

MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

**DTIC** FILE COPY

4

AD-A189 774

**RADC-TR-86-218, Vol II, Part 2 (of two)**  
**Interim Report**  
**July 1987**



***NORTHEAST ARTIFICIAL INTELLIGENCE  
CONSORTIUM (NAIC)***  
***Review of Technical Tasks***

**Syracuse University**

**J. F. Allen, P. B. Berra, J. A. Biles, K. A. Bowen, S. E. Conry,  
W. B. Croft, V. R. Lesser, R. A. Meyer, J. W. Modestino, G. Nagy,  
S. Nirenburg, H. Rhody, J. E. Searleman, S. C. Shapiro, S. N. Srihari  
and B. Woolf**

**This effort was partially funded by the Laboratory Directors' Fund**

**APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED**

**DTIC**  
**ELECTE**  
**DEC 3 1 1987**  
**S H D**

**ROME AIR DEVELOPMENT CENTER**  
**Air Force Systems Command**  
**Griffiss Air Force Base, NY 13441-5700**

**87 12 21 030**

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

Because of the size of this volume, it has been separated into Volume II, Part 1 (pages 1 - 466) and Volume II, Part 2 (pages 467 - 936).

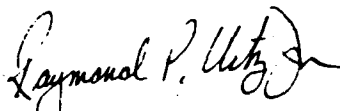
RADC-TR-86-218, Volume II, Part 2 (of two) has been reviewed and is approved for publication.

APPROVED:



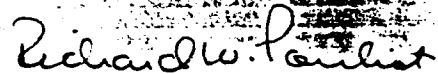
DONALD J. GONDEK  
NAIC Program Manager

APPROVED:



RAYMOND P. URTZ, Jr.  
Technical Director  
Directorate of Command & Control

FOR THE COMMANDER:



RICHARD W. POULIOT  
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS N/A	
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A		5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-86-218, Volume II, Part 2 (of two)	
6a. NAME OF PERFORMING ORGANIZATION Syracuse University ATTN: Dr. Volker Weiss, Director	6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)	
6c. ADDRESS (City, State, and ZIP Code) Northeast Artificial Intelligence Consortium 120 Hinds Hall Syracuse NY 13210		7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL (if applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0008	
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO 62702F (over)	PROJECT NO 2304
		TASK NO J5	WORK UNIT ACCESSION NO 01
11. TITLE (Include Security Classification) NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM (NAIC) Review of Technical Tasks			
12. PERSONAL AUTHOR(S) J.F. Allen, P.B. Berra, J.A. Biles, K.A. Bowen, S.E. Conry, W.B. Croft, V.R. Lesser (See reverse)			
13a. TYPE OF REPORT Interim	13b. TIME COVERED FROM Dec 84 TO Dec 85	14. DATE OF REPORT (Year, Month, Day) July 1987	15. PAGE COUNT 482
16. SUPPLEMENTARY NOTATION This effort was partially funded by the Laboratory Directors' Fund			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	Expert System, Distributed AI, Communications System Control, Image Interpretation, AI Planning, Knowledge-based Reasoning, Knowledge Acquisition, Belief Revision (See reverse)
12	05		
12	07		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made in the first year of the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photo interpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.)			
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Donald J. Gondek		22b. TELEPHONE (Include Area Code) (315) 330-7794	22c. OFFICE SYMBOL RADC (COES)

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted  
All other editions are obsoleteSECURITY CLASSIFICATION OF THIS PAGE  
UNCLASSIFIED

UNCLASSIFIED

Block 10 (Cont'd)	Project	Task	Work Unit
Program Element No.	No.	No.	Accession No.
61101F, 61102F,	5581	27	13
33126F,	4594	18	E2
	2155	02	10
	LDFP	15	C4

Block 12 (Cont'd)

R.A. Meyer, J.W. Modestino, G. Nagy, S. Nirenburg, H. Rhody, J.E. Searleman,  
S.C. Shapiro, S.N. Srihari and B. Woolf

Block 16 (Cont'd)

Beacuse of the size of this volume, it has been separated into Volume II, Part 1  
(pages 1 - 466) and Volume II, Part 2 (pages 467 - 936).

Block 18 (Cont'd)

Domain Knowledge, Service Restoral, Poplar, Procedural and Semantic Data Bases,  
Plan Recognition, Language Parsing, Interval-based Theory of Time, SNePS, Speech  
Understanding, AI Architecture, Knowledge-based Maintenance, Knowledge Explanation  
System, ThinkerToy

UNCLASSIFIED

### Acknowledgements

The authors wish to thank Dr. Northrup Fowler, Rome Air Development Center (RADC), who conceived the Northeast Artificial Intelligence Consortium (NAIC), Mr. Jake Scherer of RADC, who was instrumental in implementing the NAIC, and Mr. Donald Gondek, who, as Project Manager, rendered valuable assistance and communicated the needs of RADC. The authors are most appreciative of the encouragement, support, and friendly suggestions made by Mr. Fred I. Diamond, Chief Scientist of RADC, and Mr. Ray P. Urtz, Technical Director of the Command and Control Division of RADC. Special thanks are due to Dr. Bradley J. Strait and Mrs. Andrea Pflug of Syracuse University and to Mrs. Estella G. Bray of Clarkson University for their work in making the NAIC a success and in completing this report.



<b>Accession For</b>	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Avail and/or	
Dist	Special
A-1	

Table of Contents

Vol. II, Part 1

1	INTRODUCTION .....	1
1.1	The Northeast Artificial Intelligence Consortium .....	1
1.2	The Topics Under Study and the Principal Investigators (P.I.s) at Each Institution .....	1
2	VMES: A NETWORK-BASED VERSATILE MAINTENANCE EXPERT SYSTEM ( SUNY Buffalo ) .....	4
2.1	Device Modeling and Fault Diagnosis of VMES .....	5
2.2	Graphical Interface of VMES .....	18
	References .....	26
	Appendix 2-A: A Fault Diagnosis System Based on an Integrated Knowledge Base .....	27
	Appendix 2-B: A Logic for Belief Revision .....	31
	Appendix 2-C: A Model for Belief Revision .....	91
	Appendix 2-D: SNePS Considered as a Fully Intensional Propositional Semantic Network .....	179
	Appendix 2-E: Belief Revision in SNePS .....	227
3	DISTRIBUTED PROBLEM SOLVING ( Clarkson University ) .....	245
3.1	Introduction .....	246
3.2	Application Domain Description .....	247
3.3	System Architecture .....	251
3.4	Knowledge Representation .....	254
3.5	Problem Solving Strategies .....	260
3.6	Progress Review and Future Directions .....	264
	Bibliography .....	265
	Appendix 3-A Shared Knowledge Base for Independent Problem Solving Agents .....	267



4	PLANNER SYSTEM FOR THE APPLICATION OF INDICATIONS AND WARNING ( Colgate University ) .....	277
4.1	Introduction .....	278
4.2	Strategy .....	278
4.3	POPLAR 1.3 .....	279
4.4	POPLAR 2.0 .....	294
4.5	Background and Related Work .....	303
4.6	Status and Future Work .....	306
	Bibliography .....	308
	Appendix 1. Representation of Objects in POPLAR 1.3.....	311
	Appendix 2. Examples of PLAN representation in POPLAR 1.3 .....	315
	Appendix 3. Examples of POPLAR 1.3 rating functions .....	319
	Appendix 4. HISTORY in POPLAR 1.3 .....	321
	Appendix 5. BLACKBOARD in POPLAR 1.3 .....	321
	Appendix 6. Examples of PLAN representation in POPLAR 2.0 .....	323
	Appendix 7. Providing Intelligent Assistance in Distributed Office Environments .....	327
	Appendix 8. POPLAR: A Testbed for Cognitive Modeling .....	345
5	PLAN RECOGNITION, KNOWLEDGE ACQUISITION AND EXPLANATION IN AN INTELLIGENT INTERFACE ( University of Massachusetts ) .....	375
5.1	Basic Issues .....	378
5.2	Plan Recognition/Planning .....	379
5.3	Interface Tools .....	382
5.4	Discourse Processing in a Knowledge Acquisition Interface .....	389
5.5	Understanding Discourse Conventions in Tutoring .....	403
5.6	A Knowledge-Based Approach to Data Management for Intelligent User Interfaces .....	431
5.7	POISE Graphical Interface for Task Specification.....	453

Table of Contents

Vol. II, Part 2

6 AUTOMATIC PHOTO INTERPRETATION  
( Rensselaer Polytechnic Institute ) ..... 467

6.1 Introduction ..... 468

6.2 Transformation Invariant Attributes for  
Digitized Object Outlines ..... 469

6.3 Design of an Inference Engine for an  
Image Interpretation Expert System ..... 472

6.4 Current plans ..... 480

Attachment A: Design of an Inference Engine for an Image  
Interpretation Expert System - IPL-TR-067 ..... 485

Attachment B: Transformation Invariant Attributes for  
Digital Object Outlines ..... 597

7 SPEECH UNDERSTANDING RESEARCH  
( Rochester Institute of Technology ) ..... 701

7.1 Introduction ..... 702

7.2 Research Activities ..... 703

7.3 Speech Scientist's Workbench ..... 708

7.4 Tasks for the Coming Year..... 710

Bibliography ..... 712

8 TIME-ORIENTED PROBLEM SOLVING  
( The University of Rochester ) ..... 715

8.1 Introduction ..... 716

8.2 Description of Research Accomplished ..... 716

8.3 Future Research ..... 717

Appendix 8-A: A Common Sense Theory of Time ..... 719

Appendix 8-B: A Model of Naive Temporal Reasoning ..... 731

Appendix 8-C: Toward a Theory of Plan Recognition ..... 749

Appendix 8-D: A Formal Logic that Supports Planning with a Partial  
Description of the Future ..... 817

9	COMPUTER ARCHITECTURES FOR VERY LARGE KNOWLEDGE BASES ( Syracuse University ) .....	835
9.1	Introduction .....	836
9.2	Partial Match Retrieval .....	838
9.3	Surrogate Files .....	840
9.4	Computer Architecture for partial Match Retrievals .....	845
9.5	Summary .....	847
9.6	Future Plans .....	850
	References .....	852

and

	KNOWLEDGE BASE MAINTENANCE USING LOGIC PROGRAMMING METHODOLOGIES ( Syracuse University ) .....	853
9.7	Background and Goals .....	855
9.8	Related concerns .....	857
9.9	Future plans .....	858
	Appendix 9-A: Meta-Level Programming and Knowledge Representation .	859
	Appendix 8-B: Fast Decompilation of Compiled Prolog Clauses .....	893
	Appendix 8-C: The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler.. .....	901
	Appendix 8-D: Compiler Optimizations for the WAM .....	915
	Appendix 8-E: A Meta-Level Extension of Prolog .....	923

## 6 AUTOMATIC PHOTO INTERPRETATION

Report submitted by:

J. W. Modestino

G. Nagy

Electrical, Computer and Systems Engineering Department

Rensselaer Polytechnic Institute

Troy, NY 12180-3590

### TABLE OF CONTENTS

I. Introduction .....	468
II. Transformation Invariant Attributes for Digitized Object Outlines .....	469
References for Section II .....	471
III. Design of an Inference Engine for an Image Interpretation Expert System .....	472
References for Section III .....	476
IV. Current Plans .....	480
Attachment 6-A: Design of an Inference Engine for an Image Interpretation Expert System - IPL-TR-067 .....	485
Attachment 6-B: Transformation Invariant Attributes for Digital Object Outlines .....	597

## I. INTRODUCTION:

The research objective of this project is to develop a system to assist a photointerpreter to understand, interpret, and report the contents of a (digitized) photograph more rapidly and more consistently, and to reduce the degree of personnel expertise necessary.

The research reported here was begun during the 1984-85 academic year under the direction of Professor Herbert Freeman. The two completed subprojects are summarized below and the complete reports are included in the attachments. Professors J. Modestino and G. Nagy took over in the Fall 1985 and developed the detailed plans reported in Section IV.

The ancillary objective of the grant is to increase the number of qualified personnel in the general area of artificial intelligence. The three major ways in which we seek to accomplish this, as described in Volume I of the Annual Report, are:

1. Improve and coordinate the AI-related courses at RPI and develop a comprehensive graduate-level curriculum in Knowledge Engineering.
2. Extend faculty opportunities for AI-related research.
3. Improve communications among AI researchers, including graduate students. We have already held a number of meetings bringing together faculty who were previously unaware of common interests. We are now institutionalizing the interaction and are extending it to nearby organizations outside the RPI umbrella.

This report is organized as follows: Section II is a summary of a transform-based invariant feature extraction method for objects in digital images. Section III is a summary of the development of a simple inference engine for image interpretation. Section IV is an outline of the work currently underway, which will form the bulk of our activities in 1986.

## II. TRANSFORMATION INVARIANT ATTRIBUTES FOR DIGITIZED OBJECT OUTLINES.

Two methods for automatic recognition of objects based on their closed chain-code boundary description were investigated. Granlund [6] and Zahn and Roskies [13] both derived expressions for Fourier descriptors of closed boundary objects which are invariant under scale, rotation and translation. Persoon and Fu [11] reported good results using Fourier descriptors to perform character recognition and machine parts recognition. Wallace and Wintz [12] obtained good results using Fourier descriptors to recognize three dimensional views of aircraft boundaries. A different method for computing a scale-, rotation-, and translation-invariant description of an object boundary is given by Hu [7]. This method is based on invariant moments. Alt [1962] derives moments invariant under an affine transformation [3] (i.e., they are invariant to translation, scale, and "stretching" and "squeezing" along the horizontal axis, but are not invariant under rotation). Dudani, et al. [4] reported good results in automatic identification of aircraft using the invariant moments of Hu. Other approaches are described in [9], [12].

To test the invariance of both the Fourier descriptors and moment invariants, the capability to both rotate and scale a digitized object was programmed. To facilitate this process, several small command processing language programs were written.

The invariant attributes for 6 different scales and orientations of two objects' outlines were compared. The Fourier descriptors seem to be more sensitive to scale and orientation than do the invariant moments. The higher-order moments were also inconsistent.

A fuzzy isodata clustering analysis [2] of the invariant attributes for 20 aircraft outlines was performed (in LISP) on the Fourier and moment descriptors. The full 16 member attributes vector gave the best clustering of the test patterns. Neither the area, perimeter or Fourier descriptor attributes are capable of detecting the "delta" wing aircraft typified by the F-16XL [8], whereas the moments do seem able to group them together. The details are provided in [10].

A fuzzy clustering analysis of 4 basic shapes (rectangles, squares, triangles, and circles) was also performed. With this method, it was hoped to enable statements such as "this closed boundary is 0.3 like a rectangle

and 0.6 like a triangle", which falls very naturally from the fuzzy clustering approach. Unfortunately, the method was not even able to distinguish squares from circles. It was determined that the problem is with the original invariant moments and Fourier data. This data is not similar for similarly shaped objects.

The invariant moments and Fourier descriptors for several "blob" shapes were also analyzed. A successively larger amount of noise was added to the outlines of 2 different blob shapes. The analysis shows that the invariant moments are, on average, more susceptible to noise than are the Fourier descriptors. In addition, slight systematic variations in the outlines of the 2 original blobs were made to see how the invariant moments and Fourier descriptors reacted to them. The Fourier descriptors were again less sensitive to the systematic noise.

In summary, both the invariant moments and Fourier descriptors seem to be invariant under scale and rotation changes for simple objects such as the rectangle. For more complex shapes, the first 3 invariant moments seem to be more stable than the Fourier descriptors. The best clustering of the aircraft is obtained using all 16 of the attributes. Clustering of the basic shapes did not work due to the inconsistency of the Fourier descriptor and invariant moment data. The blob analysis showed that the Fourier descriptors used here are less sensitive to both random and systematic noise.

The fuzzy clustering analysis of the attribute data is a good way to make decisions about an object's shape without having to consider the actual raw data values themselves. This is particularly true with an expert system where one wishes to write rules which are easy to interpret and maintain. Neither the invariant moments nor the Fourier descriptors used here seem to be the ideal raw data values on which to base the decision about shape. Along with such measures as area and perimeter they do provide some help as shown by the fuzzy clustering of the aircraft outlines presented in Attachment B.

## References for Section II

1. F. L. Alt, "Digital Pattern Recognition by Moments", Journal of the ACM, Vol. 9, pp. 240-258 (1962).
2. J. C. Bezdek, Pattern Recognition with Fuzzy Objective Function Algorithms, Plenum Press, New York (1981).
3. P. S. Bhaskar, "Design of an Inference Engine for an Image Interpretation Expert System, RPI Image Processing Laboratory Technical Report IPL-TR-067 (1985).
4. S.A. Dudani, K. J. Breeding, and R. B. McGhee, "Aircraft Identification by Moment Invariants", IEEE Transactions on Computers, Vol. C-26, No. 1, pp.39-46 (1977).
5. J. D. Foley and A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Company, Reading, Massachusetts (1982).
6. G. H. Granlund, "Fourier Preprocessing for Hand Print Character Recognition", IEEE Transactions on Computers, Vol. C-21, pp. 195-201 (1972).
7. M. K. Hu, "Visual Pattern Recognition by Moment Invariants", IRE Transactions on Information Theory, Vol. IT-8, pp. 179-187 (1962).
8. Jane's All The World's Aircraft. Jane's Publishing Company Limited, London, England (1983).
9. F. Merkel and T. Lorch, "Hybrid Optical-Digital Pattern Recognition", Applied Optics, Vol. 23, NO. 10, pp. 1509-1516 (1984).
10. B. G. Nickerson, "Fuzzy Isodata Clustering of Aircraft Outline Features", Final project report for course 36.6714 Fuzzy Sets and Expert Systems, Rensselaer Polytechnic Institute, Troy, NY (1984).
11. E. Persoon and K. S. Fu, "Shape Discrimination Using Fourier Descriptors", IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-7, NO. 3, pp. 170-179 (1977).
12. T. P. Wallace and P. A. Wintz, "An Efficient Three-Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors", Computer Graphics and Image Processing 13, pp. 99-126 (1980).
13. C. T. Zahn and R. Z. Roskies, "Fourier Descriptors for Plane Closed Curves", IEEE Transactions on Computers, Vol. C-21, No. 3, pp. 269-281 (1972).



### III. DESIGN OF AN INFERENCE ENGINE FOR AN IMAGE INTERPRETATION EXPERT SYSTEM

A combination of backward-chaining and forward-chaining strategy has been implemented. Two kinds of rules are introduced: IF rules are used for backward-chaining and WHEN rules are used for forward-chaining. Primitive certainty factors are associated with each fact. The truth value of false is indicated by -1, true by +1 and 0 indicates not known.

To introduce more flexibility into the environment, provision has been made to attach VERBS and PREDICATES to antecedent and consequent clauses of each rule. The VERBS are used to take some action on the consequent clauses and PREDICATES control the manner in which the antecedent clauses are satisfied.

The VERBs implemented are:

1. WRITE: to insert a fact into the list of facts.
2. CLEAR: to delete a fact from the facts list.
3. DISPLAY: to show a fact with all its attributes to the user.
4. ASK-Y: to ask the user a question and assign a true certainty factor, if the reply is yes.

The PREDICATEs implemented are:

1. IS-TRUE: to check whether the certainty factor associated with a fact is +1,
2. IS-FALSE: to check whether the certainty factor associated with a fact is -1, and
3. ASK-Y: to ask the user whether the fact is true.

A rudimentary explanation of the conclusion has been incorporated. This involves listing and displaying all the rules used to derive a conclusion. This provides some glimpse of a train of thought. A record is also kept of the usage of each rule. This may be used to assist some meta-rules in the future.

A BEHAVIOR switch is provided with the backward-chaining interpreter, which can be toggled by the user. This causes the system to alternate from behavior B1 to behavior B2. Behavior B1 is the default, where if a needed fact is not known, the system will first try to prove it and if that is not feasible, it will ask the user. B1 has the advantage of minimizing the amount of questions asked to the user, but it also treats the

user at the lowest level of input data. In the automatic image interpretation expert system, this could directly be translated into a request for specific data from the image processor. Behavior B2 also checks whether the needed fact is known, but if that is not the case, it will ask the user directly. If the user does not know the answer, then the system will try to infer it using the rules. B2 involves the user at higher levels of difficulty, but it also asks too many questions and it may not be useful for an automatic system. However, it serves as a debugging aid for the rulebase.

The syntax adopted for the rules is given below:

```
<RULE>--> ( <RULE-ID> <CODE> )
<RULE-ID> --> unique-id
<CODE> --> <RULE-TYPE> <PREMISES> <CONSEQUENTS>
<RULE-TYPE> --> (IF/ (WHEN))
<PREMISES> --> <CONDITION> <PLEMISES-REST>
<PREMISES-REST> --> ( <CONDITION> <PREMISE-REST> )
<CONDITION> --> ( <PREDICATE. fact> )
<PREDICATE> --> IS-TRUE / IS-FALSE / ASK-Y
<CONSEQUENTS> --> ( <ACTION> <CONSEQUENTS-REST> )
<CONSEQUENTS-REST> --> ( <ACTION> <CONSEQUENTS-REST> / )
<ACTION> --> (( <VERB> fact) <CF> )
<VERB> --> WRITE / CLEAR / DISPLAY / ASK-Y
<CF> --> +1 / -1 / 0
```

The general form expected for a fact is an object-attribute-value tuple. However, this is not very rigid and facts can also be represented in other ways. A label indexes facts in the fact base. The object-attribute-value tuple, the certainty of the fact and an indicator showing the origin of the fact (the image processor, the user or the inference engine itself) for explanation purposes are tagged as the property lists of the label.

The LISP functions that carry out the above-mentioned functions are given in Attachment A.

By introducing greater flexibility in certainty factors with a dependence parameter D, it is possible to take into account the differing statistical dependencies amongst the evidences. But it may be difficult for the expert quantitatively to assess the statistical dependencies, as such assessment is no longer intuitive. It is easy to implement though but its adequacy to take care of both the necessity and sufficiency conditions may

require versions of the same rule but with different dependence parameters and certainty factors.

The subjective Bayesian method is the easiest to implement. However, it requires expertise to assign values to the sufficiency and necessity factors in a way that reflects the true association. This is complicated by the fact that multiple evidences may point to the same hypothesis. If new premises have to be added to an existing rule, the sufficiency and necessity factors may need to be appropriately modified. It was found that judging the relevance of different evidences to the conclusion requires a considerable amount of trial-and-error attempts. One distinct advantage of this method is that it is impervious to the order in which the probabilities of the evidences change. Besides, there are the assumptions that the hypotheses should be mutually exclusive and exhaustive, and all evidences should be conditionally independent under each hypothesis.

The Dempster's rule formulation is commutative and associative and thus the order in which inferences are drawn is not critical. The probability range  $(0, 0)$  corresponds to no knowledge at all and will result from any attempt to apply an inapplicable rule. Even if such a rule is applied, it has no effect on the eventual conclusions. Also,  $(a, 0) + (c, 0) = (a + c - ac, 0)$ . The probability ranges  $(a, 0)$  and  $(c, 0)$  indicate no disbelief in the corresponding rules; in this case, the probabilities combine in the usual fashion. It is possible to use this for dynamically changing evidences because the inverse of the combination can be applied. This enables us to retract the conclusion of an earlier inference without influencing conclusions drawn by other means. However, to reduce computational time complexity, the evidences are required to be independent and the hypotheses mutually exclusive. Also the normalization process may lead to incorrect results.

In conclusion, the uncertainty associated with some types of evidence or facts is complex and it is unlikely that a single, uniform representation will ever be sufficient to model it. The necessity and possibility theory, proposed by Zadeh [33], extends the Dempster-Shafer concept to handle the case when the evidence is a fuzzy set. In this approach, the normalization is not required and thus may prove valuable. It may be worthwhile to try it as an alternative for this expert system.

As a comparison between backward-chaining and forward-chaining strategies, it has not yet been ascertained which approach is more suitable for this project. If there is no division of rules, a forward-chaining approach brings out the dominant features in the evidence, whereas a backward-chaining approach may be better suited to answer the user's specific queries. A combination of both approaches is implemented, but a more clear-cut strategy may be desirable. It may perhaps be forward-chaining in the preliminary stages, thresholding the probability estimates, and then backward-chaining on a separate set of rules more pertinent to the user's interest. If the rules incorporate variables, the matching procedure between the facts and either the consequent or the antecedent clauses need to have a unification algorithm.

References for Section III

1. Barnett, J.A., "Computational Methods for a Mathematical Theory of Evidence", Proceedings of the Seventh International Conference on Artificial Intelligence, Vancouver, pp. 868-875, 1981.
2. Barstow, D.R., "An Experiment in Knowledge-based Automatic Programming", Artificial Intelligence, Vol. 12, pp. 7-119, 1979.
3. Ben-Bassat, Moshe, "Membership and Multiperspective Classification: Introduction, Applications, and a Bayesian Model", IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-10, No. 6, pp.331-336, 1980.
4. Bonissone, Piero, P., and Richard M. Tong, "Reasoning with Uncertainty in Expert Systems", International Journal of Man-Machine Studies, 1984.
5. Clancey, William J., "The Epistemology of a rule-based expert system: A Framework for Explanation", Artificial Intelligence, Vol. 20(3), pp. 215-251, 1983.
6. Clancey, William J., "Classification Problem Solving", Proc. of the National Conference on Artificial Intelligence, AAAI-84, pp. 49-55, 1984.
7. Davis, R., "Interactive Transfer of Expertise: Acquisition of New Inference Rules", Artificial Intelligence, Vol. 12, pp. 121-157, 1979.
8. Dempster, A.P., "A Generalization of Bayesian Inference", Journal of the Royal Statistical Society, Vol. 30, Series B, pp. 205-247, 1968.
9. Duda, Richard O. and J. G. Gaschnig, "Knowledge-Based Expert Systems Come of Age", Byte, pp. 238-279, 1981.
10. Duda, Richard O. and Edward H. Shortliffe, "Expert Systems Research", Science, Vol. 220, No. 4594, pp. 261-268, 1983.

11. Duda, Richard O., P. E. Hart and Nils J. Nilsson, "Subjective Bayesian Methods for Rule-based Inference Systems", Proceedings of National Computer Conference, pp. 1075-1082, 1976.
12. Forgy, C.L., "The OPS5 User's Manual", Tech Report, Carnegie-Mellon University, 1980.
13. Gevarter, William B., "Expert Systems: Limited but Powerful", IEEE Spectrum, Vol. 71, No. 8, pp. 39-45, 1982.
14. Ginsburg, Matthew L., "Non-monotonic Reasoning using Dempster's Rule", Proceedings of the National Conference on Artificial Intelligence, AAAI\_84, pp. 126-129, 1984.
15. Hayes-Roth, F., D. Waterman and D. Lenat, (eds.), Building Expert Systems, Addison-Wesley, New York, 1983.
16. Makeswari, S. N. and S. K. Hakimi, "On Models for Diagnoseable Systems and Probabilistic Diagnosis", IEEE Trans. on Computers, Vol. C-25, pp. 228-236, 1976.
17. Martin-Clauaire, Roger and Henri Prade, "On the Problems of Representation and Propagation of Uncertainty", Second NAFIP Workshop, Schenectady, NY, June 29-July 1, 1983.
18. Minsky, Marvin, "A Framework for Representing Knowledge", The Psychology of Computer Vision, P. H. Winston, ed., McGraw-Hill, New York, pp. 211-277, 1975.
19. Mitchie, D., ed., Expert Systems in the Microelectronic Age, Edinburgh University Press, Edinburgh, 1979.
20. Nau, Dana S., "Expert Computer Systems", IEEE Computer, pp. 63-85, January 1983.

21. Nilsson, Nils, J., Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, New York, 1971.
22. Nilsson, Nils, J., Principles of Artificial Intelligence. Tioga, Palo Alto, CA., 1980.
23. Pednault, E.P.D., S. W. Zucker and L. V. Muresan, "On the independent Assumption underlying Subjective Bayesian Updating", Artificial Intelligence, Vol. 16(2), pp. 2134-222, 1981.
24. Raiffa, H., Decision Analysis, Addison-Wesley, New York, 1968.
25. Rauch, Herbert E., "Probability Concepts for an Expert System used for data fusion", The AI Magazine, pp. 55-60, Fall 1984.
26. Schubert, L. K., "Extending the Expressive Power of Semantic Nets", Artificial Intelligence, Vol. 7, No.2, pp. 163-198, 1976.
27. Schafer, Glenn A., A Mathematical Theory of Evidence, Princeton University Press, New Jersey, 1976.
28. Shortcliffe, E. H., Computer-based Medical Consultations: MYCIN. Elsevier/North Holland, NY, 1976.
29. Shortcliffe, E. H. and B. G. Buchanan, "A Model of Inexact Reasoning in Medicine", Mathematical Biosciences, Vol. 23, pp. 351-379, 1975.
30. Strat, Thomas M., "Continuous Belief Functions for Evidential Reasoning", Proceedings of the National Conference on Artificial Intelligence, AAAI-84, pp. 308-313, 1984.
31. Waterman, D.A. and F. Hayes-Roth, ed., Pattern-Directed Inference Systems, New York, Academic Press, 1978.
32. Winston, P.H. and B.K.P. Horn, Lisp, Addison-Wesley, NY, 1981.

