encoded in the logical language. Rules which are sanctioned by particular situations encode the causal content of the North Korea strategic decision model while rules sanctioned by all situations encode the functional content. Candidate situation generation is done in a way similar to the practice in diagnosis. In this view of situation assessment the computational goal is not merely to minimize the number of observations needed to formulate an assessment, but also to make the most reliable observations. Kautz and Allen [1986] present a promising theoretical framework for carrying out the recognition task for situation assessment problems. There are interesting parallels and contrasts between their work and our own efforts.

Although it appears that Clarkson's model of situation assessment (as published) is adequately representable in the paradigm presented here, problems would arise should strategic situation assessment require a richer representation of temporal phenomena. Indeed, this work demonstrates essential roles for the representation of causality, time and uncertainty, and for the general issue of the control of reasoning.

In our model time/causality is in effect represented by material implications having nonmonotonic antecedents and being sanctioned by situations that serve in effect as temporal indices. In the indications and warning system of Douglas Lenat etal. [Lenat et al., 1983] time is represented explicitly through the use of a blackboard and an interval representation. The lessons provided by Lenat's work along with a theory of time and causality related

to the one developed by Shoham [1986] offer a promising framework for an improved representation.

There is an important role for uncertainty as well. For example, the number of verifiable entries in the CEF associated with an event in the PAMNACS affects the degree belief in that event. In addition, incoming data has a particular degree of meaning with respect to a particular assessment. Our model indirectly addresses uncertainty through the notion of a critical set, and assumes a fixed partial order for determining the "best discriminating observation." Lenat's system provides a way for determining those facts whose certainty is crucial to important conclusions through the use of rules which have 'strong' and 'weak' conditions. This is accomplished by running the rules in two separate worlds, one in which only strong conditions are considered and one in which only weak conditions are considered. Major discrepancies in important predictions can be traced to facts whose validation must be given further attention. It appears that the architecture presented here could provide the basis for supporting an arbitrary number of such worlds.

The use of assumption and nonmonotonic based reasoning offers a point of departure for a situation assessment system to address various issues in the control of inference, both at the domain dependent (strategic) and domain independent (tactical) levels.

Finally, Clarkson points out that strategic analysis is not really a linear process in that the assumptions made in models could be affected by the analyses of the projection stage. He also distinguishes between changes made *to* the models and changes made *in* the models, the former being the creation of a new context. As mentioned earlier, this work does not address the problem of theory formation, and does not have the equivalent of ANEMs in its view of a structural model. It appears, however, that assumption based reasoning could play an important role in projection analyses. For example, knowing what events are shared by several possible projections would enable measures to be taken which prepare against more than one possible outcome

References

[Benanav *et al.*, 1986] Dan Benanav, Allen L. Brown, Jr., and Dale E. Gaucas. Reason maintenance from a lattice-theoretic point of view. In Lee S. Baumann, editor, *Proceedings of the Expert Systems Workshop*, pages 83–87, Defense Advanced Research Projects Agency, Science Applications International Corporation, Pacific Grove, California, April 1986.

[Brown, 1987] Allen L. Brown, Jr. Logics of justified belief. These proceedings.

[Brown *et al.*, 1987] Allen L. Brown, Jr., Dale E. Gaucas, and Dan Benanav. An algebraic foundation for truth maintenance. These proceedings.

[Clarkson, 1981] Albert Clarkson. *Toward Effective Strategic Analysis.* Westview Press, Boulder, Colorado, 1981.

[de Kleer and Williams, 1986] Johan de Kleer and Brian C. Williams. Reasoning about multiple faults. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, Philadelphia, 1986.

[Doyle, 1979] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Kautz and Allen, 1986] Henry A. Kautz and James F. Allen. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986.

[Lenat *et al.*, 1983] Douglas B. Lenat, Albert Clarkson, and Garo Kiremidjian. An expert system for indications and warning analysis. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 1983.

[Shoham, 1986] Yoav Shoham. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence.* PhD thesis, Yale University, New Haven, Connecticut, 1986.

A Mathematical Theory for Diagnosis Based on the MONAD Concept*

G.B. Porter, III
GE Corporate Research and Development Center
K1-5C38A, P.O. Box 8, Schenectady, New York 12301
Phone: (518) 387–5141, Arpanet: GBPORTER@GE-CRD

April 28, 1987

Abstract

Recent work in automatic diagnosis has employed formal logic as a representation and the notion of consistency as the test for causal explanation. This paper uses the notion of variational analysis as a basis for a formal approach to diagnosis which develops a set of differential equations to represent the behavior of the system under test. Several examples in digital logic are shown in which specific failure modes are substituted into these equations to give an operational representation of the system. The resulting equations are evaluated for specific measurements to give a set of constraint equations which are solved to determine a diagnosis (which may be multiple fault). Aside from being an approach to diagnosis, the importance of this work is that it embodies a technique which extends uniformly to other reasoning tasks.

1 Introduction

Some of the recent work in the area of automated diagnosis has been moving in the direction of using formal logic to represent causal knowledge about real world systems. In [DW85] and [Rei85], for example, digital circuits are diagnosed using variations of this representation scheme. In this paper we describe an alternate representation and diagnosis method which, we believe, more easily extends to complex devices such as analog hardware and physical systems. The method is described more fully in [Por87] in which it is

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

referred to as the **relation method**. It is used as the primary reasoning technique in the **MONAD** system in which it controls most of the general search processes in addition to being central to performing diagnosis and analogy tasks which are the target applications of that system.

In this paper we are attempting to show how the relation method can be used for the diagnosis of static systems; we are not trying to reinvent diagnosis. The important point is that the technique described here is uniform with those applied to other tasks that are addressed by the **MONAD** system such as analyzing static systems, proposing modifications in the analogical reasoning method, and even revising the knowledge state as part of the reasoning control process. To understand some of the details of the concept, we apply the relation method and its associated representation to the diagnosis of digital circuits as discussed in [Rei85]. An abstract diagnosis task is also discussed to provide an illustration of the extensibility of the method and representation. We will only treat static circuitry in this paper since time dependent reasoning and reasoning about flow systems are treated using the reasoning control mechanism. Some systems with loops can be treated as an extension to the static method described here although they are generally dealt with using the flow analysis method which handles dynamic systems.

The method described here corresponds to a process in the **MONAD** system and therefore has the goal of formulating an abstract problem specification in terms of a set of equations. The method we describe here is an example of such a process for static digital logic. There is a corresponding process for dealing with analog circuits based on the Lagrange method for describing systems. It develops a set of differential equations from an abstract circuit description while the method here develops a set of *digital* differential equations from a digital circuit specification. Both of these are handled uniformly under the relation method of the **MONAD** system.

2 The Diagnosis Process

Many of the diagnosis systems developed over the past few years use the method of inconsistency to hypothesize the failed component[1] of the target system, for example, see [Bon82]. Without considering specific techniques, this method determines that the set of behavioral measurements of the target system are inconsistent with the expected measurement values. From this observation, one may deduce that the system is not functioning properly (by the definition embedded in the inconsistency test). From these inconsistencies an explanation of the malfunction is hypothesized as the failure of one component in the target system. This last step is somewhat weak. This is because it is not always clear that we can know all of the implications of a component failure.

[1]For the moment we shall restrict our comments to single fault diagnosis.

2.1 Reiter's Approaches

In this section we will briefly mention the approach of [Rei85] to give some flavor to our discussion. Reiter defines diagnosis in terms of the normal or abnormal operation of each of its components. Each failure condition of the component set can be modeled as a consistency between the behavioral specification of the system under the hypothesized failure condition and the observed measurements. This is *only* correct if the universe of failure modes and their implications is completely known. We will review this notion in Reiter's terms.

Reiter defines a system as the pair $(SD, COMPONENTS)$ where SD is a set of first order sentences representing the system description or behavior, and $COMPONENTS$ is a finite set of constants representing the components of the system. Observations of the state of the system are represented as a set of first order sentences OBS. Component failure is represented in terms of individual components. $AB(c)$ is a "distinguished" predicate whose intended meaning is that component c is "abnormal"; the predicate $\neg AB(c)$ means that c is functioning "normally". If all the components of the system are functioning normally, then $SD \cup OBS \cup \{\neg AB(c) | c \in COMPONENTS\}$ is consistent.

Then Δ is a diagnosis for $(SD, COMPONENTS, OBS)$ if, for each $c_i \in \Delta$,

$$SD \cup OBS \cup \{\neg AB(c) | c \in COMPONENTS - \Delta\} \models AB(c_i). \tag{1}$$

This notion is simplified by showing that $\Delta \subseteq COMPONENTS$ is a diagnosis for $(SD, COMPONENTS, OBS)$ iff Δ is a minimal set such that:

$$SD \cup OBS \cup \{\neg AB(c) | c \in COMPONENTS - \Delta\} \tag{2}$$

is consistent.

2.2 The Source of the Problem

Reiter has neatly separated the diagnosis task into two processes: enumerating the possible failure conditions and testing consistency of the observations with the system description. Although both of these processes are *logically* sound, it is not clear that they are always practical to implement in a direct way. In real world diagnosis problems, the nature of the set of failures modes and/or implications of the failure modes of a set of components is *not* typically *completely* known. An experienced expert human diagnostician will, in general, hedge on the certainty of a particular diagnosis based only on a given set of measurements[2]. Rather, they will usually recite processes which will lead toward the elimination of certain components from suspicion.

The technical interpretation of this difficulty is that behavior of the system may be too difficult to describe in one unified set of formulae SD. The alternative is to use a context

[2]They have countless tails of the "tough ones" which were only tracked down after several unfruitful cycles of diagnosis and component replacement.

sensitive description so that each set of hypothesized component failures uniquely specifies a system behavior model which must then be checked for consistency with the observations. The method described in this paper tries to follow the path of least resistance by using the type of information which experts tend to give: rules for eliminating search paths. It reasons about possible component failure assignments in order to avoid the process of testing their consistency. Further, it is based on a set of abstract behavior models which may be analyzed symbolically to reduce the cost of testing the consistency in some cases.