33. Zadeh, L.A., "The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems", Fuzzy Sets and Systems, North Holland, Vol. 11, pp. 199-227, 1983.



#### IV. CURRENT PLANS:

To reach our primary objective of automated photointerpretation, we have decided that it would be most effective to establish a number of intermediate goals which can be pursued simultaneously. Broadly speaking, these sub-goals can be divided into research on fundamental problems which must be solved to achieve success in automating photointerpretation tasks (tasks 1 - 4 below), and the development and adaptation of mathematical and software tools to build a demonstration system (tasks 5 and 6). At this time, the various tasks are deliberately formulated independently of each other in order to allow separate groups to make progress without interdependence and to draw on the current skills of the participants even as new skills are acquired. These diverse endeavors will be gradually integrated by the principal investigators to demonstrate both significant research contributions and a prototype photointerpretation system.

The major research tasks are the following:

1. Probabilistic model of images based on neighborhood-induced random fields. Random fields are the two-dimensional equivalents of Markov chains. Their potential usefulness in digital image registration and filtering arises from the fact that they allow a precise mathematical characterization, with a relatively small number of parameters, of the objects under consideration. This allows the application of formal techniques to analyze the performance of competing classes of algorithms. Our objectives here are to extract the relevant image parameters from real images and to compare the predicted performance of image-processing algorithms on the model with the experimentally-observed performance on the images. Various sets of model parameters would be part of the knowledge base of an expert system and would allow the selection of the most appropriate algorithm according to the statistics of the images under consideration.
2. Model for topographic terrain features. Peaks, ridges and valleys are considered significant terrain features, but most current

methods for extracting such features suffer from myopia rooted in the differential calculus and they are unable to distinguish significant landmarks from local artifacts. In order to overcome this problem, we propose a model based on visibility. We assume that the horizontal scale is much larger than the vertical scale (i.e., the horizontal extent and separation of features is much greater than their vertical extent), that the terrain is relatively uniformly sampled, and that in a given region all of the significant features are of the same order of magnitude, but that large variations occur between regions. For each point the boundary of the surface area that is visible from it is determined. Then a minimal set of observation points is selected such that the entire area under consideration is visible from these points. In addition to the applicability of the method to the extraction of features for special-purpose sketch-maps, the method should be directly applicable to the selection of observation posts, the location of hiding places, the placement of line-of-sight transmitters and receivers, and to orientation/navigation. Here again, the ultimate objective is to include a feature-oriented abstraction of the area under consideration in the knowledge base of the expert system.

3. Target detection and classification based upon multiple sensor inputs. In many image exploitation applications, including photointerpretation, there is some advantage in using multiple-sensor views of the same scene to aid in target detection and/or classification. Any scheme for accomplishing this must take into account the different characteristics of the various sensors as well as their operational status and viewing conditions. The object of this research task will be to formulate and characterize AI techniques for target detection and/or classification from multiple imaging sensors. Particular attention will be given to the case of multiple targets against cluttered backgrounds. Furthermore, we intend to investigate the situation where image frame sequences are available from each sensor and target motion is possible between frames.

4. Symbolic Image Processing. A large amount of research has been accomplished in recent years dealing with what might be called numerical image processing. Here, individual pixels are represented by real-(or integer-) valued numbers and the images are operated upon by signal processing techniques suitably adapted to two (and sometimes higher) dimensions. Typical numerical signal processing techniques include: image enhancement and restoration, edge detection and contour extraction, texture classification and discrimination, etc. More recently, a body of techniques is being developed which might properly be called symbolic image processing in distinction to the now classical numerical image processing techniques. In this case the image is no longer represented at the pixel level but at a higher level in terms of symbolic constructs. The symbolic representation generally requires less storage and/or transmission and is typically in a form more useful to subsequent image exploitation tasks. However, the distinction between these two types of image processing is somewhat cloudy since often numerical techniques are used to obtain the symbolic representation of an image. The purpose of this task is to investigate the use of AI techniques in symbolic image processing within the context of image exploitation applications. More specifically, we intend to explore symbolic representation and processing techniques useful in expert systems for image exploitation.

The principal development tasks are:

5. Relational database system for images. Database systems for images have traditionally been customized, "home-grown" systems with restricted application to selected image formats. Current database technology offers, however, significant advantages for photointerpretation, including high-level data modeling, file integration, individual user views, query optimization, independence of physical and logical data organization, information hiding, multiple-user interactive access, and many design and application tools. In this task, we will be attempting to demonstrate the feasibility of common image-processing operations in the access language (QUEL) of a

general-purpose database system (Ingres). Further, we allow the user to view images as continuous entities in both extent and intensity. In essence, we are developing for image-processing a concept analogous to floating-point arithmetic. Current work centers on automatic query translation and optimization using a problem-specific knowledge base. The development of a high-level photointerpretation-oriented application language and of low-level image store organization for a general-purpose relational DBMS is expected to pay dividends both in accelerated research and construction of the demonstration system.

6. Constrained image-segmentation and labeling techniques. In many images, only certain segmentation patterns are admissible, and results may be improved by restricting the search to these patterns. Horizontal and vertical segmentation boundaries predominate, for instance, in modern city-scapes, factory floors, technical document layouts, and integrated circuit design. In aerial photographs, the constraints are more complex. Initially we are concentrating on environments where the boundaries of the objects of interest may be found by the consecutive application of a sequence of one-dimensional filters and detectors directed by an expert system with gradually-improving knowledge of the segmentation rules. In the second phase of the project, another expert system, with more detailed knowledge about the relative positions of the different types of objects, directs the labeling process. Eventually, we expect to combine the two phases allowing image segmentation to be corrected by feedback from the labeling process. The labeling process may be envisioned as a tree-search, with heuristics based on the layout rules. The initial demonstration system will run in an interactive mode, with a human PI-expert monitoring, questioning, and correcting the labeling subsystem. Among questions currently under study are the choice of the development system (we are strongly prejudiced towards commercially-available expert systems) and the precise formalism for introducing PI expertise in the knowledge base.

Attachment A

Design of an Inference Engine  
for an Image Interpretation  
Expert System

IPL-TR-067

February 1985

by

P. S. Bhaskar

## ABSTRACT

This report describes the development of an inference engine for an image interpretation expert system. The inference engine has the capability to prove or disprove hypotheses by chaining backwards through a sequence of rules or to arrive at a conclusion from a set of observed facts. Three methods for estimating the validity of the conclusion drawn from inexact data and uncertain rules are described. The design of the inference engine was specifically directed toward possible use in an image interpretation expert system.

## ACKNOWLEDGEMENT

I wish to express my sincere gratitude to Professor H. Freeman for giving me the opportunity to work in the area of expert systems. The guidance and encouragement received under his supervision are gratefully acknowledged.

I also wish to thank the members of the AIIIP group for the many technical discussions, which have been of great benefit.

The research reported here was supported by the Rome Air Development Center's Post Doctoral Program, administered through Syracuse University, under Contract WM10751-4-12000, RADC Task I-4-4836.

## 1. INTRODUCTION

One of the recent achievements in the field of artificial intelligence is the development of expert systems. An expert system is a computer system that attains high levels of performance in areas requiring special education and training or in specialized, professional domains. The study of expert systems is concerned with methods and techniques for constructing man-machine systems with specialized problem-solving expertise. Expertise consists of knowledge about a particular domain, understanding of domain problems and skill at solving some of these problems. Expert systems lay special emphasis on the knowledge that underlies human expertise in a particular domain and not on domain-independent problem-solving or formal reasoning methods. The knowledge-based expert systems are thus a class of computer programs that use a collection of facts, "rules of thumb" of the human experts and other knowledge about a limited field to help make inferences within this field. Amassing and managing a large amount of knowledge rather than sophisticated reasoning techniques is responsible for most of the power of the system.

Expert systems generally have several distinguishing features, which include symbolic representation, symbolic inference and heuristic search. They are similar to systems developed in other branches of artificial



intelligence. The knowledge-based system approach is different from conventional programming systems as the goals here have no algorithmic solution and inferences have to be made on uncertain or incomplete information. The advantages are that an expert system can be designed to supply one or more hypotheses, request additional information and explain the reasoning process. It also allows modification of the knowledge used without extensive reprogramming and use of the knowledge for other purposes, like education. This is due to the fact that the model of problem-solving in the application domain is explicitly in view as a separate entity or knowledge base rather than appearing only implicitly as part of the coding of the program. In a conventional computer program, knowledge pertinent to the problem and methods for using this knowledge are intertwined; thus it is difficult to extract and modify the domain knowledge.

Ordinary computer programs organize knowledge on two levels: data and program. Expert systems, however, organize knowledge on three levels, data, knowledge base and control. On the data level we have declarative knowledge about the particular problem being solved and the current state toward the solution of the problem. On the knowledge-base level we have knowledge specific to the particular kind of problem that the system is set up to solve. The control structure makes decisions about how to use the specific problem-solving knowledge. Thus in an

expert system, there is a clear separation between general knowledge about the problem domain from information about the current problem and methods for applying the general knowledge to the problem.

The knowledge base is manipulated by a separate, clearly identifiable control strategy. This mechanism is analogous to the deductive reasoning of a human expert. The known facts and rules are used to infer conclusions. This component of the expert system, which can be domain-independent, is called the inference engine.

The objective of the research described here is to build an inference engine suitable for an expert system for image interpretation. The purpose of this expert system will be automatically to analyze aerial or satellite images to determine their "meaning." The system thus will attempt to perform the function of a human photointerpreter. One constituent of this expert system will be the image processor, which performs the task of extracting basic information from the input image. This includes a shape discriminator. The output of the image processor will reside in the fact base. The knowledge base will be gleaned from human experts. The inference engine will use the fact and knowledge bases probabilistically to determine the objects in an aerial image.

One well-known way to represent declarative knowledge is by means of formulas in first-order predicate calculus. Simple declarative facts can be represented as instantiated

predicates. Another way of representing declarative knowledge is in terms of frames. Frames are data structures in which all knowledge about a particular object or event is stored together. Such a representation cannot represent any more concepts than first-order predicate logic, but the organization of knowledge can be useful for modularity and accessibility of the knowledge. In addition, frame systems allow ways to specify default values for pieces of information about an object when that object is not explicitly available. Semantic networks are a third way to represent declarative knowledge. The objects are represented by nodes in a graph and the relations among them are represented by labelled arcs. However, the most popular approach for representing the domain knowledge needed for an expert system is by production rules. Rule-based systems work by applying rules, noting the results and applying new rules based on the changed situation. Rule-based systems are particularly attractive when much of the expert knowledge in the field comes from empirical associations acquired as a result of experience. When more causal information is available, the former methods may be more pertinent, as in [20]. The rule-based approach has been adopted to represent knowledge in the image interpretation expert system.

## 2. CONTROL STRATEGY

A production rule is of the form IF (A) AND (B) THEN (C), e.g., IF (region\_class is 'water') AND (regions\_surrounding are 'land') THEN (region is lake). This situation-action rule, as it is also called, captures the semi-logical response of the human expert, that is, if a certain kind of situation arises, a certain action can be taken or a certain conclusion can be inferred. Clauses that represent the situation are called antecedent clauses and those representing the action are called consequent clauses. Furthermore, the rules may be interconnected, that is, the consequent clauses of a rule may form part of the antecedent clauses of some other rule. A set of rules thus form a "chunk" of knowledge in a particular field.

The first task is to input the rules for a particular domain in a specialized language and produce an internal representation. Then a general reasoning mechanism is provided. The inference engine or rule interpreter has to decide in what order the rules can be applied and in what manner the rules should be enabled, that is, it must establish the matching process between the evidence collected and the antecedent clauses of the rules. This determines the control strategy.

The simplest strategy is to scan through the rule list until one is found such that the antecedent clauses match the facts present. Then this rule is applied, updating the

set of known facts, and scanning is resumed. This process continues until a goal state is reached or no rules can be applied to extend the facts list. This method is called forward-chaining. Since the behavior of the system is directly responsive to the set of known facts, this is a data-driven control strategy or data-directed inference.

A different strategy is to select a goal to be achieved and then scan the rules for one whose consequent matches the goal. If such a rule is found, a match is tried between the antecedent of the rule and the existing facts. If such a match is possible, then the problem is solved, otherwise the antecedent clauses are treated as problems to be solved and the same process is applied recursively. This process stops when all the problems generated are solved or if there are no further rules to establish the sub-goal. This method is called backward-chaining. It is also called the goal-driven control strategy or consequent reasoning and is similar to means-end analysis.

The two strategies serve different functions. The data-driven or forward-chaining approach has the disadvantage of generating many hypotheses not directly related to the problem under consideration. This may lead to wasterul erforts as well as runaway problems. However the goal-driven, backward-chaining approach has the disadvantage of becoming fixed on an initial set of hypotheses and having difficulty shifting focus when the

data available do not support them. If the need is to identify objects and their meaning in a given aerial image, the forward-chaining approach may be most appropriate; whereas if the purpose is to determine whether certain specific objects exist in an aerial image, the backward-chaining strategy is suitable.

To alleviate the problem, a combination of forward-chaining and backward-chaining strategies has been implemented in the inference mechanism here. The primary control lies in the backward-chaining method. While evaluating the validity of a hypothesis, if new facts are generated by the image processor supplementing the fact base, the forward-chainer gets activated. This ensures adequate control.

Provision is made for two kinds of rules in the system. "IF" and "WHEN" rules are to be used in backward-chaining and forward-chaining modes, respectively. Thus a choice exists as to in which specific way a fact is to be used in the forward or backward direction. The rulebase has to be judiciously constructed to eliminate redundancies as well as to reduce search time.

### 3. DEALING WITH UNCERTAINTY

Various kinds of uncertainty are typically encountered in expert systems. The presence of uncertain information can be attributed to at least four causes. The first type is related to the reliability of information. Uncertainty can be present in the factual knowledge (i.e. the set of assertions or facts) as a result of noisy input data, imprecise feature extraction or due to inaccuracy and poor reliability of the instruments used to make the observations. Uncertainty can also occur in the knowledge base as a result of weak implications. This occurs when the expert is unable to establish a strong correlation between the premise and the conclusion. The expert also must artificially express the degree of implication as a scalar value on an interval.

The second type of uncertainty is caused by the inherent imprecision of the rule representation language. If rules are not expressed in a formal language, their meaning cannot be interpreted exactly. A 'lexical' match does not adequately compare subsets of facts with the premise; a semantic match is required to compare the approximate meaning of facts and premises. In classical logic, modus ponens allows  $\{Y \text{ is } B\}$  to be derived from the assertion of the statements:  $\{X \text{ is } A\}$  and  $\{(X \text{ is } A) \rightarrow (Y \text{ is } B)\}$ . However, the inference can be made only if the unconditional assertion  $\{X \text{ is } A\}$  is identical to the

premise of the conditional assertion. Therefore, to cover all possible situations, we need as many rules as the number of different values that X can take.

The third type of uncertainty occurs when inference is based on incomplete information. In this case, we need partially to match facts and premise, i.e. we have to allow for a value of "unknown" during the evaluation. Furthermore, we need to be able to distinguish between necessary evidence and possible (optional) evidence and be able to treat them appropriately in the partial matching process.

The fourth type of uncertainty arises from the aggregation of rules from different knowledge sources or different experts. There are four possible errors that can occur in knowledge represented as production rules: conflicting, redundant, subsuming, and missing rules. Conflicting rules, that succeed under the same circumstances but make contradictory conclusions, increase the level of uncertainty by creating inconsistencies. Redundant rules, that under the same circumstances make the same conclusion, may create an over-inflated assessment of the certainty of the conclusion. A subsumed rule, in which the premise of the first rule, is a subset of the second, can create an over-estimate of the certainty of the common conclusion. Missing rules, that fail to provide a needed conclusion under the right circumstances, create uncertainty of the third type, in which inference is based



on incomplete information. This fourth kind of uncertainty can be reduced by compilation of the rule set into a network and analyzing the network.

In the literature surveyed, many techniques were found to deal with uncertainty in expert systems. A good guide to the various techniques is provided in [3]. Three of the methods, using certainty factors, by the subjective Bayesian method and by the Dempster-Shafer method have been implemented.

#### A. Certainty Factors

The first method, of using certainty factors, is an extension to the approach used by MYCIN, which is based on confirmation theory, [28]. Here, we have a certainty factor assigned to each rule. This is basically a measure of the expert's belief that the rule is valid. The facts also have an associated certainty factor which varies from -1 to +1; -1 indicating that the fact is known to be false, 0 indicating that the fact is unknown and +1 indicating that the fact is known to be true. The certainty factor is the difference between the degree of belief and the degree of disbelief for a given hypothesis after supporting evidence is found.

Thus all the certainty factors of the premises of a rule are found and the minimum certainty factor is chosen. The certainty factor of the conclusion is obtained by multiplying the minimum certainty factor with the strength

of the rule. Now there can be other ways to prove the same hypothesis by using other rules. For each path taken, we calculate the certainty factor and treat this as an OR condition, thus we must take the maximum certainty factor. While an AND/OR tree of the rules to prove a hypothesis is traversed bottom-up, the certainty factors can be aggregated.

As shown in [25], this method of calculating probabilities is based on the assumption that the facts are under maximum statistical dependence. In general, if the evidences are statistically independent, the probability of (A AND B) is given by  $P_a * P_b$  and the probability of (A OR B) is given by  $(P_a + P_b - P_a * P_b)$ . In the case where the statistical dependence between the items of evidence A & B is minimum (i.e. A has maximum dependence with NOT B), the probability of (A AND B) is given by  $\text{MAX}(P_a + P_b - 1, 0)$  and the probability of (A OR B) is given by  $\text{MIN}(P_a + P_b, 1)$ . We have to make some provision to take into account the statistical dependencies between evidences.

But evidences are not going to be related exactly in this manner, being under maximum or minimum dependence or being statistically independent. So, we introduce a dependence parameter D, which varies from -1 to +1. When D equals 0, the two items of evidence are independent; when D equals 1, the two items have maximum dependence; when D equals -1, the two items have minimum dependence. Suppose the probability calculated under the assumption that the

items are independent is  $C_1$ , under maximum dependence is  $C_2$  and under minimum dependence is  $C_3$ , the actual resulting probability is the appropriate linear combination,

$$P = (D * C_2 + (1 - D) * C_1) \text{ for } 0 \leq D \leq 1,$$

$$P = (|D| * C_3 + (1 - |D|) * C_1) \text{ for } -1 \leq D < 0.$$

This makes the three previous categories of dependence special cases, or in other words, provides a smooth transition from one to the other.

If  $D$  lies between  $-1$  and  $0$ , it actually yields the necessary condition. When  $D$  is  $1$ , the value obtained for AND is the greatest and for OR is the lowest. When  $D$  is  $-1$ , the value obtained for AND is the lowest and for OR the greatest.

Another aspect which should be considered is where the expert should assign a particular value of the dependence parameter  $D$ , whether it should be assigned with each rule, i.e. at each node of an AND-OR tree or for the entire tree structure. If the statistical dependencies between evidences vary depending on the rules, each rule should have a value of  $D$  specified for it.

This has been implemented with a backward chainer. In this case, the parameter  $D$  can be specified for the entire rulebase. A sample session, the rulebase used with certainty factors, and a program listing are given in Appendix A.

## B. Subjective Bayesian Method

The Subjective Bayesian method is described in [11] and was used in PROSPECTOR. This approach uses an effective likelihood ratio to quantify the strength of a given rule. Each antecedent clause or evidence E is assumed to be independent, or conditionally independent of the other antecedent clauses for the same consequent or hypothesis H. A rule is of the form  $E \rightarrow H$ . The prior probabilities of the evidence and hypothesis are set to some initial value. After some evidence is gathered, the probability associated with evidence E changes. This in turn should change the probability of the hypothesis H.

In addition, two factors, the sufficiency factor LS and the necessity factor LN, are associated with each rule. The sufficiency factor measures the sufficiency of a piece of evidence to prove a given hypothesis and is mathematically given by  $LS = P(E/H) / P(E/HBAR)$  where  $P(E/H)$  is the probability of the evidence E being true given that hypothesis H is true and  $P(E/HBAR)$  is the probability of the evidence E being true given that hypothesis H is not true. A high sufficiency factor indicates that if E is true, the probability of H is greatly increased, whereas a low sufficiency factor indicates that even if the probability of E is high (i.e. E is close to true), it serves to increase the probability of H only marginally. The necessity factor, on the other hand, quantifies the necessity of a piece of evidence to prove the negation of the hypothesis. It determines the

change in the probability of H, if the probability of the evidence reduces (or in the extreme, if E is found to be false). It is defined as  $LN = P(\overline{E}/H) / P(\overline{E}/\overline{H})$  where  $P(\overline{E}/H)$  is the probability of the evidence E being false given that hypothesis H is true and  $P(\overline{E}/\overline{H})$  is the probability of the evidence E being false given that hypothesis H is false. If the necessity factor is large, a decrease in the probability of E has little effect on the probability of H. If the necessity factor is low or close to zero, the fact that E is false serves to reduce the probability of H drastically.

Let  $P_p(E)$  be the prior probability of the evidence,  $P_p(H)$  be the prior probability of the hypothesis, LS be the sufficiency factor, LN be the necessity factor,  $P(E)$  be the probability of the evidence (which is the input),  $P(H)$  be the probability of the hypothesis, and BM be the Bayesian Multiplier. Then,  $P(H) = P_p(H) * BM$ , where BM is obtained from linear interpolation between the three points  $(0, LN)$ ,  $(P_p(E), 1)$  and  $(1, LS)$ , as follows:

For  $P(E) < P_p(E)$ ,

BM is given by  $P(E) * (1 - LN) / P_p(E) + LN$ .

For  $P(E) > P_p(E)$ ,

BM is given by  $[P(E) - P_p(E)] * (LS - 1) / (1 - P_p(E)) + 1$ .

Thus the sufficiency and the necessity factors control the piecewise linear interpolation of the change of the probability of the hypothesis H between the evidence

acquiring a truth value of 'false' and 'true' from the initial probability.

The following relations should always be true:

$LS > 1 \rightarrow LN < 1$  ,

$LS < 1 \rightarrow LN > 1$  ,

$LS = 1 \rightarrow LN = 1$  .

One of the advantages of this method as mentioned in [23] is that if no evidence is obtained, the probability of the hypothesis remains the same and the order in which evidence is obtained and rules are applied does not affect the final probability. This advantage can best be utilized by a forward-chaining system, rather than a backward-chaining system. In the implementation of this method, whenever the probability of any evidence changes (assuming the evidences correspond to some sensors or the points at which data is gathered), it should trigger the forward-chaining system and update the probability of the hypothesis.

This method has been implemented with a set of four LISP functions. A sample session, the rulebase used with the necessity and sufficiency factors, and a program listing are given in Appendix B.

### C. Dempster-Shafer Method

The Belier Function proposed by Shafer [27], was developed within the framework of Dempster's work on upper and lower probabilities induced by a multivalued mapping as

in [8]. In this context, the lower probability was associated with a degree of belief and the upper probability with a degree of plausibility. A computational problem has been encountered as the evaluation of the degree of belief and plausibility increases exponentially with the number of hypotheses. However, by introducing assumptions about the type of evidence to be combined, the computational time complexity can be reduced from exponential to linear, as shown by Barnett [1]. The requirements for the evidences to be conditionally independent and the hypotheses to be mutually exclusive still remain.

Shafer defines certainty to be a function that maps subsets in a space on a scale from zero to one, where the total certainty over the space is one. This definition allows one to assign a non-zero certainty to the entire space as an indication of ignorance. This provision for indicating ignorance is one way in which this theory differs from conventional probability theory and is a significant advantage, since in this application and in many others, the available knowledge is incomplete and involves a large degree of uncertainty.

In this method, instead of considering a rule  $E \rightarrow H$  with a certain probability, we assign a probability range to the rule, e.g.,

If (region\_class is sand)

then (region is desert) (0.9 0.05). 0.9 indicates the

extent to which a hypothesis is confirmed by the evidence and 0.95 (1 - 0.05) is the extent to which it is disconfirmed, i.e. this rule states that we believe a region is desert if its class is sand in a range from 0.9 to 0.95. It may be easier for an expert to assign a range rather than a specific probability.

Next, as shown in [14], each fact accordingly has a probability range (a,b) and the probability range of the consequent is obtained by taking the product of the range of the fact (a, b) with the range of the rule (c,d) resulting in (ca, bd). Now, if a chain of the rule is formed as X --> Y with (a, b) and Y --> Z with (c, d) the range of X --> Z is (ac, ad).

The same fact can be obtained by different paths of the tree resulting in different ranges (this is analogous to the OR condition in the AND-OR tree). Here, we combine the ranges of the same fact using Dempster's Rule of Combination,

$$(a, b) + (c, d) = (1 - ax * cx / (1 - (a * d + b * c)),$$

$$1 - bx * dx / (1 - (a * d + b * c)) )$$

where  $ax = 1 - a$ ,  $bx = 1 - b$ ,  $cx = 1 - c$ ,  $dx = 1 - d$ .

This provides a mechanism to combine the certainty of facts which can be concordant or contradictory. When this is used, it seeks consensus among all the hypotheses supported by the user's observations. This approach is attractive since problems of conflicting observations,



knowledge updating from multiple experts and ruling-out are resolved automatically by the rule of combination. The conventional approaches to knowledge processing do not have this advantage.

The processing procedure has five steps:

1. An observation of evidence and the certainty associated with the observation are entered into the system. They define a certainty function over the input space.
2. Multiple independent observations are allowed. The certainty functions defined by these observations are combined by using the rule of combination.
3. A rule or mapping is activated when the evidence receives a non-zero belief in the combined observation.
4. The certainty of the evidence is multiplied by the certainty of the rule, resulting in a certainty function defined over the output space for each rule applied.
5. By means of the rule of combination, all the activated certainty functions in the output space are combined into a single certainty function. From this certainty function, the total belief is computed.

This method has been implemented on a backward chainer. A sample session, rulebase used and program listing are given in Appendix C.

#### 4. DESIGN OF THE INFERENCE ENGINE

A combination of backward-chaining and forward-chaining strategy has been implemented. Two kinds of rules are introduced, IF rules are used for backward-chaining and WHEN rules are used for forward-chaining. Primitive certainty factors are associated with each fact. The truth value of false is indicated by -1, true by +1 and 0 indicates not known.

To introduce more flexibility into the environment, provision has been made to attach VERBs and PREDICATEs to antecedent and consequent clauses of each rule. The VERBs are used to take some action on the consequent clauses and PREDICATEs control the manner in which the antecedent clauses are satisfied.

The VERBs implemented are :

1. WRITE: to insert a fact into the list of facts,
2. CLEAR: to delete a fact from the facts list,
3. DISPLAY: to show a fact with all its attributes to the user, and
4. ASK\_Y: to ask the user a question and assign a true certainty factor, if the reply is yes.

The PREDICATEs implemented are :

1. IS\_TRUE: to check whether the certainty factor associated with a fact is +1,
2. IS\_FALSE: to check whether the certainty factor associated with a fact is -1, and

3. ASK\_Y: to ask the user whether the fact is true.

A rudimentary explanation of the conclusion has been incorporated. This involves listing and displaying all the rules used to derive a conclusion. This provides some glimpse of a train of thought. A record is also kept of the usage of each rule. This may be used to assist some meta-rules in the future.

A BEHAVIOR switch is provided with the backward-chaining interpreter, which can be toggled by the user. This causes the system to alternate from behavior B1 to behavior B2. Behavior B1 is the default, where if a needed fact is not known, the system will first try to prove it and if, that is not feasible, it will ask the user. B1 has the advantage of minimizing the amount of questions asked to the user, but it also treats the user at the lowest level of input data. In the automatic image interpretation expert system, this could directly be translated into a request for specific data from the image processor. Behavior B2 also checks whether the needed fact is known, but if that is not the case, it will ask the user directly. If the user does not know the answer, then the system will try to infer it using the rules. B2 involves the user at higher levels of difficulty, but it also asks too many questions and it may not be useful for an automatic system. However, it serves as a debugging aid for the rulebase.

The syntax adopted for the rules is given below:

```

<RULE> --> ( <RULE-ID> <CODE> )
<RULE-ID> --> unique-id
<CODE> --> <RULE-TYPE> <PREMISES> <CONSEQUENTS>
<RULE-TYPE> --> ( IF / ( WHEN
<PREMISES> --> <CONDITION> <PREMISES-REST>
<PREMISES-REST> --> ( <CONDITION> <PREMISES-REST> / )
<CONDITION> --> ( <PREDICATE> (fact) )
<PREDICATE> --> IS_TRUE / IS_FALSE / ASK_Y
<CONSEQUENTS> --> <ACTION> <CONSEQUENTS-REST>
<CONSEQUENTS-REST> --> <ACTION> <CONSEQUENTS-REST> / )
<ACTION> --> ( <VERB> (fact) <CF> )
<VERB> --> WRITE / CLEAR / DISPLAY / ASK_Y
<CF> --> +1 / -1 / 0

```

The general form expected for a fact is an object-attribute-value tuple. However, this is not very rigid and facts can also be represented in other ways. A label indexes facts in the fact base. The object-attribute-value tuple, the certainty of the fact and an indicator showing the origin of the fact (the image processor, the user or the inference engine itself) for explanation purposes are tagged as the property lists of the label.

The LISP functions that carry out the above-mentioned functions are given in Appendix D.

## 5. ILLUSTRATION OF USE OF THE INFERENCE ENGINE

Two examples are given below to show the capabilities of the inference engine. For each case, the rulebase used and a sample terminal session are given. An attempt is made to prove or disprove different hypotheses. The probabilistic estimate of the hypothesis is obtained by using certainty factors as in [28].

The first example relates a hypothetical, simple-minded scenario to distinguish objects or regions from an aerial image. The rulebase conforms to the syntax previously given. Here, the format adopted for a fact is not the usual object-attribute-value tuple. The object is 'region' in all cases. The format of the facts is used primarily to enhance readability. Verbs are attached with each consequent clause and Predicates with each antecedent clause. The strength of each rule is quantitatively assessed. Each rule has a unique label. The rules are listed below:

```
(defun setup ()  
(setq rules '(  
  (ruleif B1  
    (if (is_true (region_class is artificial))  
        (is_true (region_shape is regular)))  
    (then (write (region_class is man_made) 0.96)))  
  (ruleif B2  
    (if (is_true (region_class is steel))  
        (is_true (region_shape is round)))  
    (then (write (region is oiltank) 1)))
```

```

(ruleif B3
  (if (is_true (region_class is tar))
      (is_true (region_shape is long_and_narrow))
      (is_true (region_shape is regular)))
  (then (write (region is road) 0.9)))

(ruleif B4
  (if (is_true (region_shape is rectangular))
      (is_true (regions_adjacent_on_two_sides is road))
      (is_true (regions_adjacent_on_two_sides have class=water))
      (is_true (region_class is man_made)))
  (then (write (region is bridge) 0.91)))

(ruleif B5
  (if (is_true (region_class is concrete))
      (is_true (region_shape is regular)))
  (then (write (region is building) 0.8)))

(ruleif B6
  (if (is_true (region_class is metal))
      (is_true (region_shape is roughly_rectangular))
      (is_true (regions_surrounding have class=water)))
  (then (write (region is ship) 0.87)))

(ruleif B7
  (if (is_true (region_class is man_made))
      (is_true (most_regions_surrounding have class=water)))
  (then (write (region is offshore_oilrig) 0.84)))

(ruleif B8
  (if (is_true (region_class is man_made))
      (is_true (region_shape is long_and_narrow))
      (is_true (region_adjacent has class=water))
      (is_true (region_adjacent has class=land)))
  (then (write (region is dock) 0.91)))

(ruleif B9
  (if (is_true (region_class is metal))
      (is_true (region_shape is airplane_like)))
  (then (write (region is airplane) 0.7)))

(ruleif B10
  (if (is_true (region_class is water))
      (is_true (region_shape is regular))
      (is_true (region_area is small))
      (is_true (regions_surrounding have class=artificial)))
  (then (write (region is artificial) 0.8)))

```

```

        (then (write (region is swimming_pool) 0.83)))

(ruleif B11
  (if (is_true (region_class is water))
      (is_false (region_shape is regular))
      (is_true (regions_surrounding have class=land
)))
    (then (write (region is lake) 0.86)))

(ruleif B12
  (if (is_true (region_class is land))
      (is_false (region_shape is regular))
      (is_true (regions_surrounding have class=wate
r)))
    (then (write (region is island) 0.94)))

(ruleif B13
  (if (is_true (region_class is water))
      (is_true (region_shape is long_and_narrow)))
    (then (write (region is river) 0.75)))

(ruleif B14
  (if (is_true (region_class is sand))
      (is_true (region_adjacent has class=land))
      (is_true (region_adjacent has class=water))
      (is_true (region_width is small)))
    (then (write (region is beach) 0.89)))

(ruleif B15
  (if (is_true (region_class is marshy))
      (is_true (region_adjacent has class=land))
      (is_true (region_adjacent has class=water)))
    (then (write (region is marsh) 0.92)))

(ruleif B16
  (if (is_true (region_class is greenery))
      (is_true (region_shape is regular)))
    (then (write (region is field) 0.89)))

(ruleif B17
  (if (is_true (region_class is sand))
      (is_false (region_shape is regular)))
    (then (write (region is desert) 0.95)))

(ruleif B18
  (if (is_true (region_class is ice))
      (is_false (region_shape is regular)))
    (then (write (region is glacier) 0.84)))

(ruleif B19
  (if (is_true (region_class is water))
      (is_true (regions_surrounding have class=dese

```

```

rt)))
  (then (write (region is oasis) 0.95)))

(ruleif B20
  (if (is_true (region is road))
      (is_true (region_adjacent is airplane)))
  (then (write (region is runway) 0.97)))

(rulewhen F1
  (when (is_true (region_class is steel)))
  (then (write (region_class is artificial) 0.97)))

(rulewhen F2
  (when (is_true (region_class is tar)))
  (then (write (region_class is artificial) 0.95)))

(rulewhen F3
  (when (is_true (region_class is concrete)))
  (then (write (region_class is artificial) 0.96)))

(rulewhen F4
  (when (is_true (region_class is sand)))
  (then (write (region_class is land) 0.97)))

(rulewhen F5
  (when (is_true (region_class is greenery)))
  (then (write (region_class is land) 0.92)))

(rulewhen F6
  (when (is_true (region_shape is round)))
  (then (write (region is rectangular) 1.0)))

(rulewhen F7
  (when (is_true (region_shape is rectangular)))
  (then (write (region is rectangular) 1.0)))
))

```

```

(setq hypotheses '(
  (region is oiltank)
  (region is road)
  (region is bridge)
  (region is building)
  (region is ship)
  (region is offshore_oilrig)
  (region is dock)
  (region is airplane)
  (region is swimming_pool)
  (region is runway)
  (region is island)
  (region is lake)

```



```
(region is river)
(region is beach)
(region is marsh)
(region is field)
(region is desert)
(region is glacier)
(region is oasis) )))
```

The rules above are invoked to prove the hypotheses sequentially in the order given as shown below. The facts are obtained either from files or from the user. The factbase is also displayed for verification.

```
OK, lisp tryout
WARNING This software is for evaluation only and contains a
time out mechanism
usethenforward
testif
testwhen
usethen
tryruleif
tryrulewhen
stepforward
deduce
testif+
tryruleif+
verify
inthen
prescreen
thenp
diagnose
showrulesused
tracerules
why
ifp
inif
usedp
how
attach
startlist
buildlist
write
clear
getc
getflag
```

```

is_true
is_false
list_facts
startriplist
buildriplist
writerip
clearrip
getcfrrip
is_truerip
display
ask_y
jump
()
()
()
0
()
> (co kbase)
kbase
setup
> (setup)
()
> (co facts_oiltank)
facts_oiltank
( ( ( region_class is steel) 0.95) ( ( region_shape is rou
nd) 0.80))
> (diagnose)
( region is oiltank) is proved with certainty 0.87
( region is oiltank)
> (co facts_oasis)
facts_oasis
( ( ( region_class is water) 0.9) ( ( region_shape is
irregular) 0.95) ( ( regions_surrounding have class=desert)
0.92))
> (diagnose)
Enter certainty of fact : ( regions_surrounding have clas
s=water)
0.2
Enter certainty of fact : ( region shape rectangular)
0.3
Enter certainty of fact : ( region class artificial)
no
( region is oasis) is proved with certainty 0.63
( region is oasis)
> (co facts_dock)
facts_dock
( ( ( region_class is artificial) 0.8) ( ( region_shape is
regular) 0.85)
( ( region_shape is long_and_narrow) 0.7) ( ( region_adjac
ent
has class=water) 0.9) ( ( region_adjacent has class=land) 0
.75))

```

```

> (diagnose)
Enter certainty of fact : (most_surrounding_regions have
class=water)
0.3
Enter certainty of fact : ( region_shape is irregular)
no
Enter certainty of fact : ( region_class is water)
no
( region is dock) is proved with certainty 0.56
( region is dock)
>(co facts_ship)
( ( ( region_class is artificial) 0.9) ( ( region_shape is
rectangular)
0.95) ( ( regions_surrounding have class=water) 0.87))
> (diagnose)
Enter certainty of fact : ( region shape irregular)
no
Enter certainty of fact : ( region class land)
yes
Enter certainty of fact : ( region shape round)
no
( region is ship) is proved with certainty 0.696
( region is ship)
> (co facts_river)
facts_river
( ( ( region_class is water) 0.9) ( ( region_shape is
long_and_narrow) 0.87))
> (diagnose)
( region is river) is proved with certainty 0.696
( region is river)
> (co facts_island)
facts_island
( ( ( region_class is water) 0.93) ( ( region_shape is
irregular) 0.95) ( ( regions_surrounding have class=land) 0
.89))
> (diagnose)
Enter certainty of fact : ( region_shape is long_and_narr
ow)
0.1
Enter certainty of fact : ( region_class is water)
no
( region is island) is proved with certainty 0.744
( region is island)

```

The next example illustrates how this inference engine may be used to identify regions from an actual aerial image. The image processor extracts edges and heuristically closes region boundaries. Figure 1 displays

the regions thus obtained. This contains six closed region boundaries.

The features of the closed regions, such as area, perimeter, centroid, orientation, Fourier descriptors and invariant moments, are calculated by the feature extraction program and stored in a file. The feature data is translated into a suitable format for the fact base by means of another program. A certainty measure is introduced at this stage to determine how closely it matches a fact. This is now suitable for symbolic manipulation only. The rulebase used is listed below:

```
(defun setup ()
  (setq rules '(
    (ruleif ACT1
      (if (is_true (region_area is medium))
          (is_true (region_perimeter is medium))
          (is_true (region is roughly_rectangular)))
      (then (write (region is wing_of_airplane) 0.65
    )))

    (ruleif ACT2
      (if (is_true (region_area is very_small))
          (is_true (region_perimeter is small))
          (is_true (region_shape is bulbous)))
      (then (write (region is engine_of_airplane) 0.
    7)))

    (ruleif ACT3
      (if (is_true (region_area is large))
          (is_true (region_perimeter is long))
          (is_true (region_shape is long_and_narrow
    )))
      (is_true (region is regular)))
      (then (write (region is fuel_storage_tank) 0.
    9)))

    (ruleif ACT4
      (if (is_true (region is regular))
          (is_false (region_area is small))
          (is_true (region_shape is rectangular)))
```

```

        (then (write (region is building) 0.75))) )
)

(setq hypotheses '(
  (region is wing_of_airplane)
  (region is engine_of_airplane)
  (region is fuel_storage_tank)
  (region is building))) )

```

The inference engine then tries to find the most probable hypothesis to fit each region. The terminal sessions show the results obtained below.

For Region 1,

```

> (co facts_1)
facts_1
( ( ( region_shape is rectangular) 0.7) ( (region_area is
small) -0.7) ( ( region is regular) 0.5))
> (diagnose)
( region is building) is proved with certainty 0.4
( region is building)

```

For Region 2,

```

> (co facts_2)
facts_2
( ( ( region_area is large) 0.8) ( ( region_perimeter is
large) 0.7) ( ( region_shape is long_and_narrow) 0.6) ( ( r
egion is uniform) 0.63))
> (diagnose)
( region is fuel_storage_tank) is proved with certainty 0
.54
( region is fuel_storage_tank)

```

For Region 3,

```

> (co facts_3)
facts_3
( ( ( region_area is very_small) 0.89) ( ( region_perimet
er is small) 0.6)
( ( region_shape is bulbous) 0.7))
> (diagnose)
( region is engine_of_airplane) is proved with certainty
0.5
( region is engine_of_airplane)

```

For Region 4,

```
> (co facts_4)
facts_4
(( ( region_area is medium) 0.4) ( ( region_perimeter is
small) 0.8)
( ( region_shape is triangular) 0.75) ( ( region is unifo
rm) 0.65))
> (diagnose)
No hypothesis can be confirmed.
```

For Region 5,

```
> (co facts_5)
facts_5
(( ( region_area is very_small) 0.6) ( ( region_perimete
r is small) 0.5) ( ( region is uniform) 0.72) ( ( region_sh
ape is bulbous) 0.4))
> (diagnose)
( region is engine_of_airplane) is proved with certainty
0.28
( region is engine_of_airplane)
```

For Region 6,

```
> (co facts_6)
facts_6
(( ( region_area is medium) 0.75) ( ( region_perimeter i
s medium) 0.8) ( ( region_shape is roughly_rectangular) 0.3
6))
> (diagnose)
( region is wing_of_airplane) is proved with certainty 0.
37
( region is wing_of_airplane)
```

Thus, we see that the inference engine can be used to determine conclusions provided the knowledge base is set up appropriately. The power of the entire expert system is derived from the knowledge it possesses and not from the particular formalisms and inference schemes it employs.



**Fig. 1**      **Sample Image after Extraction**  
**of Closed Region Boundaries**

## 6. CONCLUSIONS

By introducing greater flexibility in certainty factors with the dependence parameter  $D$ , it is possible to take into account the differing statistical dependencies amongst the evidences. But it may be difficult for the expert quantitatively to assess the statistical dependencies, as such assessment is no longer intuitive. It is easy to implement though. Adequately to take care of both the necessity and sufficiency conditions may require versions of the same rule but with different dependence parameters and certainty factors.

The subjective Bayesian method is the easiest to implement. However, it requires expertise to assign values to the sufficiency and necessity factors in a way that reflects the true association. This is complicated by the fact that multiple evidences may point to the same hypothesis. If new premises have to be added to an existing rule, the sufficiency and necessity factors may need to be appropriately modified. It was found that judging the relevance of different evidences to the conclusion requires a considerable amount of trial-and-error attempts. One distinct advantage of this method is that it is impervious to the order in which the probabilities of the evidences change. Besides, there are the assumptions that the hypotheses should be mutually exclusive and exhaustive, and all evidences should be



conditionally independent under each hypothesis.

The Dempster's rule formulation is commutative and associative and thus the order in which inferences are drawn is not critical. The probability range  $(0, 0)$  corresponds to no knowledge at all and will result from any attempt to apply an inapplicable rule. Even if such a rule is applied, it has no effect on the eventual conclusions. Also,  $(a, 0) + (c, 0) = (a + c - ac, 0)$ . The probability ranges  $(a, 0)$  and  $(c, 0)$  indicate no disbelief in the corresponding rules; in this case, the probabilities combine in the usual fashion. It is possible to use this for dynamically changing evidences because the inverse of the combination can be applied. This enables us to retract the conclusion of an earlier inference without influencing conclusions drawn by other means. However, to reduce computational time complexity, the evidences are required to be independent and the hypotheses mutually exclusive. Also the normalization process may lead to incorrect results.

In conclusion, the uncertainty associated with some types of evidence or facts is complex and it is unlikely that a single, uniform representation will ever be sufficient to model it. The necessity and possibility theory, proposed by Zadeh [33], extends the Dempster-Shafer concept to handle the case when the evidence is a fuzzy set. In this approach, the normalization is not required and thus may prove valuable. It may be worthwhile to try

it as an alternative for this expert system.

As a comparison between backward-chaining and forward-chaining strategies, it has not yet been ascertained which approach is more suitable for this project. If there is no division of rules, a forward-chaining approach brings out the dominant features in the evidence, whereas a backward-chaining approach may be better suited to answer the user's specific queries. A combination of both approaches is implemented, but a more clear-cut strategy may be desirable. It may perhaps be forward-chaining in the preliminary stages, thresholding the probability estimates, and then backward-chaining on a separate set of rules more pertinent to the user's interest. If the rules incorporate variables, the matching procedure between the facts and either the consequent or the antecedent clauses need to have a unification algorithm.

## BIBLIOGRAPHY

1. Barnett, J.A., "Computational Methods for a Mathematical Theory of Evidence", Proceedings of the Seventh International Conference on Artificial Intelligence, Vancouver, pp. 868-875, 1981.

2. Barstow, D.R., "An Experiment in Knowledge-based Automatic Programming," Artificial Intelligence, Vol. 12, pp. 7-119, 1979.

3. Ben-Bassat, Moshe, "Membership and Multiperspective Classification: Introduction, Applications, and a Bayesian Model," IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-10, No. 6, pp.331-336, 1980.

4. Bonissone, Piero P., and Richard M. Tong, "Reasoning with Uncertainty in Expert Systems," International Journal of Man-Machine Studies, 1984.

5. Clancey, William J., "The Epistemology of a rule-based expert system: A Framework for Explanation," Artificial Intelligence, Vol. 20(3), pp. 215-251, 1983.

6. Clancey, William J., "Classification Problem Solving," Proc. of the National Conference on Artificial Intelligence, AAAI-84, pp. 49-55, 1984.

7. Davis, R., "Interactive Transfer of Expertise: Acquisition of New Inference Rules," Artificial Intelligence, Vol. 12, pp. 121-157, 1979.

8. Dempster, A.P., "A Generalization of Bayesian

Inference," Journal of the Royal Statistical Society, Vol. 30, Series B, pp. 205-247, 1968.

9. Duda, Richard O. and J. G. Gaschnig, "Knowledge-Based Expert Systems Come of Age," Byte, pp. 238-279, 1981.

10. Duda, Richard O. and Edward H. Shortliffe, "Expert Systems Research," Science, Vol. 220, No. 4594, pp. 261-268, 1983.

11. Duda, Richard O., Peter E. Hart and Nils J. Nilsson, "Subjective Bayesian Methods for Rule-based Inference Systems," Proceedings of National Computer Conference, pp. 1075-1082, 1976.

12. Forgy, C.L., "The OPS5 User's Manual," Tech. Report, Carnegie-Mellon University, 1980.

13. Gevarter, William B., "Expert Systems: Limited but Powerful," IEEE Spectrum, Vol. 71, No. 8, pp. 39-45, 1983.

14. Ginsburg, Matthew L., "Non-monotonic Reasoning using Dempster's Rule," Proceedings of the National Conference on Artificial Intelligence, AAAI-84, pp. 126-129, 1984.

15. Hayes-Roth, F., D. Waterman and D. Lenat, (eds.), Building Expert Systems, Addison-Wesley, New York, 1983.

16. Makeswari, S.N. and S.L. Hakimi, "On Models for Diagnoseable Systems and Probabilistic Diagnosis," IEEE Trans. on Computers, Vol. C-25, pp. 228-236, 1976.

17. Martin-Clouaire, Roger and Henri Prade, "On the Problems of Representation and Propagation of Uncertainty," Second NAFIP Workshop, Schenectady, N.Y., June 29-July 1, 1983.

18. Minsky, Marvin, "A Framework for Representing Knowledge," The Psychology of Computer Vision, P.H. Winston, ed., McGraw-Hill, New York, pp. 211-277, 1975.

19. Mitchie, D., ed., Expert Systems in the Microelectronic Age, Edinburgh University Press, Edinburgh, 1979.

20. Nau, Dana S., "Expert Computer Systems," IEEE Computer, pp. 63-85, January 1983.

21. Nilsson, Nils J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.

22. Nilsson, Nils J., Principles of Artificial Intelligence, Tioga, Palo Alto, Calif., 1980.

23. Pednault, E.P.D., S.W.Zucker and L.V.Muresan, "On the independent Assumption underlying Subjective Bayesian Updating," Artificial Intelligence, Vol. 16(2), pp. 213-222, 1981.

24. Raiffa, H., Decision Analysis, Addison-Wesley, New York, 1968.

25. Rauch, Herbert E., "Probability Concepts for an Expert System used for data fusion," The AI Magazine, pp. 55-60, Fall 1984.

26. Schubert, L.K., "Extending the Expressive Power of Semantic Nets," Artificial Intelligence, Vol. 7, No.

2, pp. 163-198, 1976.

27. Shater, Glenn A., A Mathematical Theory of Evidence, Princeton University Press, New Jersey, 1976.

28. Shortliffe, E.H., Computer-based Medical Consultations: MYCIN, Elsevier/North Holland, New York, 1976.

29. Shortliffe, E.H. and B. G. Buchanan, "A Model of Inexact Reasoning in Medicine," Mathematical Biosciences, Vol. 23, pp. 351-379, 1975.

30. Strat, Thomas M., "Continuous Belief Functions for Evidential Reasoning," Proceedings of the National Conference on Artificial Intelligence, AAAI-84, pp. 308-313, 1984.

31. Waterman, D.A. and F. Hayes-Roth, ed., Pattern-Directed Inference Systems, New York, Academic Press, 1978.

32. Winston, P.H. and B.K.P. Horn, Lisp, Addison-Wesley, New York, 1981.

33. Zadeh, L.A., "The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems," Fuzzy Sets and Systems, North Holland, Vol. 11, pp. 199-227, 1983.

APPENDIX A  
IMPLEMENTATION OF CERTAINTY FACTORS

A sample session that demonstrates the use of certainty factors to determine the validity of the hypothesis is given. The rulebase with which this was executed is included. The Lisp functions that are used to traverse the AND/OR tree of rules and aggregate the certainty factors are also listed.

A.1 Certainty Factors: Sample session

OK, lisp main

WARNING This software is for evaluation only and contains a time out mechanism

```
ge
insert
recall
inthen
thenp
finamax
finaproa
askuser
apifs
rtestif
prove
diagnose
> (co setup)
setup
setup
> (setup)
()
> (co facts_oiltank)
facts_oiltank
(( ( region class steel) 0.92) ( ( region shape round) 0.82))
> (diagnose)
( region is oiltank) is proved with certainty 0.82
( region is oiltank)
> (setq d '0.4)
0.4
> (co facts_oiltank)
facts_oiltank
(( ( region class steel) 0.92) ( ( region shape round) 0.82))
> (diagnose)
( region is oiltank) is proved with certainty 0.9488512
( region is oiltank)
> (setq d '0.7)
0.7
> (co facts_road)
facts_road
(( ( region class tar) 0.9) ( ( region shape long_and_narrow) 0.85)
( ( region shape regular) 0.75))
> (diagnose)
Enter certainty of fact : ( region shape round)
no
Enter certainty of fact : ( region class steel)
no
( region is road) is proved with certainty 0.676153035
( region is road)
```



```

> (co facts_bridge)
facts_bridge
( ( ( region class tar) 0.9) ( ( region adjacent_on_two_sides_to
region_is_road) 0.86) ( ( region adjacent_on_two_sides_to
region_with_structure_class_2) 0.92) ( ( region shape rectangular)
0.84))
> (diagnose)
Enter certainty of fact : ( region shape round)
no
Enter certainty of fact : ( region class steel)
no
Enter certainty of fact : ( region shape long_and_narrow)
yes
( region is road) is proved with certainty 0.8173629
( region is road)
> (co facts_ship)
facts_ship
( ( ( region class artificial) 0.9) ( ( region shape rectangular)
0.95) ( ( region surrounding_regions region_with_structure_class_2)
0.87))
> (diagnose)
Enter certainty of fact : ( region shape round)
no
Enter certainty of fact : ( region class steel)
no
Enter certainty of fact : ( region shape long_and_narrow)
yes
Enter certainty of fact : ( region class tar)
no
Enter certainty of fact : ( region adjacent_on_two_sides_to
region_with_structure_class_2)
yes
Enter certainty of fact : ( region adjacent_on_two_sides_to
region_is_road)
no
Enter certainty of fact : ( region class concrete)
no
Enter certainty of fact : ( region surrounding_regions region_with
structure_class_2)
yes
Enter certainty of fact : ( region shape irregular)
no
Enter certainty of fact : ( region class land)
no
Enter certainty of fact : ( region surrounding_regions

```

```

regions_with_structure_class_1)
no
Enter certainty of fact : ( region class water)
no
( region is ship) is proved with certainty 0.708488796
( region is ship)
> (co facts_beach)
facts_beach
( ( ( region class sand) 0.75) ( ( region width small) 0.9
8) ( (
region adjacent_to region_with_structure_class_1) 0.65) ( (
region
adjacent_to region_with_structure_class_2) 0.89))
> (diagnose)
Enter certainty of fact : ( region shape round)
no
Enter certainty of fact : ( region class steel)
no
Enter certainty of fact : ( region shape rectangular)
no
Enter certainty of fact : ( region adjacent_on_two_sides_
to
region_with_structure_class_2)
no
Enter certainty of fact : ( region adjacent_on_two_sides_
to
region_is_road)
no
Enter certainty of fact : ( region class tar)
no
Enter certainty of fact : ( region surrounding_regions re
gion _with
structure_class_2)
no
Enter certainty of fact : ( region shape irregular)
yes
Enter certainty of fact : ( region class land)
yes
Enter certainty of fact : ( region surrounding_regions
regions_with_structure_class_1)
no
Enter certainty of fact : ( region class water)
no
Enter certainty of fact : ( region shape long_and_narrow)
yes
Enter certainty of fact : ( region surrounding_regions
region_with_structure_class_2)
no
Enter certainty of fact : ( region most_surrounding_regio
ns
region_with_structure_class_2)
yes

```

```

Enter certainty of fact : ( region class marsh)
no
Enter certainty of fact : ( region class greenery)
no
( region is desert) is proved with certainty 0.70911
( region is desert)
> facts
( ( ( region is desert) 0.70911) ( ( region class greenery
) -1) ( (
region class marsh) -1) ( ( region is beach) 0.46295375) (
( region
belongs_to structure_class_12) 0.77625) ( ( region
most_surrounding_regions region_with_structure_class_2) 1)
( ( region
surrounding_regions region_with_structure_class_2) -1) ( (
region
shape long_and_narrow) 1) ( ( region class water) -1) ( ( r
egion
surrounding_regions regions_with_structure_class_1) -1) ( (
region
belongs_to structure_class_1) 1.09) ( ( region class land)
1) ( (
region shape irregular) 1) ( ( region surrounding_regions r
egion_with
structure_class_2) -1) ( ( region class tar) -1) ( ( regio
n
adjacent_on_two_sides_to region_is_road) -1) ( ( region
adjacent_on_two_sides_to region_with_structure_class_2) -1)
( ( region
shape rectangular) -1) ( ( region class steel) -1) ( ( reg
ion shape
round) -1) ( ( region class sand) 0.75) ( ( region width sm
all) 0.98)
( ( region adjacent_to region_with_structure_class_1) 0.65)
( ( region
adjacent_to region_with_structure_class_2) 0.89))
> alpha
0.5
> beta
0.8
> (diagnose)
( region is desert) is proved with certainty 0.70911
( region is desert)
> (setq facts '())
()
> (setq facts '())
()
> (diagnose)
Enter certainty of fact : ( region shape round)
no
Enter certainty of fact : ( region class steel)
no

```

```

Enter certainty of fact : ( region shape rectangular)
no
Enter certainty of fact : ( region adjacent_on_two_sides_
to
region_with_structure_class_2)
no
Enter certainty of fact : ( region adjacent_on_two_sides_
to
region_is_road)
yes
Enter certainty of fact : ( region class tar)
no
Enter certainty of fact : ( region surrounding_regions re
gion_with
structure_class_2)
no
Enter certainty of fact : ( region shape irregular)
yes
Enter certainty of fact : ( region class land)
yes
Enter certainty of fact : ( region surrounding_regions
regions_with_structure_class_1)
no
Enter certainty of fact : ( region class water)
no
Enter certainty of fact : ( region shape long_and_narrow)
no
Enter certainty of fact : ( region surrounding_regions
region_with_structure_class_2)
no
Enter certainty of fact : ( region most_surrounding_regio
ns
region_with_structure_class_2)
yes
Enter certainty of fact : ( region adjacent_to
region_with_structure_class_2)
yes
Enter certainty of fact : ( region adjacent_to
region_with_structure_class_1)
yes
Enter certainty of fact : ( region width small)
no
Enter certainty of fact : ( region class sand)
no
Enter certainty of fact : ( region class marsh)
no
Enter certainty of fact : ( region class greenery)
yes
( region is meadow) is proved with certainty 0.876168
( region is meadow)
> q
OK, como -e