3 Diagnosis Based on Variational Analysis

Our approach to diagnosis differs from most in that it tries to perform diagnosis on a symbolic level rather than by examining values. The problem is first formulated as a set of symbolic equations which may then be specialized in terms of the available measurements, and finally solved, again symbolically, to produce a bound on the set of possible failures.

Our approach is similar to Reiter's in that it finds a set of explanations for the malfunction of a system by finding a set of failure mode vectors which predict a system behavior which is consistent with the observed failure. Clearly, if the failure models of the components and the system are correct, then this method is sound. This approach can have the disadvantage we have cited above that, in principle, the system behavior must be tested for each allowable failure vector. We describe an interesting alternative to the enumeration process which is based on the relation method that is used as the primary search strategy in the **MONAD** system. It attempts to prune the search by manipulating a symbolic model of the system under test following which specific failure modes may be explored. By using approximations for the behavioral models, we may further simplify the search.

3.1 Relation Method

Prior to describing our approach to diagnosis, we will briefly review the relation method which is described more fully in [Por87].

In principle, the relation method is part of the differential reasoning engine which forms the heart of the **MONAD** system. It has the ability to uniformly find relations between various kinds of data objects within the knowledge base. The three basic representations of a relation between objects are equations, meaning a set, a conventional mathematical equation, or a logical expression; a process which is a procedure coded either directly or in the model language that produces an equation; and the type hierarchy which is essentially a semantic net that stores random relations between objects within the system.

The relation method knows how to selectively perform dependency analysis and differentiation on any combination of these representations. For example, figure 1 shows a circuit consisting of some multiplier and adder devices. Elements M_1, M_2, and M_3 are digital multiplier devices and A_1 and A_2 are digital adder devices. Figure 2 shows the

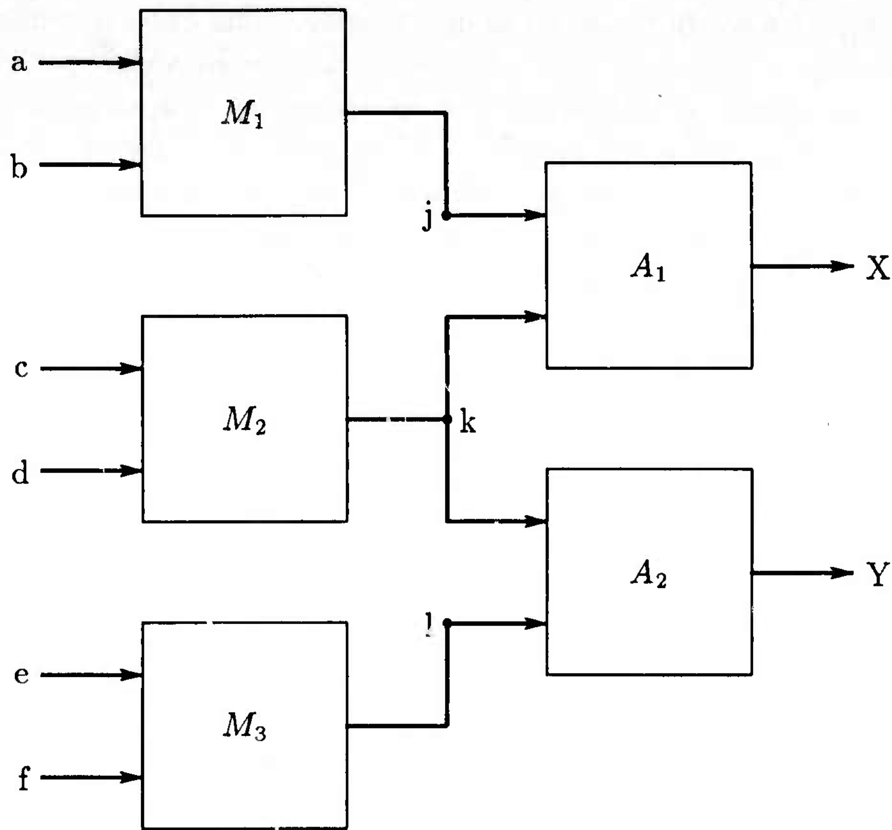


Figure 1: Multiplier Adder Block Diagram

result of differentiating the output variables X and Y with respect to two of the input variables a and c.

We will refer intuitively and somewhat formally to these concepts in developing our diagnosis method.

X depends on j and k
Y depends on k and l
j depends on a and b
k depends on c and d
l depends on e and f

$$\frac{dX}{da} = \frac{\partial X}{\partial j}\frac{dj}{da} \quad (3)$$

and
$$\frac{dX}{dc} = \frac{\partial X}{\partial k}\frac{dk}{dc} + \frac{\partial Y}{\partial k}\frac{dk}{dc} \quad (4)$$

Figure 2: Multiplier Adder Dependency Results

4 The Full Adder Example

Now we will apply the variational principles of the relation method to the problem of diagnosis. In diagnosing a system, we are actually attempting to find a subspace of the behavioral universe which contains the instance behaviors obtained from the measurements. There are two approaches to arrive at the component failures which correspond to this subspace and thereby could cause the observed behavior. One approach is to construct the subspace containing the observations by hypothesizing various component failures and testing to see that the observations are contained in the resulting subspace. An alternate is to find a mapping from the various possible subspaces to the component failures which could cause them. The first approach is the one we shall pursue since it has more potential for succeeding on complex problems. The second one corresponds to the approach used in those first generation of expert systems having shallow knowledge bases.

To gain an intuitive feeling for this approach, we will informally apply our approach to a simple example which was treated in [Rei85]. We will then provide a brief formal description to make these ideas concrete.

Figure 3 shows a circuit of a full adder. It is composed of *and* gates A_1 and A_2, *exclusive-or* gates X_1 and X_2, and an *or* gate O_1. The behavior of the circuit is to add together the three binary inputs a, b, and c resulting in the two binary outputs SUM and CAR. Each of these inputs and outputs meet the criterion $a, b, c, SUM, CAR \in \{0, 1\}$. SUM corresponds to the low order bit of the sum of the arguments and CAR corresponds to the carry bit – the 2's bit – of the sum of the input arguments. The table below specifies

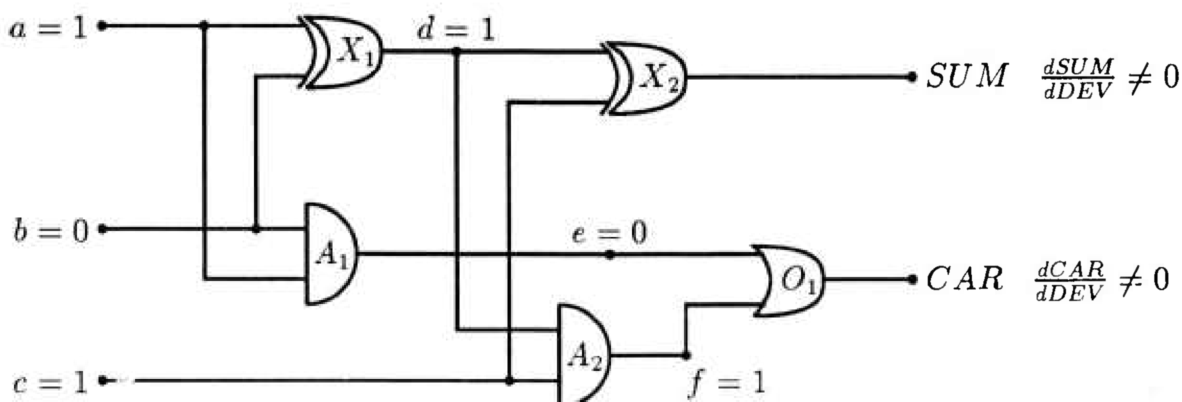


Figure 3: Full Adder Circuit Diagram

the outputs which result from any legal combination of inputs.

abc	SUM	CAR
000	0	0
001	1	0
010	1	0
011	0	1
100	1	0
101	0	1
110	0	1
111	1	1

The relations below describe the dependencies of this example:

$$\begin{aligned}
\{OUTPUTS\} &\triangleright \{SUM, CAR\} \\
\{SUM\} &\triangleright \{F_{X_2}\} \\
\{F_{X_2}\} &\triangleright \{c, d, \Delta_{X_2}\} \\
\{d\} &\triangleright \{F_{X_1}\} \\
\{F_{X_1}\} &\triangleright \{a, b, \Delta_{X_1}\} \\
\{CAR\} &\triangleright \{F_{O_1}\} \\
\{F_{O_1}\} &\triangleright \{e, f, \Delta_{O_1}\} \\
\{e\} &\triangleright \{F_{A_1}\} \\
\{F_{A_1}\} &\triangleright \{a, b, \Delta_{A_1}\} \\
\{f\} &\triangleright \{F_{A_2}\} \\
\{F_{A_2}\} &\triangleright \{c, d, \Delta_{A_2}\} \\
\{a, b, c\} &\triangleright \{INPUTS\} \\
\{\Delta_{X_1}, \Delta_{X_2}\} &\triangleright \{XORGATE\} \\
\{\Delta_{A_1}, \Delta_{A_2}\} &\triangleright \{ANDGATE\} \\
\{\Delta_{O_1}\} &\triangleright \{ORGATE\} \\
\{XORGATE, ANDGATE, ORGATE\} &\triangleright \{DEV\}
\end{aligned}$$

where $a \triangleright b$ means that a *depends* on b in some way. The functions F_{DEV} correspond to the behavioral model of the individual devices and are used to determine the details of the dependency.

For the moment, imagine that we have defined the notion of taking the derivative of the outputs with respect to the set of all devices. We would then be able to write:

$$\begin{aligned}
\frac{dSUM}{dDEV} &= \frac{\partial SUM}{\partial F_{X_2}} \frac{\partial F_{X_2}}{\partial \Delta_{X_2}} \frac{\partial \Delta_{X_2}}{\partial XORGATE} \frac{dXORGATE}{dDEV} \\
&+ \frac{\partial SUM}{\partial F_{X_2}} \frac{\partial F_{X_2}}{\partial d} \frac{\partial d}{\partial F_{X_1}} \frac{\partial F_{X_1}}{\partial \Delta_{X_1}} \frac{\partial \Delta_{X_1}}{\partial XORGATE} \frac{dXORGATE}{dDEV}
\end{aligned}$$

$$\Rightarrow \left\{\frac{dSUM}{dDEV}\right\} \quad \triangleright \quad \{X_1, X_2\}$$

$$\text{and} \quad \frac{dCAR}{dDEV} = \frac{\partial CAR}{\partial F_{O_1}} \frac{\partial F_{O_1}}{\partial \Delta_{O_1}} \frac{\partial \Delta_{O_1}}{\partial ORGATE} \frac{dORGATE}{dDEV}$$
$$+ \frac{\partial CAR}{\partial F_{O_1}} \frac{\partial F_{O_1}}{\partial e} \frac{\partial e}{\partial F_{A_1}} \frac{\partial F_{A_1}}{\partial \Delta_{A_1}} \frac{\partial \Delta_{A_1}}{\partial ANDGATE} \frac{dANDGATE}{dDEV}$$
$$+ \frac{\partial CAR}{\partial F_{O_1}} \frac{\partial F_{O_1}}{\partial f} \left[\frac{\partial f}{\partial F_{A_2}} \frac{\partial F_{A_2}}{\partial \Delta_{A_2}} \frac{\partial \Delta_{A_2}}{\partial ANDGATE} \frac{dANDGATE}{dDEV} + \frac{\partial f}{\partial F_{A_2}} \frac{\partial F_{A_2}}{\partial d} \frac{\partial d}{\partial F_{X_1}} \frac{\partial F_{X_1}}{\partial \Delta_{X_1}} \frac{\partial \Delta_{X_1}}{\partial XORGATE} \frac{dXORGATE}{dDEV} \right]$$

$$\Rightarrow \left\{\frac{dCAR}{dDEV}\right\} \quad \triangleright \quad \{O_1, A_1, A_2, X_1\}$$

where $XORGATE$, $ANDGATE$, and $ORGATE$ are the sets of device types and $DEV = \{XORGATE, ANDGATE, ORGATE\}$ is the set of all devices. The special symbol Δ is used to indicate device failure.

We may simplify the above equations by observing that, by definition, $\frac{\partial node}{\partial F_{DEV}} = \frac{\partial \Delta_{DEV}}{\partial CAT} = \frac{\partial CAT}{\partial DEV} = 1$, where the nodes are $node \in \{d, e, f\}$ and the categories $CAT \in DEV$. Substituting these values into the above equations we have:

$$\frac{dSUM}{dDEV} = \frac{\partial F_{X_2}}{\partial \Delta_{X_2}} + \frac{\partial F_{X_2}}{\partial d} \frac{\partial F_{X_1}}{\partial \Delta_{X_1}}$$

$$\text{and} \quad \frac{dCAR}{dDEV} = \frac{\partial F_{O_1}}{\partial \Delta_{O_1}} + \frac{\partial F_{O_1}}{\partial e} \frac{\partial F_{A_1}}{\partial \Delta_{A_1}} + \frac{\partial F_{O_1}}{\partial f} \left[\frac{\partial F_{A_2}}{\partial \Delta_{A_2}} + \frac{\partial F_{A_2}}{\partial d} \frac{\partial F_{X_1}}{\partial \Delta_{X_1}} \right]$$

These two equations are called the **behavioral equations** for the full adder circuit. The first may be interpreted to mean that the variation of the SUM output is a function of the failure mode of the X_2 exclusive or gate, or the failure mode of the X_1 exclusive or gate after it passes through the X_2 gate. That is, $\frac{\partial F_{X_2}}{\partial \Delta_{X_2}}$ or $\frac{\partial F_{X_1}}{\partial \Delta_{X_1}}$ are non-zero if X_2 or X_1 are operating in a failure mode respectively. The term $\frac{\partial F_{X_2}}{\partial d}$ is the behavior of the output of the X_2 with respect to changes in the d input. The second equation is read in a similar fashion but there are a few more possible causes for an output change.

Given a set of measurements on the system, we wish to evaluate the above behavioral equations in light of the model of the function of the circuit and its elements to determine a more definitive diagnosis. By the definition of normal operation of the circuit, $\frac{\partial F_{DEV}}{\partial \Delta_{DEV}} = 0$; that is, the variation of the output of a device is zero with respect to the failure mode variable of a device if and only if the device is not faulty. This notion is reflected in the above equations in that $\frac{dOUT}{dDEV}$ will be identically 0 when all the $\frac{\partial F_{DEV}}{\partial \Delta_{DEV}} = 0$. Correspondingly, a defective component has the characteristic that one or more $\frac{\partial F_{DEV}}{\partial \Delta_{DEV}} \neq 0$. If we can define an algebra for the types of circuit elements for digital logic – that is, can we define an algebra for logic – we can evaluate these equations symbolically to determine a solution for the diagnosis.

As an intuitive example, suppose we have made some measurements on the circuit and determine that both the SUM and the CAR outputs sometimes give bad results. By examining the dependencies, we may say that the possible diagnosis for this failure is a subset of the set whose elements are the sets that contain the union of all pairs of subsets of the two sets $\{X_1, X_2\}$ and $\{O_1, A_1, A_2, X_1\}$. That is, $\forall S \subseteq \{X_1, X_2\}, \forall T \subseteq \{O_1, A_1, A_2, X_1\}, DIAG \in S \cup T$. For example, $\{X_1\}$, $\{X_2, O_1\}$, $\{X_2, A_2\}$, $\{X_2, A_1\}$ are some possible diagnoses. Note that the last one falls into the class of solutions which may not be allowable due to the lack of detail of the model. Therefore, it must be evaluated to determine that it does not apply.

5 An Abstract Algebra for Logic

In this section we will define the necessary abstract algebra to perform diagnosis on simple logical systems. The basic requirements are to define the standard logical operations, their properties, and the meaning of dependency and differentiation in the abstraction.

We define a **differential logic** as any system $\mathcal{L}$ on the binary numbers $\{0, 1\}$ having the closed binary operations $+$, $\cdot$ (typically omitted), partial differentiation (written $\frac{\partial a}{\partial t}$), and differentiation (written $\frac{da}{dt}$); having the closed unary operation NOT (written $\neg$); and having the binary relation DEPENDS (written as $a \triangleright b$ to mean that a depends on b and as $a \not\triangleright b$ to mean that a does not depend on b. For $\{a, b, c, F\} \in \{0, 1\}$, these operations and relations satisfy the following:

$$
\begin{aligned}
&& 0+0 &= 0 \\
&& 1+1 &= 1 \\
a+1 &= & 1+a &= 1 \\
a+0 &= & 0+a &= a \\
&& a+a &= a \\
a+b &= & b+a &
\end{aligned}
$$

$$
\begin{aligned}
&& 0 \cdot 0 &= 0 \\
&& 1 \cdot 1 &= 1 \\
a \cdot 1 &= & 1 \cdot a &= a \\
a \cdot 0 &= & 0 \cdot a &= 0 \\
&& a \cdot a &= a \\
a \cdot b &= & b \cdot a &
\end{aligned}
$$

$$
\begin{aligned}
\neg 1 &= 0 \\
\neg 0 &= 1 \\
\neg(a \cdot b) &= (\neg a) + (\neg b) \\
\neg(a + b) &= (\neg a) \cdot (\neg b) \\
a \cdot (b \cdot c) &= (a \cdot b) \cdot c \\
a \cdot (b + c) &= (a \cdot b) + (a \cdot c)
\end{aligned}
$$

$$
\begin{array}{llcl}
a \not\triangleright b & \Rightarrow \frac{\partial a}{\partial b} & = & 0 \\
a \triangleright b & \Rightarrow \frac{\partial a}{\partial b} & = & F_{op}(a,b) \\
a \triangleright b \triangleright c & \Rightarrow \frac{\partial a}{\partial c} & = & \frac{\partial a}{\partial b} \cdot \frac{\partial b}{\partial c} \\
F \triangleright \{a,b\} \triangleright c & \Rightarrow \frac{dF}{dc} & = & \frac{\partial F}{\partial a} \cdot \frac{da}{dc} + \frac{\partial F}{\partial b} \cdot \frac{db}{dc}
\end{array}
$$

The functions $F_{op}(a,b)$ are the behavioral models for each operation under differentiation. Definitions of these functions under the normal operational model of the device are shown below. For convenience, we include the exclusive or operation $a \oplus b \equiv (a \cdot \neg b) + (\neg a \cdot b)$.

$$
\begin{aligned}
\frac{\partial(a \cdot b)}{\partial a} &= b \cdot \neg \left(\frac{\partial b}{\partial a}\right) \\
\frac{\partial(a \cdot b)}{\partial b} &= a \cdot \neg \left(\frac{\partial a}{\partial b}\right) \\
\frac{\partial(a + b)}{\partial a} &= \neg b \cdot \neg \left(\frac{\partial b}{\partial a}\right) \\
\frac{\partial(a + b)}{\partial b} &= \neg a \cdot \neg \left(\frac{\partial a}{\partial b}\right) \\
\frac{\partial(\neg a)}{\partial a} &= 1 \\
\frac{\partial(a \oplus b)}{\partial a} &= \neg \left(\frac{\partial b}{\partial a}\right) \\
\frac{\partial(a \oplus b)}{\partial b} &= \neg \left(\frac{\partial a}{\partial b}\right)
\end{aligned}
$$

We may see how these are derived by considering an example. The table below shows the input/output relations for $a \cdot b$.

a	b	$a \cdot b$	$\frac{\partial(a \cdot b)}{\partial a}$	$\frac{\partial(a \cdot b)}{\partial b}$
0	0	0	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	1	1

Note that, if we hold b constant and vary a, then if the output, $a \cdot b$ changes, the partial derivative is 1. In order to actually use this idea, we must assure that b remains constant, and thus the terms $\neg(\frac{\partial b}{\partial a})$ and $\neg(\frac{\partial a}{\partial b})$.