```

## A.2 Certainty Factors: Rulebase

```
(defun setup ()
  (setq rules '(
    (r1:      ((region class land))
              ((region belongs_to structure_class_
1) 1.0)))
    (r2:      ((region class water))
              ((region belongs_to structure_class_
2) 1.0)))
    (r3:      ((region class artificial))
              ((region belongs_to structure_class_
3) 1.0)))
    (r4:      ((region class greenery))
              ((region belongs_to structure_class_
11) 0.8)))
    (r5:      ((region class sand))
              ((region belongs_to structure_class_
12) 0.9)))
    (r6:      ((region class marsh))
              ((region belongs_to structure_class_
13) 0.8)))
    (r7:      ((region class ice))
              ((region belongs_to structure_class_
14) 0.9)))
    (r8:      ((region class tar))
              ((region belongs_to structure_class_
31) 0.9)))
    (r9:      ((region class concrete))
              ((region belongs_to structure_class_
32) 0.9)))
    (r10:     ((region class steel))
              ((region belongs_to structure_class_
33) 0.9)))
    (r11:     ((region belongs_to structure_class_1
              (region shape regular))
              ((region is field) 0.8)))
```

```

1)      (r12:  ((region belongs_to structure_class_1
              (region shape irregular))
              ((region is meadow) 0.9)))

2)      (r13:  ((region belongs_to structure_class_1
              (region shape irregular))
              ((region is desert) 0.8)))

4)      (r14:  ((region belongs_to structure_class_1
              (region shape irregular))
              ((region is glacier) 0.7)))

2)      (r15:  ((region belongs_to structure_class_1
              (region adjacent_to region_with_stru
cture_class_1)
              (region adjacent_to region_with_stru
cture_class_2)
              (region width small))
              ((region is beach) 0.7)))

3)      (r16:  ((region belongs_to structure_class_1
              (region adjacent_to region_with_stru
cture_class_1)
              (region adjacent_to region_with_stru
cture_class_2))
              ((region is marsh) 0.8)))

)      (r17:  ((region belongs_to structure_class_1
              (region shape irregular)
              (region surrounding_regions
                region_with structure_class
-2))
              ((region is island) 0.8)))

)      (r18:  ((region belongs_to structure_class_2
              (region shape irregular)
              (region surrounding_regions
                regions_with_structure_class
-1))
              ((region is lake) 0.8)))

)      (r19:  ((region belongs_to structure_class_2
              (region shape long_and_narrow))

```

```

        (((region is river) 0.8)))
    (r20: ((region belongs_to structure_class_2
)
        (region shape irregular)
        (region surrounding_regions
            region_with_structure_class_
12))
        (((region is oasis) 0.7)))
    (r21: ((region belongs_to structure_class_3
)
        (region shape regular)
        (region surrounding_regions
            region_with_structure_class_
2))
        (((region is ship) 0.8)))
    (r22: ((region belongs_to structure_class_3
)
        (region shape regular)
        (region most_surrounding_regions
            region_with_structure_class_
2))
        (((region is offshore_oilrig) 0.7)))
    (r23: ((region belongs_to structure_class_3
)
        (region shape regular)
        (region shape long_and_narrow)
        (region adjacent_to
            region_with_structure_class_
1)
        (region adjacent_to
            region_with_structure_class_
2))
        (((region is dock) 0.8)))
    (r24: ((region belongs_to structure_class_3
1)
        (region shape long_and_narrow)
        (region shape regular))
        (((region is road) 0.9)))
    (r25: ((region belongs_to structure_class_3
3)
        (region shape round))
        (((region is oiltank) 1.0)))
    (r26: ((region belongs_to structure_class_3
2)
        (region shape regular))

```

```

((region is building) 0.9))
1) (r27: ((region belongs_to structure_class_3
        (region adjacent_on_two_sides_to
         region_is_road)
        (region adjacent_on_two_sides_to
         region_with_structure_class_
2)
        (region shape rectangular))
      ((region is bridge) 0.9))
3) (r28: ((region belongs_to structure_class_3
        (region shape airplane_like))
      ((region is airplane) 0.8))
(r29: ((region shape round))
      ((region shape regular) 1.0))
(r30: ((region shape rectangular))
      ((region shape regular) 1.0)) )

```

```
(setq hypotheses '(
```

```

((region is island)          0.0)
((region is ship)            0.0)
((region is ship)            0.0)
((region is oasis)           0.0)
((region is bridge)          0.0)
((region is building)        0.0)
((region is island)          0.0)
((region is lake)            0.0)
((region is river)           0.0)
((region is offshore_oilrig) 0.0)
((region is dock)            0.0)
((region is beach)           0.0)
((region is marsh)           0.0)
((region is field)           0.0)
((region is meadow)          0.0)
((region is desert)          0.0)
((region is glacier)         0.0)
((region is airplane)        0.0) ))

```

```

(setq alpha '0.5)
(setq beta  '0.8)
(setq d     '1.0)
(setq facts '()) )

```



### A.3 Certainty Factors: Lisp Functions

```
/*
/*          THE          BACKWARD          CHAINER
/*
/*          WITH CERTAINTY FACTORS AND DEPENDENCE PARAMETE
R
/*
```

```
/* GE returns t if a>b or a=b, otherwise returns nil.
```

```
(defun ge (a b)
  (prog ()
    (cond ((greaterp a b) (return t))
          ((equal a b) (return t))
          (t (return nil)))))
```

```
/* INSERT add a 'fact' to the factbase with a certainty f
actor cf.
```

```
(defun insert (fact cf)
  (setq facts (cons (list (car fact) cf) facts)))
```

```
/* RECALL checks whether a fact is present in the fact ba
se.
/*          If present, it returns the associated cf, other
wise
/*          it returns -2.0.
```

```
(defun recall (fact)
  (prog (rcfacts)
    (setq rcfacts facts)
    rclloop
    (cond ((null rcfacts) (return '-2.0))
          ((equal (car fact) (caar rcfacts))
           (return (cadar rcfacts))))
    (setq rcfacts (cdr rcfacts))
    (go rclloop) ))
```

```
/* INTHEN strings together and returns a list of rules,
/*          each of which can prove the fact.
```

```

(defun inthen (fact)
  (prog (itrules oprules)
    (setq itrules rules)
    (setq oprules '())
    itloop
    (cond ((null itrules) (return oprules))
          ((thenp fact (car itrules))
           (setq oprules (cons (car itrules) oprules)
                    s))))
    (setq itrules (cdr itrules))
    (go itloop)))

```

/\* THENP determines whether a fact is part of the RHS of a rule.

```

(defun thenp (fact rule)
  (prog (thens)
    (setq thens (caddr rule))
    thloop
    (cond ((null thens) (return nil))
          ((equal (car fact) (caar thens))
           (return t)))
    (setq thens (cdr thens))
    (go thloop) ))

```

/\* FINDMAX finds the strength of the rule for the given fact.  
 /\* prod aggregates the certainty factor of the premise  
 /\* (minm) with the strength of the rule. It returns the  
 /\* maximum absolute value between prod and maxm  
 /\* (which  
 /\* was the previous max\* value.

```

(defun findmax (fact rule minm maxm)
  (prog (thens cf aprod amaxm prod)
    (setq thens (caddr rule))
    fmloop1
    (cond ((null thens) (print 'Error_in_findmax))
          ((equal (car fact) (caar thens))
           (setq cf (cadar thens))
           (go fmloop2)))
    (setq thens (cdr thens))
    (go fmloop1)
    fmloop2
    (setq prod (* cf minm))

```

```

      (setq aprod (abs prod))
      (setq amaxm (abs maxm))
      (cond ((greaterp aprod amaxm) (return prod))
            (t (return maxm))))

(defun findprod (fact rule prob)
  (prog (thens cf)
    (setq thens (caddr rule))
    fploop
    (cond ((null thens) (print 'Error_in_findprod)
           ((equal (car fact) (caar thens))
            (setq cf (cadar thens))
            (return (* cf prob))))
          (setq thens (cdr thens))
          (go fploop)))

/*      ASKUSER obtains the certainty factor for a fact from
the user,
/*      does the required conversion from YES, UNKNO
WN, NO,
/*      call INSERT to add the fact to the fact base
and
/*      returns the certainty factor.

(defun askuser(fact)
  (prog (aucf)
    (printlist "Enter certainty of fact : " (car
fact) )
    askloop
    (setq aucf (read))
    (cond ((equal aucf 'yes) (setq aucf '1.0) (go
corloop))
          ((equal aucf 'unknown) (setq aucf '0.0)
          (go corloop))
          ((equal aucf 'no) (setq aucf '-1.0) (go
corloop))
          ((numberp aucf) (go corloop))
          (t (print "ERROR! Please enter again.")
            (go askloop)))
    corloop
    (insert fact aucf)
    (return aucf) ))

/*      APIFS adds a 0.0 certainty factor to the premise list
to make
/*      an appropriate format for comparison with the fa

```

ct base.

```
(defun apifs (ifs)
  (prog (inifs opifs)
        (setq inifs ifs)
        (setq opifs '())
        aloop
        (cond ((null inifs) (return opifs)))
        (setq opifs (cons (list (car inifs) '0.0) opifs)
              ))
        (setq inifs (cdr inifs))
        (go aloop)))
```

```
/* RTESTIF forms part of the recursive loop. It tries to
   prove each of the premises of the 'rule' by invoking
   PROVE. If any premise cannot be proven (cf < 0), it
   returns -2.0, otherwise it returns the lowest cf found
   amongst the premises.
```

```
(defun rtestif (rule)
  (prog (ifs min prsval mul)
        (setq min '2.0)
        (setq mul '2.0)
        (setq ifs (cadr rule))
        (setq ifs (apifs ifs))
        rtfloop
        (cond ((null ifs) (return (list min mul))))
        (setq prsval (prove (car ifs)))
        (cond ((equal prsval '-2.0) (return (list prsval
l mul)))
              ((lessp prsval min) (setq min prsval))
              ((equal mul '2.0) (setq mul prsval))
              (t (setq mul (* prsval mul))))
        (setq ifs (cdr ifs))
        (go rtfloop)))
```

```
/* PROVE attempts to prove a fact. It returns the certainty
   factor, if proved and -2.0, if cannot be proved.
```

```
(defun prove (fact)
  (prog (rl asscf mincf max success
        lmm mulcf lor prprob prod cf)
```

```

/* It tries to see if the fact is present in the facts as
e.
/* If found, the corresponding cf is returned.

```

```

      (setq asscf (recall fact))
      (cond ((not (equal asscf '-2.0)) (return asscf)
))

```

```

/* All the rules are found that can prove the fact.
/* A 'reverse' is done such that the rules are ordered by rule
y rule
/* number.

```

```

      (setq rl (inthen fact))
      (setq rl (reverse rl))

```

```

/* If no rule is found, the user is asked the validity of
f the fact.

```

```

      (cond ((null rl) (return (askuser fact))))

```

```

/* MAX* is initially set to 0. It calls RTESTIF with each
h rule.

```

```

/* Success is set to t, if the premises of any rule are
satisfied.

```

```

/* It also calculates max and lor depending on the value
of D.

```

```

      (setq max '0.0)
      (setq lor '0.0)
      vpart1
      (cond ((null rl) (go vpart2)))
      (setq lmm (rtestif (car rl)))
      (setq mincf (car lmm))
      (setq mulcf (cadr lmm))
      (cond ((lessp 0.0 mincf)
              (setq success t)
              (setq prprob (+ (* d mincf) (* (minus 1
d) mulcf)))
              (setq prod (findprod fact (car rl) prprob)
              (setq max (findmax fact (car rl) mincf max)
              (setq lor (minus (+ prod lor) (* prod lor)
r)))
              (cond ((greaterp (abs max) beta)
                      (go vpart2)))
              (setq rl (cdr rl))
              (go vpart1)
      vpart2

```

```

/* If any rule has succeeded, the fact is added to the fa
ctbase
/* and the composite cf is returned; otherwise, -2.0 is
returned.

```

```

      (cond ((equal success t)
             (setq cf (+ (* d max) (* (minus 1 d) lor
)))
            (insert fact cf)
            (return cf))
            (t (return '-2.0))) )

```

```

/* DIAGNOSE tries to prove each hypothesis in turn by PR
OVE,
/* until the certainty factor of a hypothesis >
alpha,
/* or all the hypotheses are exhausted.

```

```

(defun diagnose()
  (prog (poss cf)
    (setq poss hypotheses)
    dloop
    (cond ((null poss) (return nil)))
    (setq cf (prove (car poss)))
    (cond ((ge cf alpha)
           (printlist (caar poss) "is proved with
certainty " cf )
           (return (caar poss)) ))
    (setq poss (cdr poss))
    (go dloop)))

```

## APPENDIX B

### IMPLEMENTATION OF SUBJECTIVE BAYESIAN METHOD

A sample terminal session is shown to illustrate the use of the Subjective Bayesian Method to update existing probabilities of hypotheses to account for changing evidences. The rulebase with the necessity and sufficiency factors is given. The Lisp functions used to calculate the change in probabilities along with a forward chainer are attached.

## B.1 Subjective Bayesian Method: Sample Session

OK, lisp

WARNING This software is for evaluation only and contains a time out mechanism

```
> (co probes)
probes
update
matchone
tryrule
displayfacts
> (co rules)
rules
setup
> (setup)
1
> (co settracts_build)
settracts_build
settracts
> (settracts)
( ( ( region_class is tar) 0.1 0.1) ( ( region_shape is narrow_strip)
0.1 0.1) ( ( region_class is water) 0.1 0.1) ( ( regions_surrounding
have class=land) 0.1 0.1) ( ( regions_surrounding have class=desert)
0.1 0.1) ( ( region_width is small) 0.1 0.1) ( ( region_length is
large) 0.1 0.1) ( ( region_class is constructed) 0.1 0.87)
( (
region_shape is regular) 0.1 0.78) ( ( region_shape is rectangular)
0.1 0.1) ( ( two_regions_adjoining are river) 0.1 0.1) ( (
two_regions_adjoining are road) 0.1 0.1) ( ( regions_surrounding have
class=water) 0.1 0.1) ( ( region_shape is roughly_rectangular) 0.1 0.1
) ( ( region_class is land) 0.1 0.1) ( ( region_shape is irregular)
0.1 0.1) ( ( region_size is small) 0.1 0.1) ( ( regions_surrounding
have class=constructed) 0.1 0.1) ( ( region is road) 0.1 0.1) ( (
region is lake) 0.1 0.1) ( ( region is oasis) 0.1 0.1) ( (
region is
river) 0.1 0.1) ( ( region is building) 0.1 0.1) ( ( region
is bridge)
0.1 0.1) ( ( region is offshore_oilrig) 0.1 0.1) ( ( region
is ship)
0.1 0.1) ( ( region is island) 0.1 0.1) ( ( region is swimming_pool)
0.1 0.1))
```



> (update)  
FACTS

( region\_class is tar)  
( Initial probability : 0.1)  
( Final probability : 0.1)  
  
( region\_shape is narrow\_strip)  
( Initial probability : 0.1)  
( Final probability : 0.1)  
  
( region\_class is water)  
( Initial probability : 0.1)  
( Final probability : 0.1)  
  
( regions\_surrounding have class=land)  
( Initial probability : 0.1)  
( Final probability : 0.1)

Enter any character to continue.

c

( regions\_surrounding have class=desert)  
( Initial probability : 0.1)  
( Final probability : 0.1)  
  
( region\_width is small)  
( Initial probability : 0.1)  
( Final probability : 0.1)  
  
( region\_length is large)  
( Initial probability : 0.1)  
( Final probability : 0.1)  
  
( region\_class is constructed)  
( Initial probability : 0.1)  
( Final probability : 0.87)

Enter any character to continue.

c

( region\_shape is regular)  
( Initial probability : 0.1)  
( Final probability : 0.78)  
  
( region\_shape is rectangular)  
( Initial probability : 0.1)  
( Final probability : 0.1)  
  
( two\_regions\_adjoining are river)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( two\_regions\_adjoining are road)  
( Initial probability : 0.1)  
( Final probability : 0.1)

Enter any character to continue.

c

( regions\_surrounding have class=water)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_shape is roughly\_rectangular)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_class is land)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_shape is irregular)  
( Initial probability : 0.1)  
( Final probability : 0.1)

Enter any character to continue.

c

( region\_size is small)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( regions\_surrounding have class=constructed)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region is road)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region is lake)  
( Initial probability : 0.1)  
( Final probability : 0.1)

Enter any character to continue.

c

( region is oasis)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region is river)  
( Initial probability : 0.1)

```
( Final probability : 0.1)

( region is building)
( Initial probability : 0.1)
( Final probability : 0.868681481481)

( region is bridge)
( Initial probability : 0.1)
( Final probability : 0.142777777778)
```

Enter any character to continue.

c

```
( region is offshore_oilrig)
( Initial probability : 0.1)
( Final probability : 0.255654320988)

( region is ship)
( Initial probability : 0.1)
( Final probability : 0.249722222222)

( region is island)
( Initial probability : 0.1)
( Final probability : 0.1)

( region is swimming_pool)
( Initial probability : 0.1)
( Final probability : 0.175555555556)
```

Enter any character to continue.

c

t

```
> (co setracts_river)
  setracts_river
  setracts
> (setracts)
  ( ( ( region_class is tar) 0.1 0.1) ( ( region_shape is narrow_strip)
    0.1 0.1) ( ( region_class is water) 0.1 0.9) ( ( regions_surrounding
  have class=land) 0.1 0.1) ( ( regions_surrounding have class=desert)
  0.1 0.1) ( ( region_width is small) 0.1 0.83) ( ( region_length is
  large) 0.1 0.91) ( ( region_class is constructed) 0.1 0.1)
  ( (
  region_shape is regular) 0.1 0.1) ( ( region_shape is rectangular) 0.1
  0.1) ( ( two_regions_adjoining are river) 0.1 0.1) ( (
  two_regions_adjoining are road) 0.1 0.1) ( ( regions_surrounding have
  class=water) 0.1 0.1) ( ( region_shape is roughly_rectangular)
```

```
ar) 0.1 0.1
) ( ( region_class is land) 0.1 0.1) ( ( region_shape is ir
regular)
0.1 0.1) ( ( region_size is small) 0.1 0.1) ( ( regions_sur
rounding
have class=constructed) 0.1 0.1) ( ( region is road) 0.1 0.
1) ( (
region is lake) 0.1 0.1) ( ( region is oasis) 0.1 0.1) ( (
region is
river) 0.1 0.1) ( ( region is building) 0.1 0.1) ( ( region
is bridge)
0.1 0.1) ( ( region is offshore_oilrig) 0.1 0.1) ( ( regio
n is ship)
0.1 0.1) ( ( region is island) 0.1 0.1) ( ( region is swimm
ing_pool)
0.1 0.1))
> (update)
```

FACTS

```
( region_class is tar)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_shape is narrow_strip)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_class is water)
( Initial probability : 0.1)
( Final probability   : 0.9)

( regions_surrounding have class=land)
( Initial probability : 0.1)
( Final probability   : 0.1)
```

Enter any character to continue.

c

```
( regions_surrounding have class=desert)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_width is small)
( Initial probability : 0.1)
( Final probability   : 0.83)

( region_length is large)
( Initial probability : 0.1)
( Final probability   : 0.91)

( region_class is constructed)
( Initial probability : 0.1)
```

( Final probability : 0.1)

Enter any character to continue.

c

( region\_shape is regular)

( Initial probability : 0.1)

( Final probability : 0.1)

( region\_shape is rectangular)

( Initial probability : 0.1)

( Final probability : 0.1)

( two\_regions\_adjoining are river)

( Initial probability : 0.1)

( Final probability : 0.1)

( two\_regions\_adjoining are road)

( Initial probability : 0.1)

( Final probability : 0.1)

Enter any character to continue.

c

( regions\_surrounding have class=water)

( Initial probability : 0.1)

( Final probability : 0.1)

( region\_shape is roughly\_rectangular)

( Initial probability : 0.1)

( Final probability : 0.1)

( region\_class is land)

( Initial probability : 0.1)

( Final probability : 0.1)

( region\_shape is irregular)

( Initial probability : 0.1)

( Final probability : 0.1)

Enter any character to continue.

c

( region\_size is small)

( Initial probability : 0.1)

( Final probability : 0.1)

( regions\_surrounding have class=constructed)

( Initial probability : 0.1)

( Final probability : 0.1)

( region is road)

```
( Initial probability : 0.1)
( Final probability   : 0.1)

( region is lake)
( Initial probability : 0.1)
( Final probability   : 0.33111111111111)
```

Enter any character to continue.

c

```
( region is oasis)
( Initial probability : 0.1)
( Final probability   : 0.31333333333333)

( region is river)
( Initial probability : 0.1)
( Final probability   : 0.843374074074)

( region is building)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region is bridge)
( Initial probability : 0.1)
( Final probability   : 0.1)
```

Enter any character to continue.

c

```
( region is offshore_oilrig)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region is ship)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region is island)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region is swimming_pool)
( Initial probability : 0.1)
( Final probability   : 0.26)
```

Enter any character to continue.

c

```
t
> (co setracts_island)
setracts_island
setracts
> (setracts)
```

```

( ( ( region_class is tar) 0.1 0.1) ( ( region_shape is narrow_strip)
0.1 0.1) ( ( region_class is water) 0.1 0.1) ( ( regions_surrounding
have class=land) 0.1 0.1) ( ( regions_surrounding have class=desert)
0.1 0.1) ( ( region_width is small) 0.1 0.1) ( ( region_length is
large) 0.1 0.1) ( ( region_class is constructed) 0.1 0.1) (
(
region_shape is regular) 0.1 0.1) ( ( region_shape is rectangular) 0.1
0.1) ( ( two_regions_adjoining are river) 0.1 0.1) ( (
two_regions_adjoining are road) 0.1 0.1) ( ( regions_surrounding have
class=water) 0.1 0.92) ( ( region_shape is roughly_rectangular) 0.1
0.1) ( ( region_class is land) 0.1 0.88) ( ( region_shape is irregular)
0.1 0.76) ( ( region_size is small) 0.1 0.1) ( ( regions_surrounding
have class=constructed) 0.1 0.1) ( ( region is road) 0.1 0.1) ( (
region is lake) 0.1 0.1) ( ( region is oasis) 0.1 0.1) ( (
region is river) 0.1 0.1) ( ( region is building) 0.1 0.1) ( ( region
is bridge)
0.1 0.1) ( ( region is offshore_oilrig) 0.1 0.1) ( ( region is ship)
0.1 0.1) ( ( region is island) 0.1 0.1) ( ( region is swimming_pool)
0.1 0.1))
> (update)

```

FACTS

```

( region_class is tar)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_shape is narrow_strip)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_class is water)
( Initial probability : 0.1)
( Final probability   : 0.1)

( regions_surrounding have class=land)
( Initial probability : 0.1)
( Final probability   : 0.1)

```

Enter any character to continue.

c

```
( regions_surrounding have class=desert)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_width is small)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_length is large)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_class is constructed)
( Initial probability : 0.1)
( Final probability   : 0.1)
```

Enter any character to continue.

c

```
( region_shape is regular)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_shape is rectangular)
( Initial probability : 0.1)
( Final probability   : 0.1)

( two_regions_adjoning are river)
( Initial probability : 0.1)
( Final probability   : 0.1)

( two_regions_adjoning are road)
( Initial probability : 0.1)
( Final probability   : 0.1)
```

Enter any character to continue.

c

```
( regions_surrounding have class=water)
( Initial probability : 0.1)
( Final probability   : 0.92)

( region_shape is roughly_rectangular)
( Initial probability : 0.1)
( Final probability   : 0.1)

( region_class is land)
( Initial probability : 0.1)
( Final probability   : 0.88)
```



( region\_shape is irregular)  
( Initial probability : 0.1)  
( Final probability : 0.76)

Enter any character to continue.

c

( region\_size is small)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( regions\_surrounding have class=constructed)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region is road)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region is lake)  
( Initial probability : 0.1)  
( Final probability : 0.1)

Enter any character to continue.

c

( region is oasis)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region is river)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region is building)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region is bridge)  
( Initial probability : 0.1)  
( Final probability : 0.1)

Enter any character to continue.

c

( region is offshore\_oilrig)  
( Initial probability : 0.1)  
( Final probability : 0.327777777778)

( region is ship)  
( Initial probability : 0.1)  
( Final probability : 0.259444444444)

```
( region is island)
( Initial probability : 0.1)
( Final probability   : 0.86158108642)
```

```
( region is swimming_pool)
( Initial probability : 0.1)
( Final probability   : 0.1)
```

Enter any character to continue.

c

t

```
> (co settracts_bridge)
  settracts_bridge
  settracts
> (settracts)
  ( ( ( region_class is tar) 0.1 0.1) ( ( region_shape is narrow_strip)
  0.1 0.1) ( ( region_class is water) 0.1 0.1) ( ( regions_surrounding
  have class=land) 0.1 0.1) ( ( regions_surrounding have class=desert)
  0.1 0.1) ( ( region_width is small) 0.1 0.1) ( ( region_length is
  large) 0.1 0.1) ( ( region_class is constructed) 0.1 0.1) (
  (
  region_shape is regular) 0.1 0.1) ( ( region_shape is rectangular) 0.1
  0.77) ( ( two_regions_adjoining are river) 0.1 0.93) ( (
  two_regions_adjoining are road) 0.1 0.85) ( ( regions_surrounding have
  class=water) 0.1 0.1) ( ( region_shape is roughly_rectangular) 0.1
  0.1) ( ( region_class is land) 0.1 0.1) ( ( region_shape is irregular)
  0.1 0.1) ( ( region_size is small) 0.1 0.1) ( ( regions_surrounding
  have class=constructed) 0.1 0.89) ( ( region is road) 0.1 0.1) ( (
  region is lake) 0.1 0.1) ( ( region is oasis) 0.1 0.1) ( (
  region is
  river) 0.1 0.1) ( ( region is building) 0.1 0.1) ( ( region
  is bridge)
  0.1 0.1) ( ( region is offshore_oilrig) 0.1 0.1) ( ( region
  is ship)
  0.1 0.1) ( ( region is island) 0.1 0.1) ( ( region is swimming_pool)
  0.1 0.1))
> (update)
  FACTS

  ( region_class is tar)
```

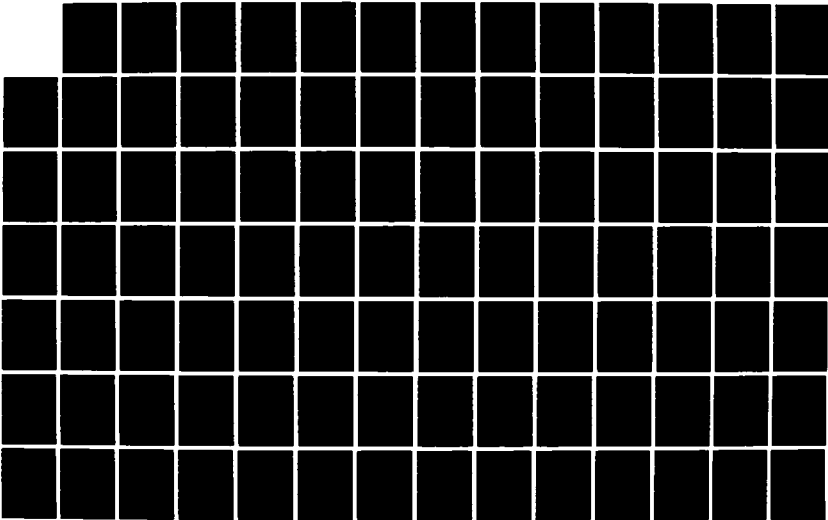
AD-A189 774

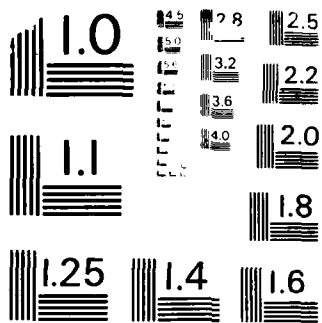
NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM (NAIC)  
REVIEW OF TECHNICAL T. (U) NORTHEAST ARTIFICIAL  
INTELLIGENCE CONSORTIUM SYRACUSE NY J F ALLEN ET AL.  
JUL 87 RADC-TR-86-218-V2-PT2 F/G 12/9