5.1 The Approach to Diagnosis

To see the utility of this definition, we will continue exploring our full adder example

above. Suppose we have made some measurements on a system. How may we use these measurements to constrain our hypotheses of possible failure sets which can explain these measurements. Referring to the table of differentiation above, we see that a failed component may be modeled as a function of the input arguments in addition to an argument that selects the appropriate model for each failure mode. So a device having two inputs $\{a, b\}$, one output OUT, and one failure mode could be modeled as:

$$OUT = F(a, b, \Delta)$$

where Δ is a variable whose value is zero if the device is functioning normally and non-zero if it is operating in the failure mode. To reason about the failure of the entire circuit, we must try to determine the combination of $\frac{\partial F_{DEV}}{\partial \Delta_{DEV}}$'s for the various devices that would produce the failed behavior of the entire circuit. We may accomplish this in two stages. First we find the variation of the outputs as a result of varying the Δ's – that is we derive the general behavioral equations for the circuit. Secondly, we hypothesize various failure modes to determine if the exact behavior can be explained. Often we will find that performing the second step using a very simple model of the failure modes of the devices is sufficient to determine the circuit fault or faults (we note here that this approach handles multiple faults identically as it handles single ones). However, in complex multiple fault problems, we may have to resort to repeating the second step with a more complex failure model for the components. Although the repetition of the second step can be combined by using a complex model the first time, we purposely separate them since each iteration reduces the search space for the next. In general, the second step may be repeated using more complex models until a unique and consistent diagnosis is obtained.

The first step is accomplished in keeping with the style of automatic problem formulation as embodied in the **MONAD** system. The circuit is modeled as a set of interconnected components. The interconnection graph determines the dependency of the various nodes on the devices. The behavioral model is derived by performing variational analysis of the system by taking the derivative of each output with respect to the set of component Δ's[3]. Each derivative will result in an equation which, collected over all outputs, form the **behavioral model** for the system.

The second step is accomplished by examining the expected output with respect to measurements on the system. Substituting these measured values into the behavioral equations yields a set of inequalities based on the difference in the observed and expected behavior. More specifically, if we take the derivative of output X with respect to the set of devices, the result is an equation. It will be a sum of terms composed of the product of system dependent variables and partial derivatives of the form $\frac{\partial F_{DEV}(in_1, in_2, \ldots, in_n, \Delta_{DEV})}{\partial \Delta_{DEV}}$. For a given observation, the difference in the expected output and the observed output is set equal to the derivative of the output with respect to the devices since the devices

[3]For efficiency, of course one would use only the relevant outputs and the relevant components.

are assumed to be the cause of the difference. The set of such equations may be solved[4] **symbolically** to give information about the possible explanation of the observed system behavior. If the set is definitive, the process may be abandoned; otherwise, substitutions for specific failure models, $\frac{\partial F_{DEV}}{\partial \Delta_{DEV}}$, may be used to find a solution which predicts the observed behavior.

For example, suppose we believe that the outputs of each gate in the full adder circuit can either behave normally, malfunction all of the time, be stuck at 0, or be stuck at 1. For each operation above, the output is then modeled as the function $F_{DEV}(a, b, \Delta)$, where Δ is the variable which moves the output from normal behavior toward that of the failure mode. For our full adder example, a suitable set of failure models are shown below[5]:

Function $F(a,b)$	Normal Behavior $\frac{\partial F}{\partial a}$	Undefined Model $\frac{\partial F}{\partial a}$	$\frac{\partial F}{\partial \Delta}$	Stuck at Zero $\frac{\partial F}{\partial \Delta}$	Stuck at One $\frac{\partial F}{\partial \Delta}$
$a \cdot b$	$b \cdot \neg(\frac{\partial b}{\partial a})$	$b \cdot \neg(\frac{\partial b}{\partial a})$	∇	$(a \cdot b) \cdot \nabla$	$\neg(a \cdot b) \cdot \nabla$
$a + b$	$\neg b \cdot \neg(\frac{\partial b}{\partial a})$	$\neg b \cdot \neg(\frac{\partial b}{\partial a})$	∇	$(a + b) \cdot \nabla$	$\neg(a + b) \cdot \nabla$
$\neg a$	1	1	∇	$\neg a \cdot \nabla$	$a \cdot \nabla$
$a \oplus b$	$\neg(\frac{\partial b}{\partial a})$	$\neg(\frac{\partial b}{\partial a})$	∇	$(a \oplus b) \cdot \nabla$	$\neg(a \oplus b) \cdot \nabla$

Note that for *stuck at 0* and *stuck at 1*, $\frac{\partial F}{\partial a} = \frac{\partial F}{\partial b} = 0$. We have used the variable $\nabla = 1$ to indicate that a given failure mode for a device is selected. This variable is generally used in the corresponding equation for $\frac{\partial F}{\partial \Delta}$.

Given a set of models m_i such as those above for the failure modes of the device, then a total model M_{total} may be composed of several failure modes at once using the relation:

$$M_{total} = \sum_n m_i \cdot \nabla^i_{DEV} \tag{5}$$

where the ∇^i_{DEV}[6] select the possible modes of failure for the device as appropriate. Mutual exclusion of models such as *stuck at 0* and *stuck at 1* must be accounted in the allowable ∇ vectors which define the search space for the solution. If we choose no model for the failure – that is, the output is always wrong – we may not be able to discern when a measurement can appear correct even though a device is faulty.

[4]Note, by solve we mean perform a solution process which gives information about the allowable constraint variables. **No solution** is an allowable return from this process.

[5]They are symmetrical about (a, b).

[6]Note that the superscript does not indicate exponentiation but simply the vector element.

5.2 Diagnostic Equations

As we have said, in order to form a diagnosis, we will define a set of inequalities which constrain the possible search space and provide a decision function for the diagnosis. If all the devices are functioning normally, then the derivatives $\frac{\partial F_i}{\partial \Delta_j} = 0$ for all outputs F_i and device failure vectors ∇j. If one or more devices malfunction, then, zero or more of the derivatives $\frac{\partial F_i}{\partial \Delta_j}$ may become non-zero. In some cases the system inputs or the functional dependencies of the outputs may cause the outputs to still remain as expected under the normal behavioral conditions due to a poor choice of device failure models. So in general, we may form two types of constraint equations based on the observed measurements. If the measurements agree with the expected behavior given the system inputs and state, then the system *may* be fault free. If the measurements do not agree, then they can be used to form an upper bound on the set of components which may be causing the failure if we know the dependencies.

Returning to our full adder example, suppose we choose the "unknown model" for the component behavior. That is, if a device fails, its output is always incorrect. Given the system behavioral model which we repeat below:

$$\frac{dSUM}{dDEV} = \frac{\partial F_{X_2}}{\partial \Delta_{X_2}} + \frac{\partial F_{X_2}}{\partial d}\frac{\partial F_{X_1}}{\partial \Delta_{X_1}}$$

and

$$\frac{dCAR}{dDEV} = \frac{\partial F_{O_1}}{\partial \Delta_{O_1}} + \frac{\partial F_{O_1}}{\partial e}\frac{\partial F_{A_1}}{\partial \Delta_{A_1}} + \frac{\partial F_{O_1}}{\partial f}\left[\frac{\partial F_{A_2}}{\partial \Delta_{A_2}} + \frac{\partial F_{A_2}}{\partial d}\frac{\partial F_{X_1}}{\partial \Delta_{X_1}}\right]$$

We first compute the partial derivatives for the devices with respect to the set of all devices (DEV) as shown in the table below:

$$\frac{\partial F_{X_2}}{\partial d} = \neg\frac{\partial c}{\partial DEV} \qquad \frac{\partial F_{O_1}}{\partial e} = \neg f \cdot \left(\neg\frac{\partial f}{\partial DEV}\right)$$

$$\frac{\partial c}{\partial DEV} = 0 \qquad \frac{\partial f}{\partial DEV} = \nabla_{A_2} + c \cdot \nabla_{X_1}$$

$$\frac{\partial F_{X_2}}{\partial d} = 1 \qquad \frac{\partial F_{O_1}}{\partial e} = \neg f \cdot \neg\nabla_{A_2} \cdot \neg\left(c \cdot \nabla_{A_1}\right)$$

$$\frac{\partial F_{A_2}}{\partial d} = c \cdot \left(\neg\frac{\partial c}{\partial DEV}\right) \qquad \frac{\partial F_{O_1}}{\partial f} = \neg e \cdot \left(\neg\frac{\partial e}{\partial DEV}\right)$$

$$\frac{\partial c}{\partial DEV} = 0 \qquad \frac{\partial e}{\partial DEV} = \nabla_{A_1}$$

$$\frac{\partial F_{A_2}}{\partial d} = c \qquad \frac{\partial F_{O_1}}{\partial f} = \neg e \cdot \neg\nabla_{A_1}$$

Note that since $\{a, b, c\}$ are held constant at the observation values, the independent variables are only $\nabla_{DEV} \in DEV$. These derivatives may be substituted into the behavioral equations giving the **operational equations** for the full adder:

$$\frac{dSUM}{dDEV} = \nabla_{X_2} + \nabla_{X_1}$$

and $$\frac{dCAR}{dDEV} = \nabla_{O_1} + \neg f \cdot \neg\nabla_{A_2} \cdot \neg\left(c \cdot \nabla_{X_1}\right) \cdot \nabla_{A_1} + \neg e \cdot \neg\nabla_{A_1} \cdot \left[\nabla_{A_2} + c \cdot \nabla_{X_1}\right]$$

Now, suppose we have made the following measurements on the full adder circuit:

							Observed		Expected			
Measurement	a	b	c	d	e	f	SUM	CAR	SUM	CAR	$\frac{dSUM}{dDEV}$	$\frac{dCAR}{dDEV}$
Ob_1	1	0	1	1	0	1	1	0	0	1	1	1
Ob_2	1	0	0	1	0	0	0	0	1	0	1	0

From the columns containing $\frac{\partial SUM}{\partial DEV}$ and $\frac{\partial CAR}{\partial DEV}$ we see that the first measurement can be used to construct two constraint equations of the form:

$$0 \neq \left.\frac{dSUM}{dDEV}\right|_{Ob_1} = \left.\nabla_{X_2} + \nabla_{X_1}\right|_{Ob_1}$$

and $$0 \neq \left.\frac{dCAR}{dDEV}\right|_{Ob_1} = \nabla_{O_1} + \neg f \cdot \neg\nabla_{A_2} \cdot \neg\left(c \cdot \nabla_{X_1}\right) \cdot \nabla_{A_1} + \neg e \cdot \neg\nabla_{A_1} \cdot \left.\left(\nabla_{A_2} + c \cdot \nabla_{X_1}\right)\right|_{Ob_1}$$

To evaluate the measurement Ob_1, we substitute the values $c = 1$, $\neg e = 1$, and $\neg f = 0$ for the measurement into the system behavioral equations giving:

$$0 \neq \left.\frac{dSUM}{dDEV}\right|_{Ob_1} = \nabla_{X_2} + \nabla_{X_1}$$

and $$0 \neq \left.\frac{dCAR}{dDEV}\right|_{Ob_1} = \nabla_{O_1} + \neg\nabla_{A_1} \cdot \left[\nabla_{A_2} + \nabla_{X_1}\right]$$

We may evaluate the operational equations for the second measurement Ob_2 in a similar fashion. The result is the third equation below since the equation for $\left.\frac{dSUM}{dDEV}\right|_{Ob_2}$ is a duplicate of $\left.\frac{dSUM}{dDEV}\right|_{Ob_1}$. Thus, the following three equations form a set of diagnostic constraint equations which we can solve in order to deduce information about the diagnosis.

$$0 \neq \left.\frac{dSUM}{dDEV}\right|_{Ob_1} = \nabla_{X_2} + \nabla_{X_1}$$

and $$0 \neq \left.\frac{dCAR}{dDEV}\right|_{Ob_1} = \nabla_{O_1} + \neg\nabla_{A_1} \cdot \left[\nabla_{A_2} + \nabla_{X_1}\right]$$

and $$0 = \left.\frac{dCAR}{dDEV}\right|_{Ob_2} = \nabla_{O_1} + \neg\nabla_{A_2} \cdot \nabla_{A_1} + \neg\nabla_{A_1} \cdot \nabla_{A_2}$$

Notice that these equations could be inconsistent since our failure model could be incorrect. This directly addresses the problem cited earlier regarding the completeness of the model. We are not requiring a complete or correct model to formulate the problem. Rather, we have formulated the problem at a higher level: the behavioral equations which are the differential equations for the variation of the measurements – which is independent of the failure models. These equations are the **problem formulation** of this particular full adder problem. Clearly, the system behavioral equations may be easily evaluated to obtain other particular solutions. The important point is that these equations are formulated in such a way that one of the embedded subsystems in the **MONAD** system – the constraint subsystem – can attempt a solution without interference from the remainder of the system. If no solution is found, the task of reformulation is not handled within the constraint subsystem but rather by the current process driving the relation method.

Returning to the full adder diagnosis, a solution set may be obtained by assuming that these equations hold. That is, by assuming that our failure model for the devices is correct, we can conclude that $\nabla_{O_1} = \neg\nabla_{A_2} \cdot \nabla_{A_1} = \neg\nabla_{A_1} \cdot \nabla_{A_2} = 0$, thereby reducing the second inequality to $0 \neq \neg\nabla_{A_1} \cdot \nabla_{X_1} \Rightarrow \nabla_{X_1} = 1, \nabla_{A_1} = 0$. From $\neg\nabla_{A_1} \cdot \nabla_{A_2} = 0$ we get $\nabla_{A_2} = 0$, and thus, the diagnostic interpretation of these equations is that device $\{X_1\}$ is defective, $\{O_1, A_1, A_2\}$ are good, and device $\{X_2\}$ is unknown under this failure model.

6 Applying Specific Failure Models

The notion of refining the diagnosis bound may be seen more clearly by an example. We will show how to substitute specific failure models into the behavioral equations in order to test a hypothesis. As we have already said, we are at liberty to apply a complex failure model from the outset. But a refinement approach can be useful when the problem is too large for this to be practical.

In the previous analysis, we have first substituted the broadest model – the unknown model – which has the simplifying characteristic that $\frac{\partial F_{DEV}}{\partial \Delta_{DEV}} = 1 \cdot \nabla$ for all devices. Now we will create a failure model in which the specific failure mode *stuck at 0* is tested. As before, ∇_{mode} is the mode selector for each operating mode of the device such that the device assumes that mode when $\nabla \neq 0$. Typically, we will assume that failure modes for a device are mutually exclusive, that is $0 = \sum_{i,j|i\neq j} \nabla_i \cdot \neg\nabla_j$ and $0 \neq \sum_i \nabla_i$. We will define two mode variables for each device: ∇_{DEV}^{NORM} and ∇_{DEV}^{SAZ} where the first selects normal behavior and the second selects the output *stuck at 0* mode.

Recall that the behavioral equations for the full adder are:

$$\frac{dSUM}{dDEV} = \frac{\partial F_{X_2}}{\partial \Delta_{X_2}} + \frac{\partial F_{X_2}}{\partial d}\frac{\partial F_{X_1}}{\partial \Delta_{X_1}}$$

and

$$\frac{dCAR}{dDEV} = \frac{\partial F_{O_1}}{\partial \Delta_{O_1}} + \frac{\partial F_{O_1}}{\partial e}\frac{\partial F_{A_1}}{\partial \Delta_{A_1}} + \frac{\partial F_{O_1}}{\partial f}\left[\frac{\partial F_{A_2}}{\partial \Delta_{A_2}} + \frac{\partial F_{A_2}}{\partial d}\frac{\partial F_{X_1}}{\partial \Delta_{X_1}}\right]$$

$F(a,b)$	Normal $\frac{\partial F}{\partial a}$	Stuck at Zero $\frac{\partial F}{\partial \Delta}$
$a \cdot b$	$\left(b \cdot \neg(\frac{\partial b}{\partial a})\right) \cdot \nabla_{DEV}^{NORM}$	$(a \cdot b) \cdot \nabla_{DEV}^{SAZ}$
$a + b$	$\left(\neg b \cdot \neg(\frac{\partial b}{\partial a})\right) \cdot \nabla_{DEV}^{NORM}$	$(a + b) \cdot \nabla_{DEV}^{SAZ}$
$\neg a$	∇_{DEV}^{NORM}	$(\neg a) \cdot \nabla_{DEV}^{SAZ}$
$a \oplus b$	$\neg\left(\frac{\partial b}{\partial a}\right) \cdot \nabla_{DEV}^{NORM}$	$(a \oplus b) \cdot \nabla_{DEV}^{SAZ}$

Substituting into the behavioral equations as before:

$$\frac{dSUM}{dDEV} = (d \oplus c) \cdot \nabla_{X_2}^{SAZ} + \left(\neg\frac{\partial c}{\partial d}\right) \cdot \nabla_{X_2}^{NORM} \cdot (a \oplus b) \cdot \nabla_{X_1}^{SAZ}$$

$$\begin{aligned}\frac{dCAR}{dDEV} &= (e+f) \cdot \nabla_{O_1}^{SAZ} + (\neg f) \cdot \left(\neg\frac{\partial f}{\partial e}\right) \cdot \nabla_{O_1}^{NORM} \cdot (a \cdot b) \cdot \nabla_{A_1}^{SAZ} \\ &\quad + (\neg e) \cdot \left(\neg\frac{\partial e}{\partial f}\right) \cdot \nabla_{O_1}^{NORM} \cdot \left[(d \cdot c) \cdot \nabla_{A_2}^{SAZ} + (c) \cdot \left(\neg\frac{\partial c}{\partial d}\right) \cdot \nabla_{A_2}^{NORM} \cdot (a \oplus b) \cdot \nabla_{X_1}^{SAZ}\right]\end{aligned}$$

where $\nabla_{DEV}^{SAZ} = \neg\nabla_{DEV}^{NORM}$. Forming the partial derivatives with respect to all the devices as before:

$$\begin{aligned}\neg\left(\frac{\partial f}{\partial e}\right) &= \nabla_{A_2}^{NORM} \cdot \left(\neg c + \nabla_{X_1}^{NORM}\right) \\ \neg\left(\frac{\partial e}{\partial f}\right) &= \nabla_{A_1}^{NORM} \\ \neg\left(\frac{\partial c}{\partial d}\right) &= 1\end{aligned}$$

which we may substitute into the operational equations above giving:

$$\frac{dSUM}{dDEV} = (d \oplus c) \cdot \nabla_{X_2}^{SAZ} + \nabla_{X_2}^{NORM} \cdot (a \oplus b) \cdot \nabla_{X_1}^{SAZ}$$

$$\begin{aligned}\frac{dCAR}{dDEV} &= (e+f) \cdot \nabla_{O_1}^{SAZ} + (\neg f) \cdot \nabla_{A_2}^{NORM} \cdot \left(\neg c + \nabla_{X_1}^{NORM}\right) \cdot \nabla_{O_1}^{NORM} \cdot (a \cdot b) \cdot \nabla_{A_1}^{SAZ} \\ &\quad + (\neg e) \cdot \nabla_{A_1}^{NORM} \cdot \nabla_{O_1}^{NORM} \cdot \left[(d \cdot c) \cdot \nabla_{A_2}^{SAZ} + (c) \cdot \nabla_{A_2}^{NORM} \cdot (a \oplus b) \cdot \nabla_{X_1}^{SAZ}\right]\end{aligned}$$