2/5

UNCLASSIFIED

ML





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS - 1963

( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_shape is narrow\_strip)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_class is water)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( regions\_surrounding have class=land)  
( Initial probability : 0.1)  
( Final probability : 0.1)

Enter any character to continue.

c

( regions\_surrounding have class=desert)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_width is small)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_length is large)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_class is constructed)  
( Initial probability : 0.1)  
( Final probability : 0.1)

Enter any character to continue.

c

( region\_shape is regular)  
( Initial probability : 0.1)  
( Final probability : 0.1)

( region\_shape is rectangular)  
( Initial probability : 0.1)  
( Final probability : 0.77)

( two\_regions\_adjoning are river)  
( Initial probability : 0.1)  
( Final probability : 0.93)

( two\_regions\_adjoning are road)  
( Initial probability : 0.1)  
( Final probability : 0.85)

Enter any character to continue.

c

( regions\_surrounding have class=water)

( Initial probability : 0.1)

( Final probability : 0.1)

( region\_shape is roughly\_rectangular)

( Initial probability : 0.1)

( Final probability : 0.1)

( region\_class is land)

( Initial probability : 0.1)

( Final probability : 0.1)

( region\_shape is irregular)

( Initial probability : 0.1)

( Final probability : 0.1)

Enter any character to continue.

c

( region\_size is small)

( Initial probability : 0.1)

( Final probability : 0.1)

( regions\_surrounding have class=constructed)

( Initial probability : 0.1)

( Final probability : 0.89)

( region is road)

( Initial probability : 0.1)

( Final probability : 0.1)

( region is lake)

( Initial probability : 0.1)

( Final probability : 0.1)

Enter any character to continue.

c

( region is oasis)

( Initial probability : 0.1)

( Final probability : 0.1)

( region is river)

( Initial probability : 0.1)

( Final probability : 0.1)

( region is building)

( Initial probability : 0.1)

```
( Final probability : 0.1)
( region is bridge)
( Initial probability : 0.1)
( Final probability : 0.529982716049)
```

Enter any character to continue.

c

```
( region is offshore_oilrig)
( Initial probability : 0.1)
( Final probability : 0.1)

( region is ship)
( Initial probability : 0.1)
( Final probability : 0.1)

( region is island)
( Initial probability : 0.1)
( Final probability : 0.1)

( region is swimming_pool)
( Initial probability : 0.1)
( Final probability : 0.117555555556)
```

Enter any character to continue.

c

t

> q

OK, como -e

## B.2 Subjective Bayesian Method: Rulebase

```
(defun setup ()
  (setq rules '(
    (rule1 (if (region_class is tar))
            (then (region is road) 0.5 3.5))
    (rule2 (if (region_shape is narrow_strip))
            (then (region is road) 0.75 3.0))
    (rule3 (if (region_class is water))
            (then (region is lake) 0.0 3.6))
    (rule4 (if (regions_surrounding have class=land)
            (then (region is lake) 0.1 2.75))
    (rule5 (if (region_class is water))
            (then (region is oasis) 0.0 3.4))
    (rule6 (if (regions_surrounding have class=dese
    rt))
            (then (region is oasis) 0.1 3.0))
    (rule7 (if (region_class is water))
            (then (region is river) 0.0 4.0))
```

```

(rule8 (if (region_width is small))
        (then (region is river) 0.1 2.0))
(rule9 (if (region_length is large))
        (then (region is river) 0.75 1.3))
(rule10 (if (region_class is constructed))
         (then (region is building) 0.1 4.0))
(rule11 (if (region_shape is regular))
         (then (region is building) 0.1 2.9))
(rule12 (if (region_class is constructed))
         (then (region is bridge) 0.1 1.5))
(rule13 (if (region_shape is rectangular))
         (then (region is bridge) 0.5 1.5))
(rule14 (if (two_regions_adjoining are river))
         (then (region is bridge) 0.1 2.2))
(rule15 (if (two_regions_adjoining are road))
         (then (region is bridge) 0.1 2.0))
(rule16 (if (region_class is constructed))
         (then (region is offshore_oilrig) 0.1 2
.0))
(rule17 (if (region_shape is regular))
         (then (region is offshore_oilrig) 0.1 1
.5))
(rule18 (if (regions_surrounding have class=wat
er))
         (then (region is offshore_oilrig) 0.1 3
.5))
(rule19 (if (region_class is constructed))
         (then (region is ship) 0.1 2.75))
(rule20 (if (region_shape is roughly_rectangula
r))
         (then (region is ship) 0.1 1.5))
(rule21 (if (regions_surrounding have class=wat
er))
         (then (region is ship) 0.1 2.75))
(rule22 (if (region_class is land))
         (then (region is island) 0.1 2.7))
(rule23 (if (regions_surrounding have class=wat
er))
         (then (region is island) 0.1 2.7))
(rule24 (if (region_shape is irregular))
         (then (region is island) 0.75 1.5))
(rule25 (if (region_class is water))
         (then (region is swimming_pool) 0.0 2.8
))
(rule26 (if (region_shape is regular))
         (then (region is swimming_pool) 0.0 2.0
))
(rule27 (if (region_size is small))
         (then (region is swimming_pool) 0.1 1.5
))
(rule28 (if (regions_surrounding have class=con
structed))

```



```

        (then (region is swimming_pool) 0.1 1.2
    ))))
    (setq threshold '1) )

```

### B.3 Subjective Bayesian Method: Lisp Functions

```

/*
/*      FORWARD    CHAINER    UPDATING    PROBABILITY
/*      BY SUBJECTIVE BAYESIAN METHOD
/*

/*      UPDATE should be triggered whenever any of the
/*      probabilities of the evidences are updated.

    (defun update ()
      (prog (progress)
        (cond ((matchone) (setq progress t)))
        (displayfacts) (return progress)))

/*      MATCHONE tries one rule at a time.

    (defun matchone ()
      (prog (morules)
        (setq morules rules)
        mloop
        (cond ((null morules) (return t)))
        (tryrule (car morules))
        (setq morules (cdr morules))
        (go mloop)))

/*      TRYRULE checks whether the probabilities of the
/*      premise have changed. If so, it calculates the
/*      new probability of the hypothesis.

    (defun tryrule (rule)
      (prog (iffact thenfact trfacts iprob fprob bm nf s
f)
        (setq iffact (cadadr rule))
        (setq thenfact (cadaddr rule))
        (setq trfacts facts)
        findiffact
        (cond ((null trfacts)
          (print iffact)
          (print "List of facts incomplete.")
          (return nil))
          ((equal (caar trfacts) iffact)
          (go findbm)))
        (setq trfacts (cdr trfacts))
        (go findiffact)

```

```

findbm
(setq iprob (cadr (car trfacts)))
(setq fprob (caddr (car trfacts)))
(setq nf (car (cddaddr rule)))
(setq sf (cadr (cddaddr rule)))
(cond ((equal iprob fprob) (return nil))
      ((greaterp fprob iprob)
       (setq bm (+ '1 (* (- fprob iprob)
                          (/ (- sf '1) (- '1
iprob))))))
      (t (setq bm (+ nf (* fprob (/ (- '1 nf
) fprob))))))
(setq trfacts facts)
(setq afacts '() )
changeprob
(cond ((null trfacts)
      (print "Then fact of rule not includ
ed in facts list")
      (return nil))
      ((equal (caar trfacts ) thenfact)
       (setq nprob (* bm (caddr (car trfacts
)))
              (setq x (list (caar trfacts)
                             (caddr trfacts)
                             nprob))
              (setq afacts (append afacts (list x)
(cdr trfacts)))
              (setq facts afacts)
              (return t)))
      (setq afacts (append afacts (list (car trfac
ts))))
      (setq trfacts (cdr trfacts))
      (go changeprob)))

```

```

/* DISPLAYFACTS shows the initial and final probabilitie
s
of all the facts in the factsbase on the screen.

```

```

(defun displayfacts ()
  (prog (dffacts)
    (setq dffacts facts)
    (print "          FACTS          ")
    (setq counter '0)
    loop
    (cond ((null dffacts) (return t)))
    (print "          ")
    (print (caar dffacts))
    (print (list "Initial probability : "
                (cadr (car dffacts))))
    (print (list "Final probability   : "
                (caddr (car dffacts))))

```

nue.")

```
(setq dffacts (cdr dffacts))
(setq counter (add1 counter))
(cond ((equal counter '4)
      (setq counter '0)
      (print ' " " )
      (print '"Enter any character to conti
            (setq x (read)) ))
      (go loop)))
```

## APPENDIX C

### IMPLEMENTATION OF DEMPSTER-SHAFER METHOD

The method of associating the degrees of belief and plausibility and thus gauging the validity of conclusions is demonstrated with the help of two sample sessions. The rules used in connection with this method are given. The Lisp functions, that determine the probability ranges of the hypotheses, are also attached.

C.1 Dempster-Shafer Method: Sample session 1

OK, lisp

WARNING This software is for evaluation only and contains a  
time out mechanism

Salford University Lisp version 33

> (co dst)

dst  
ge  
insert  
recall  
inthen  
thenp  
finddc  
askuser  
apifs  
rtestif  
prove  
diagnose

> (co setup)

setup  
setup

> (setup)

()

> (diagnose)

Enter probability of fact being true : ( region shape rou  
nd)

0.97

Enter probability of fact being false : ( region shape ro  
und)

0.01

Enter probability of fact being true : ( region class ste  
el)

0.8

Enter probability of fact being false : ( region class st  
eel)

0.1

( region is oiltank) is proved with certainty range ( 0.6  
48 0.576E-1

)

( region is oiltank)

> (setup)

()

> (diagnose)

Enter probability of fact being true : ( region shape rou  
nd)

no

Enter probability of fact being false : ( region shape ro  
und)

0.0

Enter probability of fact being true : ( region shape reg  
ular)

```

0.91
Enter probability of fact being false : ( region shape re
gular)
0.05
Enter probability of fact being true : ( region shape
long_and_narrow)
0.92
Enter probability of fact being false : ( region shape
long_and_narrow)
0.05
Enter probability of fact being true : ( region class tar
)
0.85
Enter probability of fact being false : ( region class ta
r)
0.1
( region is road) is proved with certainty range ( 0.6885
0.612E-1)
( region is road)
> (setup)
( )
> (diagnose)
Enter probability of fact being true : ( region shape rou
nd)
no
Enter probability of fact being false : ( region shape ro
und)
1.0
Enter probability of fact being true : ( region shape reg
ular)
no
Enter probability of fact being false : ( region shape re
gular)
1.0
Enter probability of fact being true : ( region shape rec
tangular)
0.82
Enter probability of fact being false : ( region shape re
ctangular)
0.1
Enter probability of fact being true : ( region
adjacent_on_two_sides_to region_with_structure_class_2)
0.86
Enter probability of fact being false : ( region
adjacent_on_two_sides_to region_with_structure_class_2)
0.02
Enter probability of fact being true : ( region
adjacent_on_two_sides_to region_is_road)
0.7
Enter probability of fact being false : ( region
adjacent_on_two_sides_to region_is_road)
0.1

```

```
Enter probability of fact being true : ( region class tar
)
0.89
Enter probability of fact being false : ( region class ta
r)
0.1
( region is bridge) is proved with certainty range ( 0.63
0.56E-1)
( region is bridge)
> (setup)
( )
> (diagnose)
Enter probability of fact being true : ( region shape rou
nd)
no
Enter probability of fact being false : ( region shape ro
und)
1.0
Enter probability of fact being true : ( region shape reg
ular)
no
Enter probability of fact being false : ( region shape re
gular)
1.0
Enter probability of fact being true : ( region shape rec
tangular)
no
Enter probability of fact being false : ( region shape re
ctangular)
1.0
Enter probability of fact being true : ( region class con
crete)
no
Enter probability of fact being false : ( region class co
ncrete)
1.0
Enter probability of fact being true : ( region surroundi
ng_regions
region_with structure_class_2)
0.9
Enter probability of fact being false : ( region surround
ing_regions
region_with structure_class_2)
0.02
Enter probability of fact being true : ( region shape irr
egular)
0.94
Enter probability of fact being false : ( region shape ir
regular)
0.05
Enter probability of fact being true : ( region class lan
d)
```

```

0.96
  Enter probability of fact being false : ( region class la
nd)
0.02
  ( region is island) is proved with certainty range ( 0.77
76
0.6912E-1)
  ( region is island)
> (co setup)
  setup
  setup
> (setup)
  ()
> (diagnose)
  Enter probability of fact being true : ( region shape rou
nd)
0.0
  Enter probability of fact being false : ( region shape ro
und)
1.0
  Enter probability of fact being true : ( region shape reg
ular)
0.0
  Enter probability of fact being false : ( region shape re
gular)
1.0
  Enter probability of fact being true : ( region shape rec
tangular)
0.0
  Enter probability of fact being false : ( region shape re
ctangular)
1.0
  Enter probability of fact being true : ( region surroundi
ng_regions
region _with structure_class_2)
0.0
  Enter probability of fact being false : ( region surround
ing_regions
region _with structure_class_2)
1.0
  Enter probability of fact being true : ( region surroundi
ng_regions
regions_with_structure_class_1)
0.0
  Enter probability of fact being false : ( region surround
ing_regions
regions_with_structure_class_1)
1.0
  Enter probability of fact being true : ( region shape
long_and_narrow)
0.9
  Enter probability of fact being false : ( region shape

```



```

long_and_narrow)
0.02
  Enter probability of fact being true : ( region class water)
0.95
  Enter probability of fact being false : ( region class water)
0.03
  ( region is river) is proved with certainty range ( 0.7695 0.684E-1)
  ( region is river)
> q
OK, como -e

```

## C.2 Dempster-Shafer Method: Sample\_session\_2

```

OK, lisp
WARNING This software is for evaluation only and contains a
time out mechanism
      Salford University Lisp version 33
> (co dst)
dst
ge
insert
recall
inthen
thenp
finddc
askuser
apifs
rtestif
prove
diagnose
> (co setup)
setup
setup
> (setup)
()
> (diagnose)
  Enter probability of fact being true : ( region surrounding_regions
region_with_structure_class_2)
0.9
  Enter probability of fact being false : ( region surrounding_regions
region_with_structure_class_2)
0.05
  Enter probability of fact being true : ( region shape regular)
0.95
  Enter probability of fact being false : ( region shape re

```

```

gular)
0.02
  Enter probability of fact being true : ( region class art
ificial)
0.8
  Enter probability of fact being false : ( region class ar
tificial)
0.1
  ( region is ship) is proved with certainty range ( 0.648
0.576E-1)
  ( region is ship)
> (setup)
  ( )
> (diagnose)
  Enter probability of fact being true : ( region surroundi
ng_regions
region_with_structure_class_2)
0.6
  Enter probability of fact being false : ( region surround
ing_regions
region_with_structure_class_2)
0.3
  Enter probability of fact being true : ( region shape reg
ular)
0.92
  Enter probability of fact being false : ( region shape re
gular)
0.04
  Enter probability of fact being true : ( region class art
ificial)
0.9
  Enter probability of fact being false : ( region class ar
tificial)
0.02
  ( region is ship) is proved with certainty range ( 0.54 0
.48E-1)
  ( region is ship)
> (setup)
  ( )
> (diagnose)
  Enter probability of fact being true : ( region surroundi
ng_regions
region_with_structure_class_2)
0.0
  Enter probability of fact being false : ( region surround
ing_regions
region_with_structure_class_2)
1.0
  Enter probability of fact being true : ( region
most_surrounding_regions region_with_structure_class_2)
0.9
  Enter probability of fact being false : ( region

```

```

most_surrounding_regions region_with_structure_class_2)
0.05
  Enter probability of fact being true : ( region shape regular)
0.8
  Enter probability of fact being false : ( region shape regular)
0.1
  Enter probability of fact being true : ( region class artificial)
0.92
  Enter probability of fact being false : ( region class artificial)
0.04
  ( region is offshore_oilrig) is proved with certainty range ( 0.72
0.64E-1)
  ( region is offshore_oilrig)
> (setup)
  ( )
> (diagnose)
  Enter probability of fact being true : ( region surrounding_regions
region_with_structure_class_2)
0.0
  Enter probability of fact being false : ( region surrounding_regions
region_with_structure_class_2)
1.0
  Enter probability of fact being true : ( region
most_surrounding_regions region_with_structure_class_2)
0.0
  Enter probability of fact being false : ( region
most_surrounding_regions region_with_structure_class_2)
1.0
  Enter probability of fact being true : ( region adjacent_to
region_with_structure_class_2)
0.9
  Enter probability of fact being false : ( region adjacent_to
region_with_structure_class_2)
0.06
  Enter probability of fact being true : ( region adjacent_to
region_with_structure_class_1)
0.82
  Enter probability of fact being false : ( region adjacent_to
region_with_structure_class_1)
0.14
  Enter probability of fact being true : ( region shape

```

```

long_and_narrow)
0.85
  Enter probability of fact being false : ( region shape
long_and_narrow)
0.12
  Enter probability of fact being true : ( region shape reg
ular)
0.95
  Enter probability of fact being false : ( region shape re
gular)
0.02
  Enter probability of fact being true : ( region class art
ificial)
0.97
  Enter probability of fact being false : ( region class ar
tificial)
0.03
  ( region is dock) is proved with certainty range ( 0.738
0.656E-1)
  ( region is dock)
> (setup)
  ()
> (diagnose)
  Enter probability of fact being true : ( region surroundi
ng_regions
region_with_structure_class_2)
unknown
  Enter probability of fact being false : ( region surround
ing_regions
region_with_structure_class_2)
unknown
  Enter probability of fact being true : ( region
most_surrounding_regions region_with_structure_class_2)
unknown
  Enter probability of fact being false : ( region
most_surrounding_regions region_with_structure_class_2)
unknown
  Enter probability of fact being true : ( region adjacent_
to
region_with_structure_class_2)
unknown
  Enter probability of fact being false : ( region adjacent
_to
region_with_structure_class_2)
unknown
  Enter probability of fact being true : ( region width sma
ll)
0.0
  Enter probability of fact being false : ( region width sm
all)
1.0
  Enter probability of fact being true : ( region shape reg

```

```

ular)
0.9
Enter probability of fact being false : ( region shape re
gular)
0.04
Enter probability of fact being true : ( region class gre
enery)
0.97
Enter probability of fact being false : ( region class gr
eenery)
0.02
( region is field) is proved with certainty range ( 0.742
05
0.4365E-1)
( region is field)
> q
OK, como -e

```

### C.3 Dempster-Shafer Method: Rulebase

```

(defun setup ()
  (setq rules '(
    (r1: ((region class land))
1) (0.9 0.05)))
    (r2: ((region class water))
2) (0.9 0.05)))
    (r3: ((region class artificial))
3) (0.9 0.05)))
    (r4: ((region class greenery))
11) (0.9 0.08)))
    (r5: ((region class sand))
12) (0.9 0.08)))
    (r6: ((region class marsh))
13) (0.9 0.08)))
    (r7: ((region class ice))
14) (0.9 0.08)))

```

```

      (r8:      ((region class tar))
31) (0.9 0.08)))      (((region belongs_to structure_class_
      (r9:      ((region class concrete))
32) (0.9 0.08)))      (((region belongs_to structure_class_
      (r10:     ((region class steel))
33) (0.9 0.08)))      (((region belongs_to structure_class_
1)      (r11:     ((region belongs_to structure_class_1
      (region shape regular))
      (((region is field) (0.85 0.05))))
1)      (r12:     ((region belongs_to structure_class_1
      (region shape irregular))
      (((region is meadow) (0.85 0.05))))
2)      (r13:     ((region belongs_to structure_class_1
      (region shape irregular))
      (((region is desert) (0.85 0.05))))
4)      (r14:     ((region belongs_to structure_class_1
      (region shape irregular))
      (((region is glacier) (0.85 0.05))))
2)      (r15:     ((region belongs_to structure_class_1
      (region adjacent_to region_with_stru
cture_class_1)
      (region adjacent_to region_with_stru
cture_class_2)
      (region width small))
      (((region is beach) (0.9 0.08))))
3)      (r16:     ((region belongs_to structure_class_1
      (region adjacent_to region_with_stru
cture_class_1)
      (region adjacent_to region_with_stru
cture_class_2))
      (((region is marsh) (0.9 0.08))))
      (r17:     ((region belongs_to structure_class_1
)

```

```

                (region shape irregular)
                (region surrounding_regions
                  region_with_structure_class_
2))
                (((region is island) (0.9 0.08))))
                (r18: ((region belongs_to structure_class_2
)
                (region shape irregular)
                (region surrounding_regions
                  regions_with_structure_class
_1))
                (((region is lake) (0.9 0.08))))
                (r19: ((region belongs_to structure_class_2
)
                (region shape long_and_narrow))
                (((region is river) (0.9 0.08))))
                (r20: ((region belongs_to structure_class_2
)
                (region shape irregular)
                (region surrounding_regions
                  region_with_structure_class_
12))
                (((region is oasis) (0.9 0.08))))
                (r21: ((region belongs_to structure_class_3
)
                (region shape regular)
                (region surrounding_regions
                  region_with_structure_class_
2))
                (((region is ship) (0.9 0.08))))
                (r22: ((region belongs_to structure_class_3
)
                (region shape regular)
                (region most_surrounding_regions
                  region_with_structure_class_
2))
                (((region is offshore_oilrig) (0.9 0.
08))))
                (r23: ((region belongs_to structure_class_3
)
                (region shape regular)
                (region shape long_and_narrow)
                (region adjacent_to
                  region_with_structure_class_
1)
                (region adjacent_to

```

```

                region_with_structure_class_
2))
                ((region is dock) (0.9 0.08)))
1)
    (r24: ((region belongs_to structure_class_3
           (region shape long_and_narrow)
           (region shape regular))
          ((region is road) (0.9 0.08))))
3)
    (r25: ((region belongs_to structure_class_3
           (region shape round))
          ((region is oiltank) (0.9 0.08))))
2)
    (r26: ((region belongs_to structure_class_3
           (region shape regular))
          ((region is building) (0.9 0.08))))
1)
    (r27: ((region belongs_to structure_class_3
           (region adjacent_on_two_sides_to
            region_is_road)
           (region adjacent_on_two_sides_to
            region_with_structure_class_
2)
           (region shape rectangular))
          ((region is bridge) (0.9 0.08))))
3)
    (r28: ((region belongs_to structure_class_3
           (region shape airplane_like))
          ((region is airplane) (0.7 0.2))))
))

```

```
(setq hypotheses '(
```

```

((region is offshore_oilrig) (0.0 0.0))
((region is beach) (0.0 0.0))
((region is marsh) (0.0 0.0))
((region is field) (0.0 0.0))
((region is meadow) (0.0 0.0))
((region is desert) (0.0 0.0))
((region is glacier) (0.0 0.0))
((region is oasis) (0.0 0.0))
((region is airplane) (0.0 0.0))
((region is oiltank) (0.0 0.0))
((region is road) (0.0 0.0))
((region is bridge) (0.0 0.0))
((region is ship) (0.0 0.0))
((region is dock) (0.0 0.0))

```



```

((region is building)      (0.0 0.0))
((region is island)       (0.0 0.0))
((region is lake)         (0.0 0.0))
((region is river)        (0.0 0.0))

(setq alpha '0.5)
(setq beta  '0.8)
(setq facts '())

```

#### C.4 Dempster-Shafer Method: Lisp Functions

```

/*
/*      BACKWARD CHAINER WITH
/*      THE PROBABILITY RANGE CALCULATED
/*      ACCORDING TO THE DEMPSTER SHAFER METH
OD
/*

```

```

/* GE returns t if a > b or a = b, otherwise returns nil.

```

```

(defun ge (a b)
  (prog ()
    (cond (greaterp a b return t)
          (equal a b return t)
          (t return nil)))

```

```

/* INSERT add a 'fact' to the factbase with a
/* probability factor range cfr.

```

```

(defun insert (fact cfr)
  (setq facts (cons (list (car fact) cfr) facts)))

```

```

/* RECALL checks whether a fact is present in the fact ba
se.
/* If present, it returns the associated cfr, othe
rwise
/* it returns -2.0.

```

```

(defun recall (fact)
  (prog (rcfacts)
    (setq rcfacts facts)

```

```

rcloop
  (cond ((null rcfacts) (return '-2.0))
        ((equal (car fact) (caar rcfacts))
         (return (cadar rcfacts))))
  (setq rcfacts (cdr rcfacts))
  (go rclloop) )

```

```

/*  INTHEN strings together and returns a list of rules,
/*  each of which can prove the fact.

```

```

(defun inthen (fact)
  (prog (itrules oprules)
    (setq itrules rules)
    (setq oprules '())
    itloop
    (cond ((null itrules) (return oprules))
          ((thenp fact (car itrules))
           (setq oprules (cons (car itrules) oprules)
                    (setq itrules (cdr itrules))
                    (go itloop)))
          (t)
          (setq itrules (cdr itrules))
          (go itloop)))
  s)))

```

```

/*  THENP determines whether a fact is part of the RHS of
/*  a rule.

```

```

(defun thenp (fact rule)
  (prog (thens)
    (setq thens (caddr rule))
    thloop
    (cond ((null thens) (return nil))
          ((equal (car fact) (caar thens))
           (return t))
          (t)
          (setq thens (cdr thens))
          (go thloop) ))

```

```

/*  FINDDC finds the cumulative probability range at th
/*  OR juncture of the tree. It first has to
/*  calculate the prob range due to the rule its
elf.

```

```

(defun finddc (fact rule cfrange upnow)
  (prog (thens fdcfr a b c d abar bbar cbar dbar x y a
         dpbc)
    (setq thens (caddr rule))

```

```

fmloop1
(cond ((null thens) (print 'Error_in_findmax))
      ((equal (car fact) (caar thens))
       (setq fdcfr (cadr thens))
       (go fmloop2)))
(setq thens (cdr thens))
(go fmloop1)
fmloop2
(setq a (* (car cfrange) (car fdcfr)))
(setq b (* (car cfrange) (cadr fdcfr)))
(setq c (car upnow))
(setq d (cadr upnow))
(setq abar (minus 1 a))
(setq bbar (minus 1 b))
(setq cbar (minus 1 c))
(setq dbar (minus 1 d))
(setq adpbc (+ (* a d) (* b c)))
1 adpbc)))
1 adpbc)))
(setq x (minus 1 (divide (* abar cbar) (minus
1 adpbc))))
(setq y (minus 1 (divide (* bbar dbar) (minus
1 adpbc))))
(return (list x y)) )

```

```

/* ASKUSER obtains the certainty factor for a fact from
the user,
/* does the required conversion from YES, UNKNO
WN, NO,
/* call INSERT to add the fact to the fact base
and
/* returns the probability range.

```

```

(defun askuser(fact)
(prog (aucfl aucf2)
      (printlist "Enter probability of fact being t
rue : "
                (car fact) )
      mark1
      (setq aucfl (read))
      (cond ((equal aucfl 'yes) (setq aucfl '1.0) (g
o mark2))
            ((equal aucfl 'unknown) (setq aucfl '0.0
) (go mark2))
            ((equal aucfl 'no) (setq aucfl '-1.0) (g
o mark2))
            ((numberp aucfl) (go mark2))
            (t (print "ERROR! Please enter again.")
               (go mark1)))
      mark2
      (printlist "Enter probability of fact being f
alse : "
                (car fact))

```

```

mark3
(setq aucf2 (read))
(cond ((equal aucf2 'yes) (setq aucf2 '1.0) (g
o mark4))
      ((equal aucf2 'unknown) (setq aucf2 '0.0
)
      (go mark4))
      ((equal aucf2 'no) (setq aucf2 '-1.0) (g
o mark4))
      ((numberp aucf2) (go mark4))
      (t (print "ERROR! Please enter again.")
(go mark3)))
mark4
(insert fact (list aucf1 aucf2))
(return (list aucf1 aucf2)) )

```

```

/*  APIFS adds a (0.0 0.0) probability range to the premi
se list
/*  to make an appropriate format for comparison wi
th
/*  the fact base.