These operational equations may be evaluated at each of the measurements to obtain the diagnostic constraint equations as before:

$$0 \neq \frac{dSUM}{dDEV}\Big|_{Ob_1} = (1 \oplus 1) \cdot \nabla_{X_2}^{SAZ} + \nabla_{X_2}^{NORM} \cdot (1 \oplus 0) \cdot \nabla_{X_1}^{SAZ}$$

$$0 \neq \frac{dCAR}{dDEV}\Big|_{Ob_1} = (0+1) \cdot \nabla_{O_1}^{SAZ} + (\neg 1) \cdot \nabla_{A_2}^{NORM} \cdot (\neg 1 + \nabla_{X_1}^{NORM}) \cdot \nabla_{O_1}^{NORM} \cdot (1 \cdot 0) \cdot \nabla_{A_1}^{SAZ} + (\neg 0) \cdot \nabla_{A_1}^{NORM} \cdot \nabla_{O_1}^{NORM} \cdot [(1 \cdot 1) \cdot \nabla_{A_2}^{SAZ} + (1) \cdot \nabla_{A_2}^{NORM} \cdot (1 \oplus 0) \cdot \nabla_{X_1}^{SAZ}]$$

$$0 \neq \frac{dSUM}{dDEV}\Big|_{Ob_2} = (1 \oplus 0) \cdot \nabla_{X_2}^{SAZ} + \nabla_{X_2}^{NORM} \cdot (1 \oplus 0) \cdot \nabla_{X_1}^{SAZ}$$

$$0 = \frac{dCAR}{dDEV}\Big|_{Ob_2} = (0+0) \cdot \nabla_{O_1}^{SAZ} + (\neg 0) \cdot \nabla_{A_2}^{NORM} \cdot (\neg 0 + \nabla_{X_1}^{NORM}) \cdot \nabla_{O_1}^{NORM} \cdot (1 \cdot 0) \cdot \nabla_{A_1}^{SAZ} + (\neg 0) \cdot \nabla_{A_1}^{NORM} \cdot \nabla_{O_1}^{NORM} \cdot [(1 \cdot 0) \cdot \nabla_{A_2}^{SAZ} + (0) \cdot \nabla_{A_2}^{NORM} \cdot (1 \oplus 0) \cdot \nabla_{X_1}^{SAZ}]$$

which reduce to:

$$0 \neq \frac{dSUM}{dDEV}\Big|_{Ob_1} = \nabla_{X_2}^{NORM} \cdot \nabla_{X_1}^{SAZ}$$

$$0 \neq \frac{dCAR}{dDEV}\Big|_{Ob_1} = \nabla_{O_1}^{SAZ} + \nabla_{A_1}^{NORM} \cdot \nabla_{O_1}^{NORM} \cdot \left[\nabla_{A_2}^{SAZ} + \nabla_{A_2}^{NORM} \cdot \nabla_{X_1}^{SAZ}\right]$$

$$0 \neq \frac{dSUM}{dDEV}\Big|_{Ob_2} = \nabla_{X_2}^{SAZ} + \nabla_{X_2}^{NORM} \cdot \nabla_{X_1}^{SAZ}$$

$$0 = \frac{dCAR}{dDEV}\Big|_{Ob_2} = 0$$

Again our model is consistent and we have learned, further, that we may not exclude $\{O_1, A_2, X_1\}$ as possibly faulty under this model and given the measurements Ob_1 and Ob_2. Now, if we take an additional measurement:

Measurement	a	b	c	d	e	f	Observed SUM	Observed CAR	Expected SUM	Expected CAR	$\frac{dSUM}{dDEV}$	$\frac{dCAR}{dDEV}$
Ob_3	1	1	0	1	0	1	1	1	1	1	1	1

We may write:

$$0 = \frac{dSUM}{dDEV} = (0 \oplus 1) \cdot \nabla_{X_2}^{SAZ} + \nabla_{X_2}^{NORM} \cdot (1 \oplus 1) \cdot \nabla_{X_1}^{SAZ}$$

$$0 = \frac{dCAR}{dDEV} = (1+0) \cdot \nabla_{O_1}^{SAZ} + (\neg 0) \cdot \nabla_{A_2}^{NORM} \cdot (\neg 0 + \nabla_{X_1}^{NORM}) \cdot \nabla_{O_1}^{NORM} \cdot (1 \cdot 1) \cdot \nabla_{A_1}^{SAZ} + (\neg 1) \cdot \nabla_{A_1}^{NORM} \cdot \nabla_{O_1}^{NORM} \cdot [(0 \cdot 1) \cdot \nabla_{A_2}^{SAZ} + (1) \cdot \nabla_{A_2}^{NORM} \cdot (1 \oplus 1) \cdot \nabla_{X_1}^{SAZ}]$$

which reduces to:

$$0 = \frac{dSUM}{dDEV} = \nabla_{X_2}^{SAZ}$$

$$0 = \frac{dCAR}{dDEV} = \nabla_{O_1}^{SAZ} + \nabla_{A_2}^{NORM} \cdot \nabla_{O_1}^{NORM} \cdot \nabla_{A_1}^{SAZ}$$

Note that the solution of these equations would normally be undertaken by the constraint satisfaction subsystem. We will reason through them here for illustration.

If we let $\nabla_{O_1}^{SAZ} = \nabla_{X_2}^{SAZ} = 0$, then, recalling that ∇_{DEV}^{SAZ} and ∇_{DEV}^{NORM} are mutually exclusive which means that $\nabla_{O_1}^{NORM} = \nabla_{X_2}^{NORM} = 1$ then, substituting into the first equation, we find that $\nabla_{X_1}^{SAZ} = 1$ and $\nabla_{X_1}^{NORM} = 0$. Substituting these into the inequalities we get:

$$0 \neq \left.\frac{dSUM}{dDEV}\right|_{Ob_1} = 1 \cdot 1$$

$$0 \neq \left.\frac{dCAR}{dDEV}\right|_{Ob_1} = 0 + \nabla_{A_1}^{NORM} \cdot 1 \cdot \left[\nabla_{A_2}^{SAZ} + \nabla_{A_2}^{NORM} \cdot 1\right]$$

$$0 \neq \left.\frac{dSUM}{dDEV}\right|_{Ob_2} = 0 + 1 \cdot 1$$

From the second equation, we conclude that $\nabla_{A_1}^{NORM} = 1 \Rightarrow \nabla_{A_1}^{SAZ} = 0$, reducing the second equation to:

$$0 \neq \nabla_{A_2}^{SAZ} + \nabla_{A_2}^{NORM}$$

which simply means that we have not tested A_2. Thus, the diagnosis is that $\{X_1\}$ is *stuck at 0*, $\{A_2\}$ is unknown, and $\{O_1, A_1, X_2\}$ are all good under this fault model.

7 An Abstract Example

To show the diversity of this technique, we now turn to an abstract example. Figure 4 shows a diagram of the multiplier-adder circuit shown previously in figure 1. One set of observations of the measured behavior are shown as values on the various nodes. First we would like to consider the problem of making various deductions about the functionality of this circuit without actually considering its detail. Figure 5 shows a diagram of the circuit device classification. The table below shows the corresponding dependencies for the devices:

$$\begin{aligned}
\{X\} &\triangleright \{F_{A_1}\} \\
\{F_{A_1}\} &\triangleright \{j, k, \triangle_{A_1}\} \\
\{j\} &\triangleright \{F_{M_1}\} \\
\{F_{M_1}\} &\triangleright \{a, b, \triangle_{M_1}\} \\
\{k\} &\triangleright \{F_{M_2}\} \\
\{F_{M_2}\} &\triangleright \{c, d, \triangle_{M_2}\} \\
\{Y\} &\triangleright \{F_{A_2}\} \\
\{F_{A_2}\} &\triangleright \{k, l, \triangle_{A_2}\} \\
\{l\} &\triangleright \{F_{M_3}\} \\
\{F_{M_3}\} &\triangleright \{e, f, \triangle_{M_3}\}
\end{aligned}$$