```

```

(defun apifs (ifs)
  (prog (inifs opifs)
    (setq inifs ifs)
    (setq opifs '())
    ailoop
    (cond ((null inifs) (return opifs)))
    (setq opifs (cons (list (car inifs) '(0.0 0.0))
opifs))
    (setq inifs (cdr inifs))
    (go ailoop)))

```

```

/*  RTESTIF forms part of the recursive loop. It tries t
o prove
/*  each of the premises of the 'rule' by invoki
ng PROVE.
/*  If any premise cannot be proven (cf < 0), it
returns
/*  -2.0, otherwise it returns the lowest cf fou
nd
/*  amongst the premises.
/*  (Here, cf represents the first value in the
probability range, i.e. the min degree to wh
ich
/*  the fact can be confirmed.)

```

```

(defun rtestif (rule)
  (prog (ifs x y prsval)
    (setq x '10.0)
    (setq y '10.0)
    (setq ifs (cadr rule))
    (setq ifs (apifs ifs))
    rtfloop
    (cond ((null ifs) (return (list x y))))
    (setq prsval (prove (car ifs)))
    (cond ((ge '0.0 (car prsval)) (return prsval))
          ((lessp (car prsval) x) (setq x (car prsval)
    al)))
          ((lessp (cadr prsval) y) (setq y (cadr prsval)
    sval))))
    (setq ifs (cdr ifs))
    (go rtfloop)))

```

```

/* PROVE attempts to prove a fact. It returns the probability,
/* if proved and (0.0 0.0), if cannot be proved.

```

```

(defun prove (fact)
  (prog (rl asscf cfr dc success)

```

```

/* It tries to see if the fact is present in the factbase.
/* If found, the corresponding cf is returned.

```

```

      (setq asscf (recall fact))
      (cond ((not (equal asscf '-2.0)) (return '(0.0
0.0))))

```

```

/* All the rules are found that can prove the fact.
/* A 'reverse' is done such that the rules are ordered by rule
/* number.

```

```

      (setq rl (inthen fact))
      (setq rl (reverse rl))

```

```

/* If no rule is found, the user is asked the validity of the fact.

```

```

      (cond ((null rl) (return (askuser fact))))

```

```

/* DC is initially set to (0 0). It calls RTESTIF with each rule.
/* Success is set to t, if the premises of any rule are satisfied.
/* It also updates dc with each rule satisfied.

```

```

    (setq dc '(0.0 0.0))
    vpart1
    (cond ((null rl) (go vpart2)))
    (setq cfr (rtestif (car rl)))
    (cond ((lessp 0.0 (car cfr))
           (setq success t)
           (setq dc (finddc fact (car rl) cfr dc)))
          )
    (setq rl (cdr rl))
    (go vpart1)
    vpart2

/* If any rule has succeeded, the fact is added to the fa
ctbase
/* and the cfr is returned; otherwise, (0 0) is returned.

    (cond ((equal success t)
           (insert fact dc)
           (return dc))
          (t (return '(0.0 0.0)))) )

/* DIAGNOSE tries to prove each hypothesis in turn by PR
OVE,
/* until the certainty factor of a hypothesis >
alpha,
/* or all the hypotheses are exhausted.

(defun diagnose()
  (prog (poss cfr)
    (setq poss hypotheses)
    dloop
    (cond ((null poss) (return nil)))
    (setq cfr (prove (car poss)))
    (cond ((ge (car cfr) alpha)
           (printlist (caar poss) "is proved with
certainty range "
                      cfr )
           (return (caar poss)) )
          )
    (setq poss (cdr poss))
    (go dloop)))

```

## APPENDIX D

### LISTING OF THE INFERENCE ENGINE

The Lisp functions of the combined backward and forward chainer, which incorporate the rule syntax mentioned above, is included next. This calculates the probabilities of the hypotheses with the help of certainty factors.

```

(defun usethenforward (rule)
  (prog (thens success verb cfvalue flagval
ue new_fact)
        (setq thens (cdr (caddr rule)))
        (print 'in_usethenforward)
        loop
        (setq verb (caar thens))
        (cond ((equal verb 'ask_y)
                (setq cfvalue (car (cdddar t
hens))))
              (T (setq cfvalue (caddar then
s))))
        (setq flagvalue 'system_inferred)
        (setq new_fact (cadar thens))
        (cond ((null thens) (return success)
              ((equal verb 'ask_y)
               (setq fact (list (caar thens)
                                (cadar thens)
                                (caddar then
s)))
               (cond ((ask_y fact)
                      (print 'ask_y_returns_
T)
                      (setq success T))))
              (T (verb new_fact cfvalue flag
value)))
        (cond ((and (equal verb 'write)
                    (equal cfvalue '1))
              (setq success T))
              (print success)
              (setq thens (cdr thens))
              (go loop)))

(defun testif (rule)
  (prog (ifs predicate fact)
        (print 'in_testif)
        (print rule)
        (setq ifs (cdaddr rule))
        (setq ckcf '1)
        loop
        (setq predicate (caar ifs))
        (cond ((null ifs) (return T))
              ((predicate (car ifs))
               (T (return NIL)))
              (setq ifs (cdr ifs))
              (go loop)))

(defun testwhen (rule)
  (prog (whens predicate fact)
        (setq whens (cdaddr rule))

```



```

        (setq ckcf '1)
    loop
    (setq predicate (caar whens))
    (cond ((null whens) (return T))
          ((predicate (car whens))
           (T (return NIL)))
          (T (return NIL)))
    (setq whens (cdr whens))
    (go loop)))

(defun usethen (rule)
  (prog (thens success verb cfvalue flagvalu
e new_fact)
    (setq thens (cdr (caddr rule)))
    (print 'in_usethen)
    loop
    (setq verb (caar thens))
    (cond ((equal verb 'ask_y)
           (setq cfvalue (car (cdddar th
ens))))
          (T (setq cfvalue (caddar thens
))))
    (setq flagvalue 'system_inferred)
    (setq new_fact (cadar thens))
    (cond ((null thens) (return success)
          ((equal verb 'ask_y)
           (setq fact (list (caar thens)
                             (cadar thens)
                             (caddar then
s))))
          (cond ((ask_y fact)
                  (print 'ask_y_returned
                        (setq success T))))
          (T (verb new_fact cfvalue flag
value)))
    (cond ((and (equal verb 'write)
                 (equal cfvalue '1))
           (setq success T)))
    (deduce)
    (print success)
    (setq thens (cdr thens))
    (go loop)))

(defun tryruleif (rule)
  (prog (ruletype)
    (print 'in_tryruleif)
    (setq ruletype (caaddr rule))
    (cond ((and (equal ruletype 'if) (testif
rule))
           (writerip rule 1)

```

```

        (usewhen rule)
        (setq rulesused (cons rule rulesused
ed))

        (return T))
        (T (return NIL))))))

(defun tryrulewhen (rule)
  (prog (ruletype)
    (setq ruletype (caaddr rule))
    (cond ((and (equal ruletype 'when) (testw
hen rule))

      (writerip rule 1)
      (usewhenforward rule)
      (setq rulesused (cons rule rulesused
))

      (return T))
      (T (return NIL))))))

(defun stepforward ()
  (prog (rulelist)
    (setq rulelist rules)
    loop
    (cond ((is_truerip (car rulelist)) (go s
kip)))

    (cond ((null rulelist) (go exit)))
    (cond ((tryruleif (car rulelist)) (retur
n T)))

    skip
    (setq rulelist (cdr rulelist))
    (go loop)
    exit
    (return NIL)))

(defun deduce ()
  (prog (progress ckcf)
    (print 'in_deduce)
    loop
    (cond ((stepforward) (setq progress T))
      (T (return progress)))
    (go loop)))

(defun testif+ (rule)
  (prog (ifs predicate fact)
    (setq ifs (cdaddr rule))
    (setq ckcf '1)
    loop
    (print 'in_testif+)
    (setq predicate (caar ifs))
    (cond ((null ifs) (return T))
      ((verify (cadar ifs))
      (T (return NIL)))
    (setq ifs (cdr ifs))

```

```

        (go loop)))

(defun tryruleif+ (rule)
  (prog (ruletype)
    (print 'in_tryruleif+)
    (setq ruletype (caaddr rule))
    (cond ((and (equal ruletype 'if) (testif
+ rule))
          (writerip rule 1)
          (cond ((usethen rule)
                (clearrip rule 0)
                (setq rulesused (cons rule
rulesused))
                (return T))
              (T (return NIL))))))

(defun verify (fact)
  (prog (relevant1 relevant cfval flagval new_f
act ckcf)
    (print 'in_verify)
    (print fact)
    (setq fact (list 'predicate fact 'comme
nt))
    (cond ((is_true fact) (return T)))
    (setq fact (cadr fact))
    (setq relevant1 (inthen fact))
    (setq relevant relevant1)
    (print relevant)
    loop
    (cond ((null relevant1)
          (cond ((member fact facts) (ret
urn NIL))
              ((and (print " ")
                    (print " ")
                    (print "Is this tr
ue :"))
              (print fact)
              (setq cfval (read))
              (equal cfval '1))
              (setq flagval 'user_suppl
ied)
              (setq new_fact fact)
              (setq asked (cons fact as
ked))
              (write new_fact cfval fla
gval)
              /*
              (deduce)
              (return T))
              ((equal cfval '-1)
              (setq flagval 'user_suppli
ed)
              (setq new_fact fact)

```

```

                                (setq asked (cons fact ask
ed))
                                (write new_fact cfval flag
val)
                                (return NIL))
                                ((equal cfval '0)
ed)
                                (setq flagval 'user-suppli
ed))
                                (setq new_fact fact)
                                (setq asked (cons fact ask
val)
                                (write new_fact cfval flag
                                (return NIL))
                                ((equal cfval 'WHY)
                                (why fact)
                                (go loop))
sked))
                                (T (setq asked (cons fact a
                                (return NIL))))))
loop1
(cond ((null relevant1) (go loop))
      ((equal (prescreen relevant1) 'NIL)
(go exit))
      ((tryruleif (car relevant1)) (retur
n T)))
(print 'in_loop1_verify)
(setq relevant1 (cdr relevant1))
(go loop1)
loop
(cond ((null relevant) (go exit))
      ((and (writerip (car relevant) 1)
            (tryruleif+ (car relevant)))
(return T)))
(setq relevant (cdr relevant))
(go loop)
exit
(return NIL)))

(defun inthen (fact)
  (prog (itrules oprules)
    (setq itrules rules)
    (setq oprules '())
    itloop
    (cond ((null itrules) (return oprules))
          ((thenp fact (car itrules))
           (setq oprules (cons (car itrules)
oprules))))
    (setq itrules (cdr itrules))
    (go itloop)))

(defun prescreen (rule)

```

```

(prog (ifs predicate)
  (setq ifs (cdaddr rule))
  loop
  (setq predicate (caar ifs))
  (cond ((null ifs) (return T))
        ((member rule rules_in_progress) (
return T))
        ((lessp (get '(cadar ifs) 'cf) thr
eshold)
         (return NIL)))
  (setq ifs (cdr ifs))
  (go loop)))

```

```

(defun thenp (fact rule)
  (prog (consequents)
    (setq thens (cdr (caddr rule)))
    loop
    (setq verb (caar thens))
    (cond ((or (equal 'write verb) (equal 'a
sk_y verb))
           (setq consequents (list (cadar th
ens))))))
    (setq thens (cdr thens))
    (cond ((null thens) (go exit)))
    (go loop)
  exit
  (setq actfact (cadr fact))
  (cond ((member actfact consequents) (ret
urn T))))))

```

```

(defun diagnose ()
  (prog (possibilities asked)
    (setq possibilities hypotheses)
    loop
    (cond ((null possibilities)
           (print "No hypothesis can be con
firmed.")
           (showrulesused)(tracerules)
           (return NIL))
          ((verify (car possibilities))
           (PRIN1 "Hypothesis")
           (PRIN1 (car possibilities))
           (PRIN1 "is true.")
           (terpri)
           (showrulesused)(tracerules)
           (return (car possibilities))))
    (setq possibilities (cdr possibilities))
    (go loop)))

```

```

(defun showrulesused ()
  (prog (x y z)

```

```

(setq x rulesused)
(setq y '() )
loop1
(cond ((null x) (go loop))
      ((member (cadar x) y)
       (putprop (cadar x) (add '1 (get (cadar x)
                                     (T (setq y (cons (cadar x) y))
                                     (putprop (cadar x) '1 'no)))
              (setq x (cdr x))
              (go loop1)
              loop
              (setq z y)
              (print ' " " )
              (print ' " " )
              (print ' " RULES - NO. OF TIMES USED
              (print ' " " )
              loop
              (cond ((null z) (return T)))
              (princ (car z))
              (princ ': )
              (princ (get (car z) 'no))
              (print ' " " )
              (setq z (cdr z))
              (go loop)))
) 'no)) 'no))

```

```

(defun tracerules ()
  (prog (x y)
    (print ' " " )
    (print ' " " )
    (print ' " TRACE OF RULES TRIED " )
    (print ' " " )
    (setq y '() )
    (setq x rules_in_progress)
    loop1
    (cond ((null x) (go loop)))
    (setq y (cons (cadar x) y))
    (setq x (cdr x))
    (go loop1)
    loop
    (cond ((null y) (return T)))
    (print (car y))
    (setq y (cdr y))
    (go loop)))

```

```

(defun why (fact)
  (prog (possibilities success)
    (setq possibilities rulesused)
    loop
    (cond ((null possibilities)
           (cond (success (return T))

```

```

((is_true fact)
 (PRIN1 fact)
 (PRIN1 "was hypothesis.")
 (terpri)
 (return T))
(T (PRIN1 fact)
 (PRIN1 "is not establis
hed.")
 (terpri)
 (return NIL))))
((ifp fact (car possibilities))
 (setq success T)
 (PRIN1 fact) (PRIN1 "needed to
show:")
 (terpri)
 (mapcar '(lambda (a) (p a))
 (cdr (caddr (car possib
ilities))))))
(setq possibilities (cdr possibilities))
(go loop)))

(defun ifp (fact rule)
 (member fact (caddr rule)))

(defun inif (fact)
 (mapcan '(lambda (r)
 (cond ((ifp fact r)
 (list r))))
 rules))

(defun usedp (rule)
 (prog (possibilities)
 (setq possibilities rulesused)
 loop
 (cond ((null possibilities)(return NIL))
 ((equal rule (cadar possibilities)
 (return T)))
 (setq possibilities (cdr possibilities))
 (go loop)))

)

(defun how (fact)
 (prog (possibilities success cfval flagval)
 (setq possibilities rulesused)
 loop
 (cond ((null possibilities)
 (cond (success (return T))
 ((is_true fact)
 (setq cfval (getcf fact))
 (setq flagval (getflag fac
t))
 (PRIN1 fact)

```

```

(PRIN1 "was")
(PRIN1 flagval)
(PRIN1 "with certainty fa
ctor")
(PRIN1 cfval)
(terpri)
(return T)
((is_false fact) (setq succ
ess T)
(setq cfval (getcf fact))
(setq flagval (getflag fac
t))
(PRIN1 fact)
(PRIN1 "was")
(PRIN1 flagval)
(PRIN1 "with certainty fa
ctor")
(PRIN1 cfval)
(terpri)
(return T))
(T (PRIN1 fact)
(PRIN1 "is not establis
hed")
(return NIL))))
(setq possibilities (cdr possibilities))
(go loop)))

(defun attach (c p)
  (putprop c p 'parent)
  (putprop p (append (get p 'children) (list c))
'children))

(defun startlist (fact cfvalue flagvalue)
  ((lambda (parent child)
    (attach child parent)
    (putprop child cfvalue 'cf)
    (putprop child fact 'fact)
    (putprop child flagvalue 'flag))
'root
(gensym)))

(defun buildlist (fact cfvalue flagvalue)
  (cond ((member fact facts)
    (prog (queue progeny)
      (setq queue (list 'root))
      tryagain
      (cond ((null queue) (go exit))
            ((equal fact (get (car queue
e) 'fact))
            (putprop (car queue) cfval
ue 'cf)
            (putprop (car queue) flagv

```



```

value 'flag)
                                (return T)))
children))
                                (setq progeny (get (car queue) 'c
))
                                (setq queue (cdr queue))
                                (setq queue (append progeny queue
))
                                (go tryagain)
                                exit
                                (print "Error in building the fa
ct list.")
                                (return NIL)))
                                (T (prog (queue next last)
                                (setq queue (list 'root))
                                loop
                                (cond ((null queue) (go expand)))
                                (setq next (get (car queue) 'chil
dren))
                                (setq last queue)
                                (setq queue (cdr queue))
                                (cond ((null next) (go expand)))
                                (setq queue (append next queue))
                                (go loop)
                                expand
                                ((lambda (parent child)
                                (attach child parent)
                                (putprop child cfvalue '
cf)
                                (putprop child fact 'fac
t)
                                (putprop child flagvalue
'flag))
                                (car last)
                                (gensym)) ))))

(defun write (new cfvalue flagvalue)
  (cond ((null facts) (startlist new cfvalue fla
gvalue))
        (T (builddlist new cfvalue flagvalue)))
  (cond ((member new facts) )
  (T (setq facts (cons new facts)) ) )

(defun clear (fact cfavluue flagvalue)
  (cond ((setq cfval '0)
        (setq flagval 'system_inferred)
        (write fact cfval flagval))
        (T ( return NIL))))

(defun getcf (fact)
  (cond ((member fact facts)
        (prog (queue progeny)
              (setq queue (list 'root))

```

```

                                loop
                                (cond ((null queue) (go exit))
                                        ((equal fact (get (car queue) '
e) 'fact))
                                        (return (get (car queue) '
cf))))
                                (setq progeny (get (car queue) 'c
hildren))
                                (setq queue (cdr queue))
                                (setq queue (append progeny queue
))
                                (go loop)
                                exit
                                (return '0)))
                                (T (return '0)))

(defun getflag (fact)
  (cond ((member fact facts)
        (prog (queue progeny)
              (setq queue (list 'root))
              loop
              (cond ((null queue) (go exit))
                    ((equal fact (get (car queue) '
e) 'fact))
                    (return (get (car queue) '
flag))))
              (setq progeny (get (car queue) 'c
hildren))
              (setq queue (cdr queue))
              (setq queue (append progeny queue
))
              (go loop)
              exit
              (return NIL)))
        (T (return NIL))))

(defun is_true (fact)
  (prog ()
    (setq fact (cadr fact))
    (cond ((member fact facts) (go checkcf))
          (T (go exit)))
    checkcf
    (cond ((equal (getcf fact)) (return T))
          (T (go exit)))
    exit
    (return NIL)))

(defun is_false (fact)
  (prog ()
    (setq fact1 (cadr fact))
    (cond ((member fact1 facts) (go checkcf))
          (T (go exit)))
    exit
    (return NIL)))
)

```

```

                (T (go exit)))
checkcf
    (cond ((equal -1 (getcf fact1)) (return
T))
                (T (go exit)))
exit
    (return NIL)))

(defun list_facts ()
  (prog (queue progeny)
    (setq queue (list 'root))
    (setq progeny (get (car queue) 'children
))
    (setq queue (cdr queue))
    (setq queue (append progeny queue))
loop
    (cond ((null queue) (return T))
          (T (print "  ")
              (print "  ")
              (PRIN1 "FACT    ")
              (PRIN1 (get (car queue) 'fact))
              (terpri)
              (print "  ")
              (PRIN1 "      CF  ")
              (PRIN1 (get (car queue) 'cf))
              (terpri)
              (print "  ")
              (PRIN1 "      SOURCE ")
              (PRIN1 (get (car queue) 'flag))
))
          (terpri)
          (print "  ")
n))
    (setq progeny (get (car queue) 'childre

    (setq queue (cdr queue))
    (setq queue (append progeny queue))
    (go loop)))

(defun startriplist (rule cfvalue)
  ((lambda (parent child)
    (attach child parent)
    (putprop child cfvalue 'cf)
    (putprop child rule 'rule))
   'riproot
  (gensym)))

(defun buildriplist (rule cfvalue)
  (cond ((member rule rules_in_progress)
        (prog (queue progeny)
          (setq queue (list 'riproot))
          tryagain
          (cond ((null queue) (go exit))

```

```

                                ((equal rule (get (car queue
e) 'rule))
                                (putprop (car queue) cfval
ue 'cf)
                                (return T)))
                                (setq progeny (get (car queue) 'c
hildren))
                                (setq queue (cdr queue))
                                (setq queue (append progeny queue
))
                                (go tryagain)
                                exit
                                (return NIL)))
(T (prog (queue next last)
      (setq queue (list 'riproot))
      loop
      (cond ((null queue) (go expand)
            (setq next (get (car queue) 'ch
ildren))
            (setq last queue)
            (setq queue (cdr queue))
            (cond ((null next) (go expand))
                  (setq queue (append next queue)
                    (go loop)
                    expand
                    ((lambda (parent child)
                      (attach child parent)
                      (putprop child cfvalue
                                (putprop child rule 'r
ule))
                                (car last)
                                (gensym)))))))

```

```

(defun writerip (rule cfvalue)
  (cond ((null rules_in_progress)
        (startriplist rule cfvalue))
        (T (buildriplist rule cfvalue)))
  (cond ((member rule rules_in_progress)
        (T (setq rules_in_progress (cons rule
e rules_in_progres
)) ) ) )

```

```

(defun clearrip (rule cfvalue)
  (cond ((setq cfval '0) (writerip rule cfva
l))
        (T (return NIL)))

```

```

(defun getcfrip (rule)
  (cond ((member rule rules_in_progress)

```

```

        (prog (queue progeny)
              (setq queue (list 'riproct))
              loop
              (cond ((null queue) (go exit))
                    ((equal rule (get (car queue)
                                       (return (get (car queue) '
e) 'rule))
                    (return (get (car queue) '
cf))))
              (setq progeny (get (car queue) 'c
hildren))
              (setq queue (cdr queue))
              (setq queue (append progeny queue
))
              (go loop)
              exit
              (return '99)))
        (T (return '99)))

    (defun is_truerip (rule)
      (prog ()
        (cond ((member rule rules_in_progress) (
go checkcf))
              (T (go exit)))
        checkcf
        (cond ((equal 1 (getcfrip rule)) (return
T))
              (T (go exit)))
        exit
        (return NIL)))

    (defun display (fact cbfvalue flagvalue)
      (prog ()
        (print ' " ")
        (print ' " ")
        (print fact)
        (print ' " ")
        (print ' " ENTER ANY CHARACTER TO CONTINU
E")
        (setq don't_care (read))
        (return T)))

    (defun ask_y (fact)
      (prog ()
        (print 'in_ask_y)
        (cond ((member fact facts)
              (cond ((equal 'user_supplied (get
flag fact))
                    (return (getcf fact))))))
        (setq question (cadr fact))
        (print ' ". ")
        (print ' " ")
        (print ' "Is this true : ")

```

```

(print question)
(setq cfvalue (read))
(setq flagvalue 'user_supplied)
(setq fact1 (cadr fact))
(write fact1 cfvalue flagvalue)
(setq asked (cons (cadr fact) asked))
(print asked)
(writerip rule 1)
  (deduce)
(cond ((greaterp cfvalue threshold) (ret
urn T))
      (T (return NIL))) )

(defun jump (fact cfvalue flagvalue)
  (prog ()
    (print '*      *')
    (print "WE ARE MOVING TO RULE BASE")
    (print fact)
    fact
    (return T)))

(setq rulesused ())
(setq facts ())
(setq rules_in_progress () )
(setq threshold '0)
(setq asked ())

```

Attachment B

Transformation Invariant  
Attributes for Digitized  
Object Outlines

IPL-TR-068

April 1985

by

B. G. Nickerson

## ABSTRACT

The objective of this project is to develop techniques for classifying objects in aerial and satellite imagery; e.g. aircraft on the ground, motor vehicles, oil storage tanks, ships, large buildings, and bridges. Recognition of these objects is based on their boundary (shape) only. Two distinct approaches were investigated, one based on Fourier descriptors and one based on invariant moments. A set of programs which compute and analyze these scale-, translation-, and rotation-invariant attributes of object outlines is described. The Fourier descriptor and invariant moments methods were applied to test data representing 21 digitized aircraft outlines to give a list of invariant attributes for each aircraft. A fuzzy clustering analysis of these invariant attributes was made to see if aircraft of similar shape clustered together based on the invariant attribute data. Two irregular blob shapes were analyzed to see the effect of random and systematic noise on their invariant attributes. It is shown that the area, perimeter, invariant moments and Fourier descriptors are useful in uniquely describing individual object shapes.



## ACKNOWLEDGEMENT

The research described here was supported by the Rome Air Development Center under contract number F30602-85-C-0008, and subcontract purchase order number 353-9023-7 via Syracuse University.

## TABLE OF CONTENTS

Abstract.....	598
Acknowledgement.....	599
List of Tables.....	601
List of Illustrations.....	603
I. Introduction.....	605
II. Fourier Descriptions.....	608
III. Invariant Moments.....	610
IV. Computer Implementation.....	613
A. Digitizing.....	613
B. Mask Description.....	614
C. Chain Code Description.....	617
D. Attribute Calculation.....	618
V. Summary of Results and Conclusions.....	619
References.....	622
Appendix of Results.....	623
A. Comparison of Attributes Under Transformation.....	623
B. Fuzzy Clustering Results.....	624
C. Analysis of Basic Shapes.....	625
D. Analysis of Blobs With and Without Noise.....	628
Tables.....	629
Figures.....	667

## LIST OF TABLES

- Table 1. Digitized data file DC9.DIG for McDonnell Douglas DC-9
- Table 2. Format of the DC-9 mask data file
- Table 3. Example of the attribute data file format
- Table 4. Effect of scale and rotation changes on a digitized rectangle
- Table 5. Complete attributes list for all seven rectangle transformations
- Table 6. Effect of scale and rotation changes on a digitized Boeing 727
- Table 7. Complete attributes list for all seven Boeing 727 transformations
- Table 8. The invariant attributes for all 21 aircraft
- Table 9. Partition coefficients of the cluster analysis
- Table 10. Clustering results for all 16 attributes
- Table 11. Clustering results for area and perimeter
- Table 12. Clustering results for first four moments
- Table 13. Clustering results for all seven moments
- Table 14. Clustering results for all seven Fourier descriptors
- Table 15. Invariant attributes of the four basic shapes
- Table 16. Fuzzy clustering results for all four basic shapes; invariant moments and Fourier descriptors
- Table 17. Fuzzy clustering results for all four basic shapes; invariant moments only
- Table 18. Fuzzy clustering results for all four basic shapes; Fourier descriptors only
- Table 19. Fuzzy clustering results for circles and squares; invariant moments and Fourier descriptors
- Table 20. Fuzzy clustering results for circles and squares; invariant moments only
- Table 21. Fuzzy clustering results for circles and squares; Fourier descriptors only

Table 22. The input data for blob1

Table 23. The input data for blob2

Table 24. The input data for blob3

Table 25. The input data for blob4

Table 26. A comparison of the invariant attributes of the  
noisy blobs to the non-noisy blob1

Table 27. A comparison of the invariant attributes of the  
noisy blob2s to the non-noisy blob2

Table 28. A comparison of the invariant attributes of  
blob3 to blob1

Table 29. A comparison of the invariant attributes of  
blob4 to blob2

## LIST OF ILLUSTRATIONS

- Figure 1. The complex function description of a closed boundary
- Figure 2. Example original object outline
- Figure 3. Mask description of an object
- Figure 4. The 8 direction chain code
- Figure 5. Format of the chain code data file
- Figure 6. Flowchart of one computer run to generate an object's attributes
- Figure 7. Six transformations of a rectangle
- Figure 8. Six transformations of a digitized Boeing 727 outline
- Figure 9. Plot of the 512 by 512 pixel space representation of the Boeing 727
- Figure 10. Boeing 727-200 (Boe727)
- Figure 11. Boeing 747-200B (Boe747)
- Figure 12. Boeing 757-200 (Boe757)
- Figure 13. Mikoyan MiG-25 (MiG-25)
- Figure 14. Cessna Citation III (Ccit3)
- Figure 15. Northrop F-5E Tiger II (F-5E)
- Figure 16. Tupolev Tu-28P (Tu-28P)
- Figure 17. McDonnell Douglas DC-10 (DC-10)
- Figure 18. McDonnell Douglas DC-9 (DC-9)
- Figure 19. McDonnell Douglas F-15C Eagle (F-15C)
- Figure 20. Lockheed P-3C Orion (LP-3C)
- Figure 21. Grumman A6-E/TRAM (A6-E)
- Figure 22. General Dynamics F-16XL (F-16XL)
- Figure 23. de Havilland DHC-6 Twin Otter (DHC-6)
- Figure 24. Cessna Skylane RG (CRG)
- Figure 25. Cessna 402C (C402C)

- Figure 26. Lockheed C-5B Galaxy (C-5B)
- Figure 27. Piper Chieftain (Piper)
- Figure 28. Rockwell International B-1B (B-1B)
- Figure 29. Lockheed C-130E Hercules (Herc)
- Figure 30. de Havilland DHC-7 Dash 7
- Figure 31. Basic shapes used to test the fuzzy clustering of invariant attributes
- Figure 32. Blob1 with random noise added
- Figure 33. Blob2 with random noise added
- Figure 34. Blob3 and blob4 (slightly changed versions of blob1 and blob2)

## I. INTRODUCTION

Black and white aerial photographs taken with optical aerial cameras have long been and still are the primary source of aerial reconnaissance information. Considering that the resolution of aerial film is at least 50 lines per mm (with some films capable of up to 800 lines per mm) and that each photograph measures 230 by 230 mm, it is obvious why aerial photographs are such a valuable source of information.

The primary objective of this project is to automatically interpret aerial photographs as they would be interpreted (or exploited) by a human photointerpreter. Computer processing of aerial photographs first requires that they be digitized. The next step is to extract the edges from the digitized photograph. These edges are one of the primary information sources a human uses to distinguish various objects in the photograph. After extraction, the edges are represented as lists of  $(x,y)$  coordinate pairs. These lists can be closed (i.e. the first coordinate pair is the same as the last) or open. This report concentrates on recognizing objects based on their closed boundary representation.

Two methods for automatic recognition of objects based on their closed boundary description are considered here. Granlund [1972] and Zahn and Roskies [1972] both derived expressions for Fourier descriptors of closed boundary objects which are invariant under scale, rotation and

translation. Persoon and Fu [1977] reported good results using Fourier descriptors to perform character recognition and machine parts recognition. Wallace and Wintz [1960] obtained good results using Fourier descriptors to recognize three dimensional views of aircraft boundaries. A different method for computing a scale-, rotation-, and translation-invariant description of an object boundary is given by Hu [1962]. This method is based on invariant moments. Alt [1962] derives moments invariant under an affine transformation (i.e. they are invariant to translation, scale, and "stretching" and "squeezing" along the horizontal axis, but are not invariant under rotation). Dudani et al [1977] reported good results in automatic identification of aircraft using the invariant moments of Hu. This report describes the implementation and analysis of both the Fourier descriptors and invariant moments methods.

As a separate part of this project, an expert system inference engine [Bhaskar, 1985] was implemented with the end goal being automatically to interpret aerial photographs. The data provided by the invariant attributes discussed above is the basis for the fact base of this expert system. To facilitate the writing of rules such as "IF object is round, THEN object is oiltank", the shape description of objects must be presented in a more natural way than a list of Fourier descriptors and invariant moments. A pattern recognition technique called fuzzy clustering [Bezdek, 1981] was implemented to provide a



natural transition from the Fourier descriptor and invariant moment data to a more natural shape description language such as "this object is shaped like a circle (0.8)", where the number in brackets is a possibility measure in the range 0 to 1.

Section II describes the Fourier descriptor method. The invariant moments are discussed in Section III. Implementation of these two methods on the computer is discussed in Section IV. Section V contains a summary of the results presented in the Appendix along with a conclusion.

## II. FOURIER DESCRIPTORS

The closed boundary of a region is considered as a complex valued function  $z(l)$  of the arc length  $l$  (see Figure 1). Due to the finite number of sampling points, function  $z(l)$  is band-limited, and can be approximated by a Fourier series as

$$z(l) = \sum_{n=-N/2}^{N/2} C_n \exp(jnwl), \quad (1)$$

$$\text{where } C_n = (1/L) \int_0^L z(l) \exp(-jnwl) dl, \quad (2)$$

for  $N$  = number of contour points and  $w = 2\pi/L$ ,

and  $L$  = total length of the closed boundary. The complex Fourier coefficients can be represented in polar form as

$$C_n = |C_n| \exp(j\phi_n). \quad (3)$$

Fourier descriptors which are invariant to scale, rotation and translation (for a proof see Merkle and Lorch (1984)) are given as

$$D_n = \frac{|C_n|}{|C_1|} \exp\{j[\phi_n + (1-n)\phi_2 - (2-n)\phi_1]\}. \quad (4)$$

These descriptors can also be written in polar form as

$$D_n = |D_n| \exp(j\psi_n). \quad (5)$$

The actual evaluation of (2) for the Fourier coefficients requires that the object boundary be stored as a sequence of  $x, y$  coordinates. The discrete Fourier transform of the series  $z(l)$  is then taken to give the

Fourier coefficients (Wallace and Wintz, 1980) as

$$C_n = \sum_{k=0}^{N-1} z(k) \exp(-jnwk) , \quad (6)$$

where  $k$  = distance from the starting point.

### III. INVARIANT MOMENTS

The two dimensional (p+q)th order moments of a density distribution function  $f(x,y)$  are defined as

$$m_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^p y^q f(x,y) dx dy, \quad p,q = 0,1,2, \dots \quad (7)$$

For the discrete case,  $f(x,y)$  represents a closed boundary region, and the moments are computed as sums over the area within the boundary with  $f(x,y) = 1$ .

The central moments are defined as

$$u_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (x-\bar{x})^p (y-\bar{y})^q f(x,y) dx dy, \quad (8)$$

$$\text{where } \bar{x} = \frac{m_{10}}{m_{00}}, \quad \bar{y} = \frac{m_{01}}{m_{00}}$$

are the centroid coordinates for the region. These central moments are invariant under translation of the region's boundary coordinates.

Scale invariant moments are obtained by use of

$$v_{pq} = \frac{u_{pq}}{r}, \quad p,q = 2,3, \dots \quad (9)$$

where  $r = (p+q)/2 + 1$  (Hu, 1962).

Notice that  $r$  is a real number and can take on nonintegral values.

Eliminating the dependence on orientation results in the following 7 invariant moments:

$$w_1 = v_{20} + v_{02}, \quad (10.a)$$

$$w_2 = (v_{20} - v_{02})^2 + 4v_{11}^2, \quad (10.b)$$

$$w_3 = (v_{30} - 3v_{12})^2 + (3v_{21} - v_{03})^2, \quad (10.c)$$

$$w_4 = (v_{30} + v_{12})^2 + (v_{21} + v_{03})^2, \quad (10.d)$$

$$w_5 = (v_{30} - 3v_{12})(v_{30} + v_{12})[(v_{30} + v_{12})^2 - 3(v_{21} + v_{03})^2] + (3v_{21} - v_{03})(v_{21} + v_{03})^2 [3(v_{30} + v_{12})^2 - (v_{21} + v_{03})^2], \quad (10.e)$$

$$w_6 = (v_{20} - v_{02})[(v_{30} + v_{12})^2 - (v_{21} + v_{03})^2] + 4v_{11}(v_{30} + v_{12})(v_{21} + v_{03}), \quad (10.f)$$

$$w_7 = (3v_{21} - v_{03})(v_{30} + v_{12})[(v_{30} + v_{12})^2 - 3(v_{21} + v_{03})^2] - (v_{30} - v_{12})(v_{21} + v_{03}) [3(v_{30} + v_{12})^2 - (v_{21} + v_{03})^2]. \quad (10.g)$$

The proof of the rotation invariance of these quantities is found in Hu [1962]. Equations (10) are the translation, scale and rotation invariant moments.

An extension of these 7 invariant moments to include higher order moments has been done using the general invariant moment formulas given by Hu [1962]. A Macsyma

program was written to evaluate the general formulas and to  
output FORTRAN statements to implement them.

#### IV. COMPUTER IMPLEMENTATION

Three separate programs were written on the Image Processing Laboratory PRIME 750 computer. They are

1. DIGTIZ - digitize object outlines on the Talos digitizer
2. MAKMSK - generate the mask data and chain codes for the digitized object
3. MAKATT - compute the transformation invariant attributes of the object boundary

Some simple file naming conventions are followed. Each different file has the object name plus a unique identifier attached to the end. For example, for the DC-9 shown in Figure 2, the following files exist:

1. DC9.DIG = digitized data file (EMACS editable)
2. DC9.MSK = mask data file (EMACS editable)
3. DC9.CHN = chain code data file (not EMACS editable)
4. DC9.ATT = object attributes file (EMACS editable)

##### A. Digitizing

The Talos digitizer is a 0.001 inch resolution rectangular solid state tablet. After attaching an object outline to the digitizer, program DIGTIZ is run to enter the coordinates in a point by point mode. Figure 2 shows a typical aircraft outline which was digitized [Jane's, 1983]. The digitized result is shown in Figure 18. When digitizing the object outline, the following information is requested:

1. File name in which to store the digitized data.
2. Up to 80 characters giving the title of the object.
3. Two ends of a scale bar and its length in meters.

For aircraft, the scale bar information is commonly the wingspan. After digitizing the entire closed boundary, the file generated is in an editable form with a format as shown in the example in Table 1.

The digitized data can now be plotted on the Versatec plotter using program DIGPLT. The data is automatically scaled to fit on one page of output Versatec paper. A complete library of all 21 aircraft outlines which were digitized during the course of this project is contained in the illustrations.

#### B. Mask Description

Calculation of the invariant moments (as described in section III) requires a so-called mask description of the digitized object to give  $f(x,y)$ . Figure 3 depicts the mask description of a simple object.

Generating this mask description from the original digitized outline is done in a sequence of steps as follows:

1. Convert from digitizer to image coordinates (i.e. digitizer origin = lower left; image origin = upper left corner) by complementing the y coordinates.
2. Compute the Bresenham algorithm version [Foley and Van Dam, 1982] of the boundary so that the boundary consists of connected adjacent pixels. The result is a sequence of boundary points defined as



$(y_1, x_1), (y_2, x_2), \dots, (y_i, x_i), \dots$

3. Flag any horizontal tangent occurring in a down-up-down fashion; e.g.

	1	2	3	4	5	6	7	8	Here, pixel (4,4) will
4				x	x	x			be flagged by setting
5		x	x	-	-	x	x	x	the x value negative.

In the above example, pixel (4,4) becomes (4,-4).

4. Sort the boundary pixels in a 2D sort so that the lines are in ascending order with the elements in ascending order for each line. This list takes the form

$(y_1, x_{11}), (y_1, x_{12}), \dots, (y_1, x_{1j}), \dots$

$(y_2, x_{21}), (y_2, x_{22}), \dots, (y_2, x_{2j}), \dots$

⋮

$(y_i, x_{i1}), (y_i, x_{i2}), \dots, (y_i, x_{ij}), \dots$

5. Generate the mask description using the maskit algorithm.

This last step is the most difficult.

#### The maskit algorithm

This algorithm is used to generate the mask description of an object given a sorted list of boundary pixels with down-up-down horizontal tangent points flagged by setting the first element of the tangent line to the negative of the element. Subroutines MASKIT and MASKLI are used to perform this process. The algorithm proceeds as follows:

1. Scan the first row. All elements encountered here must belong to the object's mask. A bit map  $b_1$  is prepared for line 1 with 1's for every element inside the mask,

0's for every element outside the mask. For example,

$$b_1 = (0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ \dots \ 1 \ 0 \ 0 \ 0)$$

2. Increment line counter  $i$ . If  $i >$  no. of lines, stop. Otherwise, for line  $y_i$ , gather the  $m$  elements of one row together in a vector as follows:

$$e_i = (x_{i1}, x_{i2}, \dots, x_{ij}, \dots, x_{im}).$$

3. Initialize  $b_{i-1}$  to  $b_i$ , and set  $b_i = 0$ .

4. If  $m = 1$ , set the bit  $b_{i,x_{i1}} = 1$ , and go to step 6.

5. For each  $x_{ij}$ , then

- 5.1 If  $x_{ij} < 0$  a new tangent line is found.

If  $b_{i-1,x_{i-1,j}}$  = 1, then this is the beginning of a

new hole inside the mask. Increment  $k$ , and keep track of mask holes with array  $h_k$ , where

$$h_k = h_{ik}, \quad i=1,2,3, \quad k$$

$h_{1k}$  = present line for mask hole  $k$ ,

$h_{2k}$

= beginning column for mask hole  $k$ ,

$h_{3k}$

= ending column for mask hole  $k$ .

$h_{ik}$

- 5.2 If  $x_{i,j-1}$  is not connected to a previous mask hole, then check to see if  $x_{ij}$  is connected to a previous mask hole as follows:

$$\text{If } h_{1k} = y_{i-1} \text{ and } x_{ij} \geq (h_{2k} - 1) \text{ and } x_{ij} \leq (h_{3k} + 1),$$

then  $x_{ij}$  is attached to  $h_k$ ; set  $h_{2k} = h_{3k} = x_{ij}$ .

- 5.3 Set the bit position for  $x_{ij}$  and  $x_{i,j+1}$  on.

- 5.4 If  $x_{i,j-1}$  is connected to a previous mask hole, or  $x_{ij}$  is connected to a previous mask hole, then

propagate the mask hole to these new elements by reassigning  $h_{2k}$  and  $h_{3k}$  as required.

5.5 For  $x_{ij}$  and  $x_{i,j-1}$  not connected to a previous mask hole, check all bits in  $b_{i-1}$  from  $x_{i,j-1}$  to  $x_{ij}$ . If all bits are = 1, then add the elements from  $x_{i,j-1}$  to  $x_{ij}$  to  $b_i$ .

5.6 Increment  $j$ ; if  $j \leq m$ , go to 5.1.

6. Assign the mask elements for line  $i$  based on  $b_i$ ; i.e. for all contiguous bit strings in  $b_i$  for  $b_i = 1$ , assign  $m_k = m_{ik}$ ,  $i=1,2,3$  for  $m_{ik} = \text{line}$ ,  $m_{2k} = \text{beginning column}$ ,  $m_{3k} = \text{ending column}$ .

7. Increment  $i$ ; if  $i \leq \text{no. of lines in the object}$ , go to 2; otherwise, stop.

After finishing the algorithm, the mask description consists of a list of mask elements  $m_{ik}$ ,  $i=1,2,3$ ,  $k=1,n$  for  $n = \text{no. of mask elements in this mask description}$ . Table 2 shows how the file of mask data is stored.

### C. Chain Code Description

The standard 8 direction chain code (see Figure 4) is generated directly from the Bresenham version of the object boundary. Once computed, the data is stored in a separate file encoded 5 codes per 16 bit word. Figure 5 shows the exact format for storing the chain code. Subroutines PUTCOD and GETCOD perform the transformation to and from this storage format.

## D. Attribute Calculation

Using equations (4) and (10), the Fourier descriptors and moment invariants, respectively, are computed. The area is calculated in meters squared directly from the moments (as  $m_{00}$ ), and the perimeter is calculated in meters directly from the chain code data. After calculation, this data is stored in an editable file in the format indicated by the example in Table 3. For the Fourier descriptors, only the magnitudes are stored. The descriptors for  $n=0$  and  $n=1$  are not stored since they are scale and position dependent and do not contribute to the identification of an object. Seven Fourier coefficients for  $n = -4, -3, -2, -1, 2, 3, 4$  are computed and stored along with the other attributes. Subroutines DIGATT and FDESC of program MAKATT are the primary computation routines.

Attributes computed from closed boundary objects obtained from a digitized photograph will also include the centroid coordinates and orientation of the major axis. Program MAKATT2 was designed to compute attributes of objects from an actual digitized image. This program also includes the extended moments calculation mentioned in section III above.

## V. SUMMARY OF RESULTS AND CONCLUSIONS

To test the invariance of both the Fourier descriptors and moment invariants, the capability to both rotate and scale a digitized object was added to program MAKMSK. Figure 6 shows the standard flow of processing to make one "run" of the above programs after digitizing an object's outline. To facilitate this process, several small command processing language (CPL) programs were written.

A comparison of the invariant attributes for 6 different scales and orientations of 2 object's outlines is presented in the Appendix. The Fourier descriptors seem to be more sensitive to scale and orientation than do the invariant moments. The higher order moments (i.e.  $w_4$ ,  $w_5$ ,  $w_6$ , and  $w_7$ ) are inconsistent also (see Table 7).

A fuzzy isodata clustering analysis of the invariant attributes for 20 aircraft outlines (see Figures 9 through 30) is presented in the Appendix. The full 16 member attributes vector gives the best clustering as to the aircraft's utility. Neither the area, perimeter or Fourier descriptor attributes are capable of detecting the "delta" wing aircraft typified by the F-16XL, whereas the moments do seem able to group them together. Program FISODAT contains the LISP source code used for the fuzzy isodata clustering.

A fuzzy clustering analysis of 4 basic shapes (rectangles, squares, triangles, and circles) is presented in the Appendix. With this method, it was hoped to enable

statements such as "this closed boundary is 0.3 like a rectangle and 0.6 like a triangle", which falls very naturally from the fuzzy clustering approach. Unfortunately, the method was not even able to distinguish squares from circles. It was determined that the problem is with the original invariant moments and Fourier data. This data is not similar for similarly shaped objects.

The Appendix presents an analysis of the invariant moments and Fourier descriptors for several "blob" shapes. A successively larger amount of noise was added to the outlines of 2 blob shapes. The analysis shows that the invariant moments are, on average, more susceptible to noise than are the Fourier descriptors. In addition, slight systematic variations in the outlines of the 2 original blobs were made to see how the invariant moments and Fourier descriptors reacted to them. The Fourier descriptors were again less sensitive to the systematic noise.

In summary, both the invariant moments and Fourier descriptors seem to be invariant under scale and rotation changes for simple objects such as the rectangle. For more complex shapes, the first 3 invariant moments seem to be more stable than the Fourier descriptors. The best clustering of the aircraft is obtained using all 16 of the attributes. Clustering of the basic shapes did not work due to the inconsistency of the Fourier descriptor and invariant moment data. The blob analysis showed that the Fourier descriptors used here are less sensitive to both

random and systematic noise.

The fuzzy clustering analysis of the attribute data is a good way to make decisions about an object's shape without having to consider the actual raw data values themselves. This is particularly true with an expert system where one wishes to write rules which are easy to interpret and maintain. Neither the invariant moments nor the Fourier descriptors used here seem to be the ideal raw data values on which to base the decision about shape. Along with such measures as area and perimeter they do provide some help as shown by the fuzzy clustering of the aircraft outlines presented in the Appendix.

## REFERENCES

- F.L. Alt, "Digital Pattern Recognition by Moments", Journal of the ACM, Vol.9, pp.240-258 (1962).
- J.C. Bezdek, Pattern Recognition with Fuzzy Objective Function Algorithms, Plenum Press, New York (1981).
- P.S. Bhaskar, Design of an Inference Engine for an Image Interpretation Expert System, RPI Image Processing Laboratory Technical Report IPL-TR-067 (1985).
- S.A. Dudani, K.J. Breeding, and R.B. McGhee, "Aircraft Identification by Moment Invariants", IEEE Transactions on Computers, Vol.C-26, No.1, pp.39-46 (1977).
- J.D. Foley and A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Company, Reading, Massachusetts (1982).
- G.H. Granlund, "Fourier Preprocessing for Hand Print Character Recognition", IEEE Transactions on Computers, Vol.C-21, pp.195-201 (1972).
- M.K. Hu, "Visual Pattern Recognition by Moment Invariants", IRE Transactions on Information Theory, Vol.IT-8, pp.179-187 (1962).
- Jane's All The World's Aircraft. Jane's Publishing Company Limited, London, England (1983).
- F. Merkle and T. Lorch, "Hybrid Optical-Digital Pattern Recognition", Applied Optics, Vol.23, No.10, pp.1509-1516 (1984).
- B.G. Nickerson, Fuzzy Isodata Clustering of Aircraft Outline Features, Final project report for course 35.6714 Fuzzy Sets and Expert Systems, Rensselaer Polytechnic Institute, Troy, NY (1984).
- E. Persoon and K.S. Fu, "Shape Discrimination Using Fourier Descriptors", IEEE Transactions on Systems, Man, and Cybernetics, Vol.SMC-7, No.3, pp.170-179 (1977).
- T.P. Wallace and P.A. Wintz, "An Efficient Three-Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors", Computer Graphics and Image Processing 13, pp.99-126 (1980).
- C.T. Zahn and R.Z. Roskies, "Fourier Descriptors for Plane Closed Curves", IEEE Transactions on Computers, Vol.C-21, No.3, pp.269-281 (1972).



## APPENDIX OF RESULTS

### A. Comparison of Attributes Under Transformation

Figure 7 contains 6 plots of the outline of a rectangle whose original digitized coordinates are (5750,4959), (8793,4959), (8793,3952), and (5750,3952). The rectangle has been scaled to fit into 2 different pixel spaces of 128 by 128 and 256 by 256 pixels, with rotation angles of 0, 22, and 45 degrees. Table 4 contains a list of 7 attributes (area, perimeter,  $w_1$ ,  $w_2$ ,  $|D_{-3}|$ ,  $|D_{-1}|$ , and  $|D_3|$ ) for each of these 6 rectangles. These 7 attributes were also computed for a rectangle in a pixel space of 512 by 512 with a rotation of 0 degrees. The percent differences between the attributes for this 512 by 512 pixel space rectangle and the above mentioned 6 rectangles were computed (see Table 4). The overall average difference was 1.42%, while the average difference for the 128 by 128 pixel space was 2.15%. For the 256 by 256 pixel space, the average difference was 0.70%.

A similar analysis was performed on 6 different transformations of the outline of a Boeing 727 (see Figure 8). Table 6 contains a comparison of the 10 attributes computed for them. Table 7 contains a list of all the actual invariant attributes for each case. The 3 invariant moments used in the comparison have an average difference of 4.2% while the Fourier descriptors used have an average difference of 59.0%. The Fourier descriptors are far more

sensitive to scale and rotation changes than the invariant moments. Notice from Table 7 that the invariant moments  $w_1$ ,  $w_5$ ,  $w_6$ , and  $w_7$  have a very small magnitude. They fluctuate widely among the different scales and rotations used. These would not be good choices for transformation invariant attributes.

### B. Fuzzy Clustering Results

To further investigate the usefulness of invariant moments and Fourier descriptors, a fuzzy isodata clustering analysis [Nickerson, 1984; Bezdek, 1981] of the first 20 aircraft shown in Figures 9 through 30 was made. The number of cluster centers was chosen as 2, 3, 4, and 5 for each of

- A. The full 16 member attributes vector,
- B. The area and perimeter attributes only,
- C. The first 4 invariant moment attributes,
- D. All 7 invariant moment attributes,
- E. All 7 Fourier descriptors.

Table 8 contains the input data values used for the fuzzy cluster analysis of the aircraft invariant attributes. The LISP program FISODAT in Appendix 2 contains the software used to make the analysis. Table 9 lists the partition coefficient for each of these cluster analyses. The closer a partition coefficient is to 1, the more closely it

resembles a "hard" clustering, and the better separated are the cluster centers.

Tables 10 through 14 list the aircraft clusters for each of the above attribute vectors and number of cluster centers. The abbreviations used for each aircraft are defined in the list of illustrations. The number between 0 and 1 following each aircraft (e.g. .73 in Tu-28P.73) is the largest element of its membership distribution in the cluster center space. For each value of  $c$ , the aircraft are arranged in descending order of the magnitude of this largest element of their membership distribution.

The full 16 member attributes vector gives the best clustering as to the aircraft's utility. Neither the area, perimeter or Fourier descriptors attributes seem capable of detecting the "delta" wing aircraft typified by the F-16XL, whereas the moments do seem able to group them together.

#### C. Analysis of Basic Shapes

Figure 31 shows 20 different objects defining 4 basic shapes (rectangles, squares, triangles, and circles) which were used to test the fuzzy clustering of invariant attributes. The figures were digitized as explained in section IV.A. Table 15 lists the invariant moments and Fourier descriptors used to describe the 20 object outlines of Figure 31. Tables 16, 17, and 18 summarize the fuzzy clustering results of these 4 basic shapes.

Table 16 shows the results for both invariant moments and Fourier descriptors considered simultaneously. There is no evident clustering of the 4 basic shapes with one another. Two of the circles are clustered with the squares and 4 of the rectangles in the second cluster. One triangle has established its own cluster center for cluster 3. Table 17 shows the results for the clustering of invariant moments only. Again, the circles, squares and 4 of the rectangles cluster together, with the seventh triangle establishing its own class. Table 18 contains the results for the Fourier descriptors only. The results here are slightly different, but again there is no clear clustering of the same shapes with one another. The rectangles cluster with the triangles, and 2 of the circles define their own cluster centers.

Tables 19, 20 and 21 show the results of fuzzy clustering with just the circles' and squares' invariant attributes. The first 4 rows are for the squares with the last 4 rows for the circles. Table 20 (invariant moments only) shows that one of the squares established its own cluster, whereas with Table 21 (Fourier descriptors), one of the circles attempts to establish its own class.

Why is the fuzzy clustering of these invariant attributes not able to distinguish between circles and squares? The answer lies in the shapes themselves.

For the invariant moments, by performing the integration of the equations of a square and a circle, one can determine theoretically what the invariant moments

should be. Of the 7 invariant moments used in the above analysis, only 1 should be nonzero for the square and circle. The first invariant moment should be 0.1667 for the square and 0.1592 for the circle. The remaining 6 invariant moments should theoretically be zero. Due to small digitizing errors and the fitting of each object into a 256 by 256 pixel space before computing the invariant attributes, the 6 invariant moments which should be zero are not. They represent the quantization noise.

The fuzzy clustering algorithm weights all of the invariant attributes equally by normalizing their value w.r.t. the largest attribute value. For example, in Table 15, the largest attribute value for the second invariant moment for all 20 shapes is 0.3016 for shape no. 16. All of the object's second invariant moments are divided by this value to obtain the numbers actually used in the fuzzy clustering. The end result is that numbers representing noise are being used to do the clustering, which simply does not work.

The reason why the Fourier descriptors do not work well may be due to the fact that the very sharp discontinuities at the corners of the squares are difficult to account for in the frequency domain. These sharp corners represent a very high frequency content, and we are here using only the 7 lowest order Fourier descriptors. Looking at the raw data in Table 15, one can see that the values of the Fourier descriptors for both the squares and circles are quite different. It is not clear why the

circles do not have consistent Fourier descriptors.

#### D. Analysis of Blobs With and Without Noise

Figures 32 and 33 show two blobs and their noisy versions used to test the sensitivity of both the invariant moments and the Fourier descriptors. Tables 26 and 27 give a comparison of the noisy and non-noisy blob data. In both cases, the Fourier descriptors varied less (on average) than did the invariant moments. The largest average percentage difference was 76.3% for the invariant moments of the noisiest case in Figure 32. The largest average percentage difference for the Fourier descriptors was 13.7%.

Figure 34 shows a third and fourth blob which are slightly different than the first and second blobs, respectively. These represent a systematic discrepancy between the shapes. Tables 28 and 29 compare blob 1 against blob 3 and blob 2 against blob 4, respectively. Again, the Fourier descriptors seem less disturbed by the systematic differences than do the invariant moments.

Tables 22 through 25 contain the original input data for all 4 blobs used above. The noisy blobs were obtained by adding a randomly distributed distance perpendicular to the blob outline in a positive direction only (i.e. on one side of the blob outline).

Table 1. Digitized data file DC9.DIG  
for McDonnell Douglas DC-9.

	1	2	3	4	5
12345678901234567890123456789012345678901234567890					
McDonnell Douglas DC-9 Super 80 'stretched' version					
	938.69	11.23			
5834	2678				
5888	2687				
5737	3135				
.	.				
.	.				
5835	2674				

- Notes: 1. Line 2 format = 2F12.2 = scale bar length in digitizer units and meters, respectively.  
2. Lines 3 to end format = I5,I1X,I5

Table 2. Format of the DC-9 mask data file.

	1	2	3	4	5
12345678901234567890123456789012345678901234567890					
McDonnell Douglas DC-9 Super 80 'stretched' version					
0.9185871E-01					
26	186	193			
27	185	194			
28	186	196			
29	187	198			
30	188	200			
31	188	201			
.	.				
.	.				
359	186	191			

- Notes: 1. Line 2 format = E15.7 = scale of the mask data in meters per pixel.  
2. Lines 3 to end format = 3I6 = line, beginning column and ending column, respectively.

Table 3. Example of the attribute data file format.