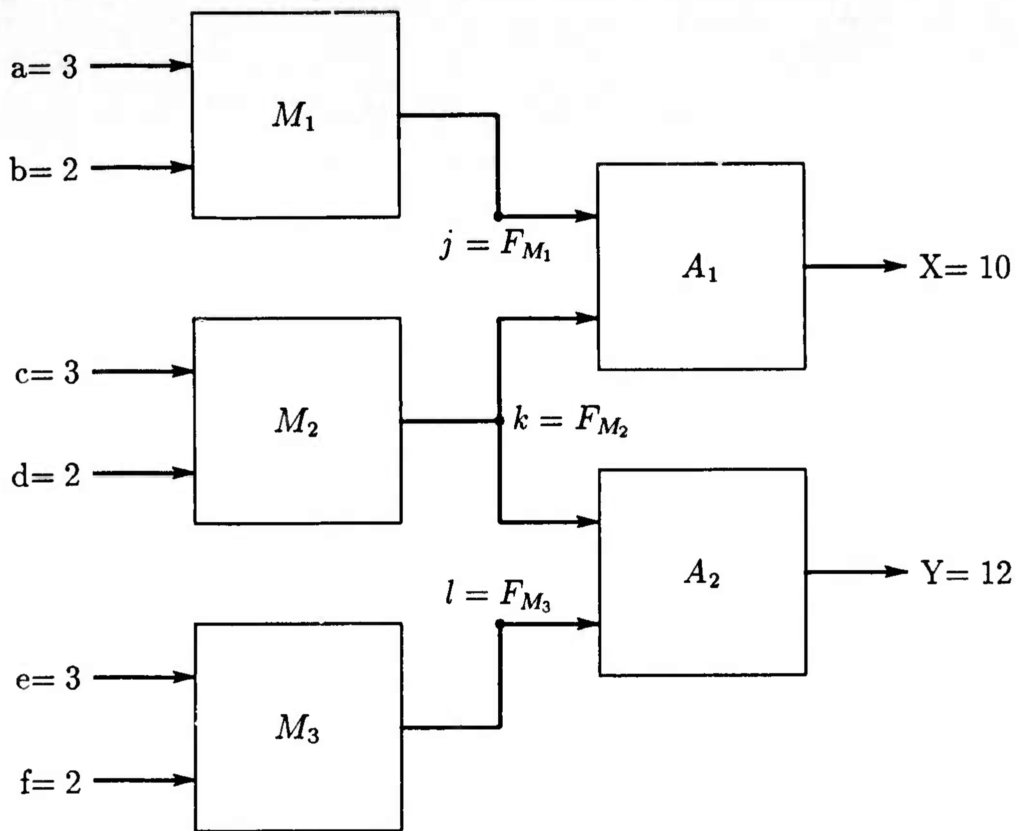


Figure 4: Multiplier Adder Observation Set

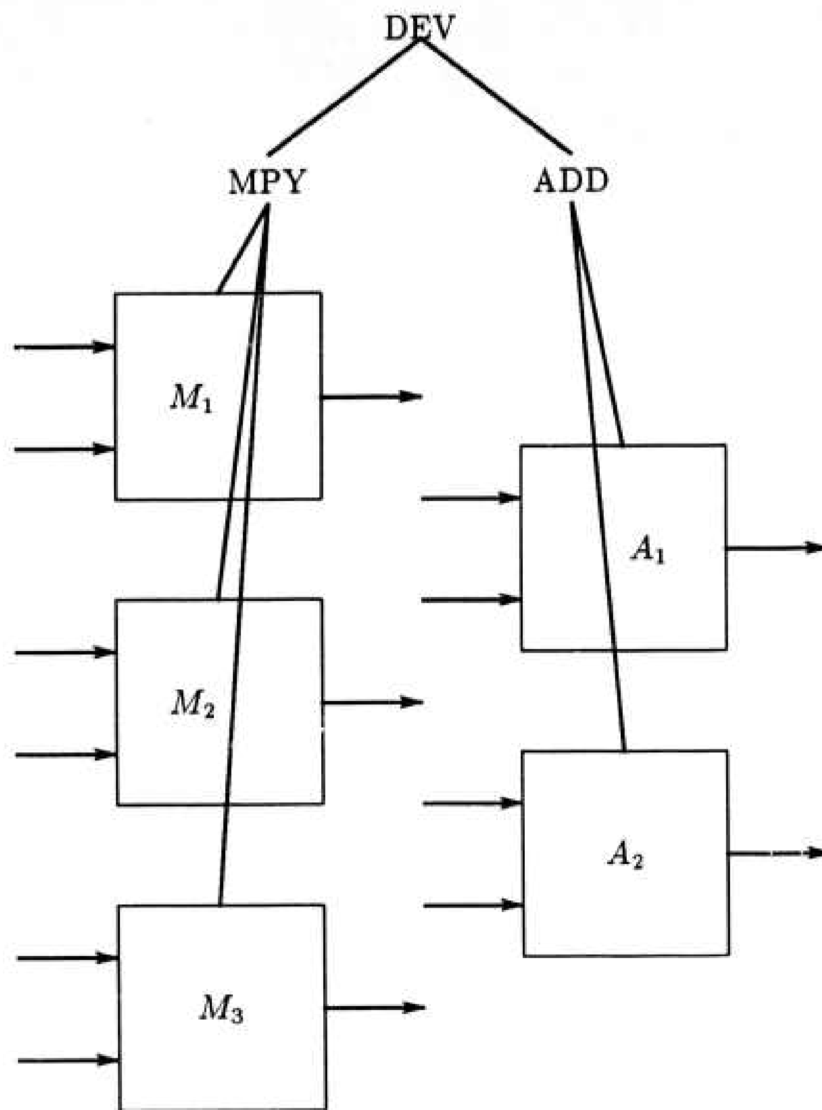


Figure 5: Multiplier Adder Device Dependency

$$\{\triangle_{M_1}, \triangle_{M_2}, \triangle_{M_3}\} \triangleright \{MPY\}$$
$$\{\triangle_{A_1}, \triangle_{A_2}\} \triangleright \{ADD\}$$
$$\{MPY, ADD\} \triangleright \{DEV\}$$

The corresponding behavioral model is derived as:

$$\frac{dX}{dDEV} = \frac{\partial X}{\partial F_{A_1}}\left[\frac{\partial F_{A_1}}{\partial k}\frac{\partial k}{\partial F_{M_2}}\frac{\partial F_{M_2}}{\partial \triangle_{M_2}}\frac{\partial \triangle_{M_2}}{\partial MPY}\frac{dMPY}{dDEV} + \frac{\partial F_{A_1}}{\partial j}\frac{\partial j}{\partial F_{M_1}}\frac{\partial F_{M_1}}{\partial \triangle_{M_1}}\frac{\partial \triangle_{M_1}}{\partial MPY}\frac{dMPY}{dDEV} + \frac{\partial F_{A_1}}{\partial \triangle_{A_1}}\frac{\partial \triangle_{A_1}}{\partial ADD}\frac{dADD}{dDEV}\right]$$

and

$$\frac{dY}{dDEV} = \frac{\partial Y}{\partial F_{A_2}}\left[\frac{\partial F_{A_2}}{\partial k}\frac{\partial k}{\partial F_{M_2}}\frac{\partial F_{M_2}}{\partial \triangle_{M_2}}\frac{\partial \triangle_{M_2}}{\partial MPY}\frac{dMPY}{dDEV} + \frac{\partial F_{A_2}}{\partial l}\frac{\partial l}{\partial F_{M_3}}\frac{\partial F_{M_3}}{\partial \triangle_{M_3}}\frac{\partial \triangle_{M_3}}{\partial MPY}\frac{dMPY}{dDEV} + \frac{\partial F_{A_2}}{\partial \triangle_{A_2}}\frac{\partial \triangle_{A_2}}{\partial ADD}\frac{dADD}{dDEV}\right]$$

By definition of a connection:

$$1 = \frac{\partial X}{\partial F_{A_1}} = \frac{\partial Y}{\partial F_{A_2}}$$
$$1 = \frac{\partial j}{\partial F_{M_1}} = \frac{\partial k}{\partial F_{M_2}} = \frac{\partial l}{\partial F_{M_3}}$$
$$1 = \frac{\partial \triangle_{M_1}}{\partial MPY} = \frac{\partial \triangle_{M_2}}{\partial MPY} = \frac{\partial \triangle_{M_3}}{\partial MPY}$$
$$1 = \frac{\partial F_{A_1}}{\partial ADD} = \frac{\partial F_{A_2}}{\partial ADD} = \frac{dMPY}{dDEV} = \frac{dADD}{dDEV}$$

Substituting these into the above we get:

$$\frac{dX}{dDEV} = \frac{\partial F_{A_1}}{\partial k}\frac{\partial F_{M_2}}{\partial \triangle_{M_2}} + \frac{\partial F_{A_1}}{\partial j}\frac{\partial F_{M_1}}{\partial \triangle_{M_1}} + \frac{\partial F_{A_1}}{\partial \triangle_{A_1}}$$

and

$$\frac{dY}{dDEV} = \frac{\partial F_{A_2}}{\partial k}\frac{\partial F_{M_2}}{\partial \triangle_{M_2}} + \frac{\partial F_{A_2}}{\partial l}\frac{\partial F_{M_3}}{\partial \triangle_{M_3}} + \frac{\partial F_{A_2}}{\partial \triangle_{A_2}}$$

7.1 Diagnosis by Dependency

From simply examining the device dependencies we may make conclusions similar to those made about the full adder. For example, if our information is that the X output seems to

produce the wrong answer sometimes while the Y output is always correct, then we may write:

$$
\begin{aligned}
0 \neq \frac{dX}{dDEV} &= \frac{\partial F_{A_1}}{\partial k}\frac{\partial F_{M_2}}{\partial \Delta_{M_2}} + \frac{\partial F_{A_1}}{\partial j}\frac{\partial F_{M_1}}{\partial \Delta_{M_1}} + \frac{\partial F_{A_1}}{\partial \Delta_{A_1}} \\
\text{and} \quad 0 = \frac{dY}{dDEV} &= \frac{\partial F_{A_2}}{\partial k}\frac{\partial F_{M_2}}{\partial \Delta_{M_2}} + \frac{\partial F_{A_2}}{\partial l}\frac{\partial F_{M_3}}{\partial \Delta_{M_3}} + \frac{\partial F_{A_2}}{\partial \Delta_{A_2}}
\end{aligned}
$$

From the second equation, we can hypothesize that $\frac{\partial F_{M_2}}{\partial \Delta_{M_2}} = \frac{\partial F_{M_3}}{\partial \Delta_{M_3}} = \frac{\partial F_{A_2}}{\partial \Delta_{A_2}} = 0$ which means that $\{M_2, M_3, A_2\}$ are all good. This reduces the first equation to $0 \neq \frac{dX}{dDEV} = \frac{\partial F_{A_1}}{\partial j}\frac{\partial F_{M_1}}{\partial \Delta_{M_1}} + \frac{\partial F_{A_1}}{\partial \Delta_{A_1}}$. By using the simple failure model that a device failure is always observable: $\frac{\partial F_{A_1}}{\partial j} = 1$, then we have $0 \neq \frac{dX}{dDEV} = \frac{\partial F_{M_1}}{\partial \Delta_{M_1}} + \frac{\partial F_{A_1}}{\partial \Delta_{A_1}}$ which means that devices either or both of $\{M_1, A_1\}$ could be faulty.

7.2 Diagnosis Using an Abstract Model

Suppose we wish to decide which device is faulty but we have no detailed information about the functional behavior of the multipliers and adders other than the mathematical notions of their purpose. So for example, the multiplier circuit might be modeled as:

$$
\begin{aligned}
j &= F_{M_1}(a, b, \Delta_{M_1}) &= a \cdot b \\
k &= F_{M_2}(c, d, \Delta_{M_2}) &= c \cdot d \\
l &= F_{M_3}(e, f, \Delta_{M_3}) &= e \cdot f \\
X &= F_{A_1}(j, k, \Delta_{A_1}) &= j + k \\
Y &= F_{A_2}(k, l, \Delta_{A_2}) &= k + l
\end{aligned}
$$

where, in this abstract model, + means addition, and $\cdot$ means multiplication in the conventional sense. Repeating the simplified behavioral equations:

$$
\begin{aligned}
\frac{dX}{dDEV} &= \frac{\partial F_{A_1}}{\partial k}\frac{\partial F_{M_2}}{\partial \Delta_{M_2}} + \frac{\partial F_{A_1}}{\partial j}\frac{\partial F_{M_1}}{\partial \Delta_{M_1}} + \frac{\partial F_{A_1}}{\partial \Delta_{A_1}} \\
\text{and} \quad \frac{dY}{dDEV} &= \frac{\partial F_{A_2}}{\partial k}\frac{\partial F_{M_2}}{\partial \Delta_{M_2}} + \frac{\partial F_{A_2}}{\partial l}\frac{\partial F_{M_3}}{\partial \Delta_{M_3}} + \frac{\partial F_{A_2}}{\partial \Delta_{A_2}}
\end{aligned}
$$

Taking the device model derivatives for normal behavior from the abstract device models we find:

$$
\begin{aligned}
\frac{\partial F_{A_1}}{\partial j} &= \frac{\partial (j+k)}{\partial j} &= 1 \\
\frac{\partial F_{A_1}}{\partial k} &= \frac{\partial (j+k)}{\partial k} &= 1 \\
\frac{\partial F_{A_2}}{\partial k} &= \frac{\partial (k+l)}{\partial k} &= 1 \\
\frac{\partial F_{A_2}}{\partial l} &= \frac{\partial (k+l)}{\partial l} &= 1
\end{aligned}
$$

Substituting these values into the above equations we have:

$$\frac{dX}{dDEV} = \frac{\partial F_{M_2}}{\partial \Delta_{M_2}} + \frac{\partial F_{M_1}}{\partial \Delta_{M_1}} + \frac{\partial F_{A_1}}{\partial \Delta_{A_1}}$$

$$\text{and} \quad \frac{dY}{dDEV} = \frac{\partial F_{M_2}}{\partial \Delta_{M_2}} + \frac{\partial F_{M_3}}{\partial \Delta_{M_3}} + \frac{\partial F_{A_2}}{\partial \Delta_{A_2}}$$

In order to perform diagnosis, we might specify a failure model based on the notion of the failure of a particular bit in either the input circuitry or the output circuitry. For example, to specify that bit i of the output (OUT_i) has failed we will assume that $\frac{\partial F_{DEV}}{\partial \Delta_{OUT_i}} \neq 0$. The corresponding failure model might be:

Device	Normal $\frac{\partial F}{\partial a}$	Bit i Stuck at 0 $\frac{\partial F}{\partial \Delta}$	Bit i Stuck at 1 $\frac{\partial F}{\partial \Delta}$
MPY	b	$DEV(OUT_i) \cdot (-2^i) \cdot \nabla_{DEV}$	$DEV(\neg OUT_i) \cdot (2^i) \cdot \nabla_{DEV}$
ADD	1	$DEV(OUT_i) \cdot (-2^i) \cdot \nabla_{DEV}$	$DEV(\neg OUT_i) \cdot (2^i) \cdot \nabla_{DEV}$

where $DEV(OUT_i)$ is a predicate indicating that output bit i of OUT of device DEV is set to a logical 1. Substituting these into the above equations we have:

$$\frac{dX}{dDEV} = M_1(OUT_1) \cdot (-2^1) \cdot \nabla_{M_1} + M_2(OUT_1) \cdot (-2^1) \cdot \nabla_{M_2} + A_1(OUT_1) \cdot (-2^1) \cdot \nabla_{A_1}$$

$$\frac{dY}{dDEV} = M_2(OUT_1) \cdot (-2^1) \cdot \nabla_{M_2} + M_3(OUT_1) \cdot (-2^1) \cdot \nabla_{M_3} + A_2(OUT_1) \cdot (-2^1) \cdot \nabla_{A_2}$$

Suppose that we take two measurements as in the full adder example:

							Observed		Expected				
Measurement	a	b	c	d	e	f	X	Y	X	Y	j	k	l
Ob_1	3	2	3	2	3	2	10	12	12	12	6	6	6
Ob_2	2	2	3	2	2	2	10	10	10	10	4	6	4

Then we may form the following inequalities:

$$-2 = \left.\frac{dX}{dDEV}\right|_{Ob_1} = 1 \cdot (-2) \cdot \nabla_{M_1} + 1 \cdot (-2) \cdot \nabla_{M_2}$$

$$0 = \left.\frac{dY}{dDEV}\right|_{Ob_1} = M_2(OUT_1) \cdot (-2) \cdot \nabla_{M_2} + M_3(OUT_1) \cdot (-2) \cdot \nabla_{M_3}$$

$$0 = \left.\frac{dX}{dDEV}\right|_{Ob_2} = 1 \cdot (-2) \cdot \nabla_{M_2} + 1 \cdot (-2) \cdot \nabla_{A_1}$$

$$0 = \left.\frac{dY}{dDEV}\right|_{Ob_2} = M_2(OUT_1) \cdot (-2) \cdot \nabla_{M_2} + A_2(OUT_1) \cdot (-2) \cdot \nabla_{A_2}$$

which are easily solved to give $\nabla_{M_1} = 1$ and all others being 0 which means that M_1 having a *stuck at 0* bit 1 could explain the measurements.

8 Diagnosis with Incomplete Information

One technique that good diagnosticians use is to reason about devices for which only incomplete state information is available. In the world of digital logic, for example, there is even a device for precisely that purpose. It allows the troubleshooter to inject a short pulse into the input of a device without regard for the state of that or other inputs. By observing a pulse on one or more of the outputs, it is concluded that the device is probably functioning. This heuristic is used in spite of the fact that the actual behavior of the circuit and/or its current state is not completely known. The prime reason that this heuristic is fairly successful is that the most common fault in digital devices is an output stuck at either 0 or 1. Therefore, if an output pin can be made to show activity, there is some chance that the device is good. In this section we will consider how to implement this weak method using the relation method.

The table below shows the activity which could be expected at the two output pins of the full adder given that the various inputs are momentarily toggled from their current state.

INPUT		*OUTPUT*		$\frac{\partial OUTPUT}{\partial a}$		$\frac{\partial OUTPUT}{\partial c}$	
abc	*def*	*SUM*	*CAR*	*SUM*	*CAR*	*SUM*	*CAR*
000	000	0	0	1	0	1	0
001	000	1	0	1	1	1	0
010	100	1	0	1	1	1	1
011	101	0	1	1	0	1	1
100	100	1	0	1	0	1	1
101	101	0	1	1	1	1	1
110	010	0	1	1	1	1	0
111	010	1	1	1	0	1	0

The partial derivatives $\frac{\partial SUM}{\partial a}$, $\frac{\partial CAR}{\partial a}$, $\frac{\partial SUM}{\partial c}$, and $\frac{\partial CAR}{\partial c}$ are simply the results that would be observed in a normally operating circuit if the a or c inputs were changed. We can compute these derivatives as follows:

$$
\begin{aligned}
\frac{\partial SUM}{\partial a} &= \frac{\partial F_{X_1}}{\partial a}\frac{\partial F_{X_2}}{\partial d} \\
&= \neg\frac{\partial b}{\partial a} \cdot \neg\frac{\partial a}{\partial b} \\
\frac{\partial SUM}{\partial a} &= 1
\end{aligned}
$$

and

$$
\begin{aligned}
\frac{\partial CAR}{\partial a} &= \frac{\partial F_{A_2}}{\partial d}\frac{\partial F_{O_1}}{\partial f}\frac{\partial F_{X_1}}{\partial a} + \frac{\partial F_{A_1}}{\partial a}\frac{\partial F_{O_1}}{\partial e} \\
&= c \cdot \neg\frac{\partial c}{\partial a} \cdot \neg e \cdot \neg\frac{\partial e}{\partial a} \cdot \neg\frac{\partial b}{\partial a} + b \cdot \neg\frac{\partial b}{\partial a} \cdot \neg f \cdot \neg\frac{\partial f}{\partial a}
\end{aligned}
$$

which may be symbolically simplified to

$$
\frac{\partial CAR}{\partial a} = b \oplus c
$$

These agree with the observed changes as shown in the previous table.

9 Conclusions

We have described a formal approach to **problem formulation** for the diagnosis of static, loop-free, digital logic circuits. The approach may be extended to the modeling of formal logic by their differential behavior for such tasks as the control of reasoning and knowledge state revision. The approach is easily extended to quasi-static systems by defining internal state for the models. Future reports on the current effort to formalize reasoning about dynamic processes will directly deal with time varying systems with loops.

The important point of this paper is that the approach is uniform in its use of the relation method of the **MONAD** system in which it is also used for more complex tasks such as reasoning by analogy. In these tasks, it is essential that various kinds of objects be treated uniformly by the reasoning processes so that the control of the reasoning process may be abstracted. Our work in the near future will address these issues.

References

[Bon82] Piero P. Bonissone. Outline of the design and implementation of a diesel electric engine troubleshooting aid. In *Proceeding of Expert Systems 82*, pages 68–72, Brunel University, Egham, England, September 1982.

[DW85] Johan DeKleer and Brian C. Williams. *Diagnosing Multiple Faults.* Technical Report, Xerox Palo Alto Research Center, 1985.

[Por87] G. B. Porter. *An Approach to Automated Problem Formulation.* Technical Report, General Electric Corp. R&D Center, April 1987. Working paper in Computer Science.

[Rei85] Raymond Reiter. *A Theory of Diagnosis from First Principles.* Technical Report 187/86, University of Toronto and The Canadian Institute for Advanced Study, Toronto, Canada, December 1985.