McDonnell Douglas DC-9 Super 80 'stretched' version  
246.64 175.04  
0.4726E+00 0.8529E-01 0.1143E-01 0.4054E-02  
0.2759E-04 0.1182E-02 0.4852E-06  
0.1881E+00 0.6259E+00 0.4973E+00 0.2968E+01  
0.7632E-01 0.1107E+01 0.5542E+00

- Notes:
1. Line 2 contains the area and perimeter in meters squared and meters, respectively in the format 2F12.2.
  2. Lines 3 and 4 contain the 7 invariant moments in format 4(E12.4,2X)/.
  3. Lines 5 and 6 contain 7 Fourier descriptor magnitudes in the same format as line 3; Index n = -4, -3, -2, -1, 2, 3, 4, resp.



Table 4. Effect of scale and rotation changes on a digitized rectangle.

Scale	Rot.	Area	Per.	Moments	Fourier	% Change
512	0	314.25	81.69	.2787 (1) .0499 (2)	.2674 (-3) 2.397 (-1) .1084 (3)	0 0 0 0 0 0
128	0	319.7	81.5	.2777 (1) .0494 (2)	.2667 (-3) 2.380 (-1) .1057 (3)	1.7 0.2 0.4 1.1 0.3 0.7 2.5
128	22	323.2	88.3	.2734 (1) .0470 (2)	.2664 (-3) 2.423 (-1) .1162 (3)	2.8 8.1 1.9 5.9 0.4 1.1 7.2
128	45	320.4	81.7	.2752 (1) .0480 (2)	.2684 (-3) 2.414 (-1) .1112 (1)	1.9 0.1 1.2 3.9 0.4 0.7 2.6
256	0	317.0	81.7	.2778 (1) .0494 (2)	.2675 (-3) 2.397 (-1) .1084 (3)	0.9 0.0 0.3 1.1 0.0 0.0 0.0
256	22	316.4	88.4	.2782 (1) .0496 (2)	.2675 (-3) 2.394 (-1) .1078 (3)	0.7 8.2 0.2 0.7 0.0 0.1 0.6
256	45	315.0	81.7	.2791 (1) .0501 (2)	.2672 (-3) 2.391 (-1) .1075 (1)	0.2 0.1 0.1 0.4 0.1 0.2 1.7

Table 5. Complete attributes list for  
all 7 rectangle transformations.

1. 512 by 512, 0 degrees			
Test rectangle with aspect ratio of 3:1			
314.25	81.69		
0.2787E+00	0.4992E-01	0.0000E+00	0.0000E+00
0.0000E+00	0.0000E+00	0.0000E+00	
0.9406E-12	0.2674E+00	0.2358E-11	0.2397E+01
0.1733E-11	0.1084E+00	0.8939E-12	
2. 128 by 128, 0 degrees			
Test rectangle with aspect ratio of 3:1			
319.70	81.52		
0.2777E+00	0.4938E-01	0.0000E+00	0.0000E+00
0.0000E+00	0.0000E+00	0.0000E+00	
0.2699E-12	0.2667E+00	0.6931E-12	0.2380E+01
0.5385E-12	0.1057E+00	0.2472E-12	
3. 128 by 128, 22 degrees			
Test rectangle with aspect ratio of 3:1			
323.18	88.31		
0.2734E+00	0.4698E-01	0.2631E-08	0.2924E-09
-0.2564E-18	-0.6337E-10	-0.2459E-20	
0.4548E-02	0.2664E+00	0.1697E-02	0.2423E+01
0.1842E-02	0.1161E+00	0.4244E-02	
4. 128 by 128, 45 degrees			
Test rectangle with aspect ratio of 3:1			
320.37	81.74		
0.2752E+00	0.4799E-01	0.7313E-08	0.7707E-09
-0.7442E-18	-0.1443E-09	-0.1671E-17	
0.2032E-12	0.2684E+00	0.4551E-12	0.2414E+01
0.4179E-12	0.1112E+00	0.2126E-12	
5. 256 by 256, 0 degrees			
Test rectangle with aspect ratio of 3:1			
316.99	81.70		
0.2778E+00	0.4938E-01	0.0000E+00	0.0000E+00
0.0000E+00	0.0000E+00	0.0000E+00	
0.5215E-12	0.2675E+00	0.1240E-11	0.2397E+01
0.9723E-12	0.1084E+00	0.4553E-12	
6. 256 by 256, 22 degrees			
Test rectangle with aspect ratio of 3:1			
316.43	88.40		
0.2782E+00	0.4959E-01	0.0000E+00	0.0000E+00
0.0000E+00	0.0000E+00	0.0000E+00	
0.5674E-04	0.2675E+00	0.1520E-03	0.2394E+01
0.1520E-03	0.1078E+00	0.5674E-04	
7. 256 by 256, 45 degrees			
Test rectangle with aspect ratio of 3:1			
314.98	81.74		
0.2791E+00	0.5010E-01	0.5104E-09	0.5519E-10
-0.3316E-20	-0.1029E-10	-0.8648E-20	
0.3902E-12	0.2672E+00	0.8955E-12	0.2391E+01
0.7508E-12	0.1075E+00	0.3719E-12	

Table 6. Effect of scale and rotation changes on a digitized Boeing 727.

Scale	Rot.	Area	Per.	Moments	Fourier	% Change
512	0	342.99				0
			194.24			0
				.4188 (1)		0
				.05142 (2)		0
				.00139 (3)		0
					.8041 (-3)	0
					2.953 (-1)	0
					.07683 (2)	0
					1.133 (3)	0
					.6468 (4)	0
128	0	369.88				7.8
			197.04			1.4
				.4014 (1)		4.2
				.04758 (2)		9.5
				.00150 (3)		7.8
					.8660 (-3)	7.7
					3.126 (-1)	5.8
					.1435 (2)	86.8
					1.229 (3)	8.5
					.6868 (4)	6.2
128	22	363.45				6.0
			201.46			3.7
				.4089 (1)		2.4
				.05043 (2)		1.9
				.00153 (3)		9.9
					.6146 (-3)	23.6
					3.124 (-1)	5.8
					.2321 (2)	202.1
					.9696 (3)	14.4
					.6077 (4)	6.0
128	45	361.52				5.4
			195.62			0.7
				.4040 (1)		3.5
				.04445 (2)		13.6
				.00147 (3)		6.0
					.4282 (-3)	46.7
					3.572 (-1)	21.0
					.4772 (2)	521.1
					.7914 (3)	30.2
					.6834 (4)	5.6

Table 6 continued on next page.

Table 6, continued.

Scale	Rot.	Area	Per.	Moments	Fourier	% Change
256	0	349.16				1.8
			195.71			0.2
				.4169 (1)		0.4
				.05163 (2)		0.4
				.00141 (3)		1.3
					.8062 (-3)	0.3
					2.951 (-1)	0.1
					.08350 (2)	8.7
					1.129 (3)	0.4
					.6585 (4)	1.8
256	22	354.26				3.3
			197.42			1.6
				.4102 (1)		2.0
				.05072 (2)		1.4
				.00131 (3)		5.5
					.5782 (-3)	26.1
					3.040 (-1)	2.9
					.2007 (2)	161.2
					.9158 (3)	19.2
					.5902 (4)	8.8
256	45	346.98				1.2
			194.63			0.2
				.4158 (1)		0.7
				.05014 (2)		2.5
				.00136 (3)		1.9
					.3884 (-3)	51.7
					3.420 (-1)	15.8
					.4166 (2)	442.2
					.7406 (3)	34.6
					.6387 (4)	1.2

Table 7. Complete attributes list for  
all 7 Boeing 727 transformations.

1. 512 by 512, 0 degrees				
Boeing 727-200 (Boe727)				
	342.99	194.24		
	0.4188E+00	0.5142E-01	0.1389E-02	0.1416E-05
	0.5394E-10	-0.2622E-07	-0.3213E-10	
	0.1371E+00	0.8041E+00	0.5359E+00	0.2953E+01
	0.7683E-01	0.1133E+01	0.6468E+00	
2. 128 by 128, 0 degrees				
Boeing 727-200 (Boe727)				
	369.88	197.04		
	0.4014E+00	0.4758E-01	0.1498E-02	0.9491E-05
	0.8531E-09	0.1810E-05	-0.7437E-09	
	0.1617E+00	0.8660E+00	0.5667E+00	0.3126E+01
	0.1435E+00	0.1229E+01	0.6868E+00	
3. 128 by 128, 22 degrees				
Boeing 727-200 (Boe727)				
	363.45	201.46		
	0.4089E+00	0.5043E-01	0.1526E-02	0.2959E-04
	0.6262E-08	0.6603E-05	-0.5607E-09	
	0.3464E+00	0.6146E+00	0.4887E+00	0.3124E+01
	0.2321E+00	0.9696E+00	0.6077E+00	
4. 128 by 128, 45 degrees				
Boeing 727-200 (Boe727)				
	361.52	195.62		
	0.4040E+00	0.4445E-01	0.1473E-02	0.5303E-05
	0.4319E-09	0.1065E-05	-0.1821E-09	
	0.5320E+00	0.4282E+00	0.5016E+00	0.3572E+01
	0.4772E+00	0.7914E+00	0.6834E+00	
5. 256 by 256, 0 degrees				
Boeing 727-200 (Boe727)				
	349.16	195.71		
	0.4169E+00	0.5163E-01	0.1407E-02	0.7583E-06
	-0.6527E-11	0.5030E-07	-0.2389E-10	
	0.1281E+00	0.8062E+00	0.5277E+00	0.2951E+01
	0.8350E-01	0.1129E+01	0.6585E+00	
6. 256 by 256, 22 degrees				
Boeing 727-200 (Boe727)				
	354.26	197.42		
	0.4102E+00	0.5072E-01	0.1313E-02	0.9787E-06
	-0.2698E-10	-0.1929E-06	0.2243E-10	
	0.3156E+00	0.5782E+00	0.4837E+00	0.3040E+01
	0.2007E+00	0.9158E+00	0.5902E+00	
7. 256 by 256, 45 degrees				
Boeing 727-200 (Boe727)				
	346.98	194.63		
	0.4158E+00	0.5014E-01	0.1362E-02	0.5300E-06
	0.1001E-10	0.7382E-08	-0.1012E-10	
	0.5067E+00	0.3884E+00	0.4925E+00	0.3420E+01
	0.4166E+00	0.7406E+00	0.6387E+00	

Table 8. The invariant attributes for all 21 aircraft.

1. Boeing 727-200 (Boe727)			
342.99	194.24		
0.4188E+00	0.5142E-01	0.1389E-02	0.1416E-05
0.5394E-10	-0.2622E-07	-0.3213E-10	
0.1371E+00	0.8041E+00	0.5359E+00	0.2953E+01
0.7683E-01	0.1133E+01	0.6468E+00	
2. Boeing 747-200B (Boe747)			
990.68	359.65		
0.3734E+00	0.1079E-01	0.1146E-04	0.1562E-02
-0.1963E-06	0.1622E-03	-0.7188E-07	
0.2371E+01	0.3006E+01	0.2539E+01	0.1407E+02
0.1140E+01	0.5695E+01	0.6010E+00	
3. Boeing 757-200 (Boe757)			
401.69	206.79		
0.4011E+00	0.3218E-01	0.6562E-05	0.4333E-03
-0.3422E-08	0.7772E-04	0.2285E-07	
0.4623E+00	0.6348E+00	0.3589E+00	0.4424E+01
0.1903E+00	0.1559E+01	0.3450E+00	
4. Mikoyan MiG-25 (MiG-25)			
222.05	148.83		
0.2421E+00	0.6912E-02	0.7122E-02	0.1450E-02
0.4659E-05	0.1205E-03	0.3679E-07	
0.8779E+00	0.3601E+00	0.1706E+00	0.6346E+01
0.1584E+01	0.8978E+00	0.4220E+00	
5. Cessna Citation III (Ccit3)			
63.06	82.91		
0.3477E+00	0.6218E-02	0.1684E-03	0.3954E-03
-0.9802E-07	0.3093E-04	0.2823E-07	
0.9151E+00	0.2499E+01	0.1652E+01	0.9928E+01
0.1568E+01	0.3534E+01	0.2266E+01	
6. Northrop F-5E Tiger II (F-5E)			
34.45	57.24		
0.3072E+00	0.2955E-01	0.7850E-02	0.3817E-02
0.2090E-04	0.6562E-03	0.8802E-08	
0.3581E+00	0.6761E+00	0.4744E-01	0.4412E+01
0.7289E+00	0.1459E+01	0.1744E+00	
7. Tupolev Tu-28P (Tu-28P)			
184.49	123.15		
0.2988E+00	0.2006E-01	0.1325E-01	0.2470E-02
0.1413E-04	0.3498E-03	0.8348E-07	
0.3068E+00	0.6372E+00	0.8053E+00	0.3241E+01
0.8064E+00	0.1054E+01	0.1927E+00	

Table 8 continued on next page.

Table 8, continued.

8. McDonnell Douglas DC-10 (DC-10)			
773.47	276.56		
0.3232E+00	0.1097E-01	0.4864E-02	0.8598E-04
0.5550E-07	0.8986E-05	0.3429E-08	
0.5549E+00	0.5992E+00	0.6978E+00	0.5561E+01
0.4426E+00	0.1499E+01	0.1566E+00	
9. McDonnell Douglas DC-9 (DC-9)			
246.64	175.04		
0.4726E+00	0.8529E-01	0.1143E-01	0.4054E-02
0.2759E-04	0.1182E-02	0.4852E-06	
0.1882E+00	0.6260E+00	0.4973E+00	0.2968E+01
0.7634E-01	0.1107E+01	0.5542E+00	
10. McDonnell Douglas F-15C Eagle (F-15C)			
96.92	74.87		
0.2281E+00	0.6876E-02	0.5226E-02	0.8307E-03
0.1730E-05	0.6888E-04	-0.3502E-07	
0.5318E-01	0.1154E+00	0.4689E+00	0.2962E+01
0.7792E+00	0.3791E+00	0.1437E+00	
11. Lockheed P-3C Orion (LP-3C)			
226.99	167.23		
0.3643E+00	0.1543E-02	0.1736E-01	0.1656E-02
0.8878E-05	0.6503E-04	0.4270E-07	
0.5297E+00	0.6308E-01	0.2219E+00	0.5872E+01
0.1506E+01	0.1055E+01	0.9400E+00	
12. Grumman A6-E/TRAM (A6-E)			
74.52	74.29		
0.3020E+00	0.8562E-04	0.3137E-03	0.9690E-03
0.4648E-06	0.8841E-05	-0.2634E-06	
0.1491E+01	0.7446E+00	0.1372E+01	0.8595E+01
0.1173E+01	0.1776E+01	0.1280E+01	
13. General Dynamics F-16XL (F-16XL)			
71.26	58.57		
0.2292E+00	0.9834E-02	0.8638E-02	0.1198E-02
0.3852E-05	0.1187E-03	-0.1145E-06	
0.1672E+00	0.1660E-01	0.6080E+00	0.3453E+01
0.9877E+00	0.4660E+00	0.9338E-01	
14. de Havilland DHC-6 Twin Otter (DHC-6)			
72.73	89.59		
0.4193E+00	0.2004E-01	0.1428E-01	0.3992E-03
0.9295E-06	-0.5584E-04	0.2099E-06	
0.3074E+01	0.1434E+01	0.3464E+00	0.2422E+02
0.5416E+01	0.6330E+01	0.3068E+01	

Table 8 continued on next page.

Table 8, continued.

## 15. Cessna Skylane RG (CRG)

24.23	43.44		
0.3792E+00	0.1161E-01	0.2360E-01	0.2605E-03
0.6127E-06	-0.2753E-04	0.2045E-06	
0.9046E+01	0.3727E+01	0.9003E+01	0.5227E+02
0.1767E+02	0.3774E+01	0.1133E+02	

## 16. Cessna 402C (C402C)

38.48	62.89		
0.3428E+00	0.4550E-02	0.8726E-02	0.2245E-03
0.2282E-06	-0.1348E-04	-0.2159E-06	
0.4740E+01	0.1060E+01	0.9519E+00	0.3082E+02
0.6998E+01	0.6157E+01	0.4430E+01	

## 17. Lockheed C-5B Galaxy (C-5B)

1160.51	386.29		
0.3651E+00	0.6432E-02	0.7851E-03	0.2087E-02
0.2656E-05	0.1673E-03	-0.2829E-06	
0.7124E+00	0.7942E+00	0.4091E+00	0.1070E+02
0.1658E+01	0.3107E+01	0.4097E+00	

## 18. Piper Chieftain (Piper)

39.34	60.68		
0.2928E+00	0.1007E-02	0.5686E-02	0.4764E-03
0.7800E-06	-0.1504E-04	0.7880E-07	
0.3593E+01	0.3624E+00	0.5415E+00	0.2165E+02
0.4170E+01	0.3332E+01	0.2916E+01	

## 19. Rockwell International B-1B (B-1B)

384.89	171.74		
0.3085E+00	0.3759E-01	0.1484E-01	0.3385E-02
0.2399E-04	0.6559E-03	-0.6099E-06	
0.1819E+00	0.1837E+00	0.5466E+00	0.2365E+01
0.6599E+00	0.4550E+00	0.6577E-01	

## 20. Lockheed C-130E Hercules (Herc)

322.47	186.90		
0.3287E+00	0.6477E-02	0.4747E-02	0.5058E-05
0.7185E-09	-0.3943E-06	0.3130E-09	
0.2286E+00	0.3617E+00	0.4457E+00	0.6713E+01
0.8517E+00	0.1688E+01	0.3150E+00	



Table 9. Partition coefficients of the cluster analyses.

c	Attributes vector used for the cluster analysis				
	A	B	C	D	E
2	0.53	0.91	0.73	0.74	0.86
3	0.48	0.90	0.62	0.53	0.86
4	0.42	0.86	0.54	0.53	0.85
5	0.43	0.87	0.55	0.50	0.69

Note: c = Number of cluster centers.  
 A = full 16 member attribute vector.  
 B = area and perimeter attributes only.  
 C = first 4 invariant moment attributes.  
 D = all 7 invariant moment attributes.  
 E = all 7 Fourier descriptors.

Table 10. Clustering results for all 16 attributes.

c	Cluster center				
	1	2	3	4	5
2	Tu-28P.73	Ccit3 .65			
	MiG-25.72	Piper .63			
	LP-3C .67	C402C .62			
	F-16XL.65	DHC-6 .61			
	F-5E .63	Boe747.58			
	F-15C .62	CRG .62			
	B-1B .56	A6-E .53			
	DC-9 .55	Herc .53			
		DC-10 .52			
		Boe757.52			
	C-5B .51				
	Boe727.51				
3	Tu-28P.88	Herc .84	C402C .68		
	F-5E .77	Boe757.78	DHC-6 .68		
	DC-9 .51	DC-10 .71	Piper .63		
	B-1B .47	Boe727.66	CRG .48		
	LP-3C .44	MiG-25.64	Ccit3 .47		
		F-15C .64			
		A6-E .54			
		F-16XL.53			
		C-5B .46			
		Boe747.39			
4	F-5E .85	DHC-6 .68	MiG-25.78	Boe757.75	
	Tu-28P.62	C402C .62	F-15C .74	DC-10 .68	
	DC-9 .46	Piper .46	F-16XL.73	Boe727.59	
	B-1B .36	CRG .42	A6-E .47	Herc .55	
			LP-3C .45	C-5B .41	
				Boe747.38	
				Ccit3 .32	
5	F-16XL.79	Boe747.80	Boe757.85	F5-E .90	DHC-6 .71
	F-15C .77	C-5B .55	Boe727.67	Tu-28P.52	C402C .52
	MiG-25.75		Herc .64	DC-9 .41	Piper .42
	LP-3C .43		DC-10 .54	B-1B .30	CRG .36
	A6-E .41		Ccit3 .30		

Table 11. Clustering results for area and perimeter.

c	Cluster center					
	1	2	3	4	5	
2	Boe747.99	Ccit3 .99				
	C-5B .96	Tu-28P.99				
	DC-10 .66	F-15C .99				
		A6-E .99				
		DHC-6 .99				
		F-16XL.98				
		F-5E .98				
		MiG-25.98				
		C402C .98				
		Piper .98				
		CRG .97				
		LP-3C .96				
		DC-9 .94				
3	Boe747.99	Ccit3 .99	Herc .99			
	C-5B .96	F-5E .99	Boe727.98			
	DC-10 .66	F-15C .99	DC-9 .97			
		A6-E .99	B-1B .96			
		F-16XL.99	LP-3C .93			
		C402C .99	Boe757.92			
		Piper .99	MiG-25.83			
		DHC-6 .97				
		CRG .97				
		Tu-28P.51				
	4	Boe747.99	A6-E .99	MiG-25.97	Boe727.96	
		C-5B .96	F-16XL.99	LP-3C .96	Boe757.94	
		DC-10 .48	C402C .99	DC-9 .85	B-1B .84	
		Piper .99	Tu-28P.68	Herc .84		
		F-5E .98				
		Ccit3 .96				
		F-15C .96				
		CRG .94				
		DHC-6 .91				
5		C-5B .96	Boe727.98	MiG-25.98	A6-E .99	DC-10 1.0
		Boe747.89	Herc .92	LP-3C .93	F-16XL.99	
			Boe757.89	DC-9 .75	C402C .99	
			B-1B .86	Tu-28P.71	Piper .99	
				F-5E .98		
				F-15C .96		
				Ccit3 .95		
				CRG .94		
				DHC-6 .90		

Table 12. Clustering results for first 4 moments.

----- Cluster center -----					
c	1	2	3	4	5
2	B-1B .94	DC-10 .96			
	Tu-28P.91	Piper .96			
	F-5E .85	Herc .95			
	DC-9 .74	Ccit3 .93			
	LP-3C .59	C402C .92			
		F-15C .91			
		A6-E .90			
		Boe757.82			
		Boe747.81			
		MiG-25.79			
3	B-1B .91	DHC-6 .87	Ccit3 .90		
	F-5E .86	LP-3C .76	A6-E .88		
	DC-9 .67	CRG .73	DC-10 .87		
	Tu-28P.50		Piper .87		
			Herc .85		
			F-15C .81		
			Boe747.76		
			Boe757.70		
			MiG-25.63		
			C-5B .62		
4	B-1B .91	Boe747.90	CRG .75	Piper .90	
	F-5E .80	C-5B .72	DHC-6 .72	F-15C .68	
	DC-9 .58	A6-E .58	LP-3C .71	DC-10 .87	
	Tu-28P.44	Boe757.52		Herc .84	
		Ccit3 .47		C402C .73	
		Boe727.37		F-16XL.52	
				MiG-25.47	
5	F-16XL.92	DC-10 .97	B-1B .89	Boe747.98	CRG .78
	MiG-25.92	Herc .93	F-5E .73	C-5B .76	DHC-6 .60
	F-15C .68	Piper .61	DC-9 .51	A6-E .44	LP-3C .55
		C402C .60	Tu-28P.36		
		Ccit3 .54			
		Boe757.36			
	Boe727.33				

Table 13. Clustering results for all 7 moments.

c	Cluster center				
	1	2	3	4	5
2	F-5E .93	DC-10 .97			
	Tu-28P.85	Herc .96			
	DC-9 .71	F-15C .95			
	B-1B .62	Ccit3 .94			
		Piper .93			
		Boe747.89			
		C402C .88			
		F-16XL.88			
		Boe757.88			
		MiG-25.86			
		A6-E .84			
3		Boe727.81			
		C-5B .73			
		DHC-6 .73			
		CRG .64			
		LP-3C .58			
	F-16XL.76	DC-10 .87	F-5E .92		
	C-5B .68	Herc .82	DC-9 .62		
	A6-E .68	Piper .82	Tu-28P.54		
	C402C .64	Ccit3 .72	B-1B .40		
	Boe747.60	Boe757.64			
		DHC-6 .62			
4		Boe727.56			
		MiG-25.52			
		CRG .51			
		F-15C .49			
		LP-3C .43			
	F-5E .93	DHC-6 .78	C-5B .80	DC-10 .92	
	DC-9 .51	CRG .76	A6-E .78	Herc .88	
	Tu-28P.39	LP-3C .62	C402C .53	Ccit3 .82	
	B-1B .33		F-16XL.48	Piper .76	
			Boe747.48	F-15C .68	
				Boe757.65	
5				MiG-25.52	
				Boe727.50	
	F-5E .93	DC-10 .90	C-5B .89	MiG-25.75	DHC-6 .82
	DC-9 .43	Herc .84	A6-E .70	F-16XL.73	CRG .80
	Tu-28P.32	Ccit3 .79	C402C .33	F-15C .62	LP-3C .36
	B-1B .27	Boe757.61			
		Piper .57			
		Boe727.45			
		Boe747.33			

Table 14. Clustering results for all 7 Fourier descriptors.

c	Cluster center				
	1	2	3	4	5
2	C402C .92	Boe727.99			
	DHC-6 .88	Boe757.99			
	Boe747.72	MiG-25.99			
	CRG .71	F-5E .99			
	Piper .51	Tu-28P.99			
		DC-10 .99			
		DC-9 .99			
		Herc .99			
		LP-3C .98			
		B-1B .98			
		F-16XL.97			
	F-15C .97				
	A6-E .96				
	C-5B .88				
	Ccit3 .52				
3	DHC-6 .95	CRG 1.0	Boe757.99		
	C402C .86		MiG-25.99		
	Boe747.76		F-5E .99		
	Piper .62		Tu-28P.99		
	Ccit3 .60		DC-10 .99		
			DC-9 .99		
			Herc .99		
			LP-3C .98		
			Boe727.98		
			B-1B .97		
			F-15C .96		
		F-16XL.96			
		A6-E .92			
		C-5B .76			

Table 14 continued on next page.

Table 14, continued.

----- Cluster center -----					
c	1	2	3	4	5
4	CRG 1.0	Ccit3 .88	Tu-28P.99	C402C .92	
		Boe747.82	Boe757.98	DHC-6 .91	
			MiG-25.98	Piper .50	
			F-5E .98		
			DC-10 .98		
			DC-9 .98		
			Herc .98		
			Boe727.96		
			LP-3C .96		
			B-1B .96		
			F-15C .95		
			F-16XL.94		
			A6-E .84		
		C-5B .63			
5	Boe747.86	B-1B .91	DC-10 .96	C402C .92	CRG 1.0
	Ccit3 .73	F-16XL.91	Boe757.95	DHC-6 .90	
		F-15C .90	F-5E .87	Piper .38	
		LP-3C .77	Herc .77		
		MiG-25.68	Boe727.73		
			DC-9 .68		
			A6-E .68		
			Tu-28P.64		
			C-5B .57		

Table 15. Invariant attributes of the 4 basic shapes  
 Numbers 1-5 are rectangles, 6-9 are squares,  
 10-16 are triangles, and 17-20 are circles.

1.	0.2383E+00	0.2902E-01	0.1916E-06	0.1498E-07
	-0.8009E-15	-0.2550E-08	0.5230E-16	
	0.4891E-02	0.2888E+00	0.6453E-02	0.2890E+01
	0.6720E-02	0.1848E+00	0.4873E-02	
2.	0.3071E+00	0.6651E-01	0.2602E-06	0.2424E-07
	-0.1911E-14	-0.6225E-08	-0.2364E-15	
	0.5424E-02	0.2518E+00	0.5390E-02	0.2182E+01
	0.5072E-02	0.7834E-01	0.4748E-02	
3.	0.1857E+00	0.6720E-02	0.4619E-06	0.1869E-07
	0.3837E-15	0.4243E-09	0.1694E-14	
	0.7218E-02	0.3141E+00	0.1054E-01	0.5411E+01
	0.9700E-02	0.5269E+00	0.1453E-01	
4.	0.2054E+00	0.1443E-01	0.1631E-06	0.1031E-07
	-0.4214E-15	-0.1236E-08	-0.3343E-16	
	0.7019E-02	0.2997E+00	0.7746E-02	0.3849E+01
	0.8292E-02	0.3246E+00	0.7150E-02	
5.	0.5613E+00	0.2872E+00	0.9343E-04	0.6790E-04
	0.5408E-08	0.3633E-04	-0.5562E-10	
	0.1137E-02	0.1971E+00	0.4408E-02	0.1520E+01
	0.1026E-01	0.3321E-01	0.5133E-02	
6.	0.1668E+00	0.5157E-04	0.2982E-06	0.3824E-08
	0.5552E-17	0.2173E-10	-0.1290E-15	
	0.5544E-01	0.2537E+00	0.2123E+00	0.6306E+02
	0.1660E+00	0.7008E+01	0.5522E-01	
7.	0.1667E+00	0.1368E-04	0.3331E-05	0.4775E-07
	0.1650E-13	0.1262E-09	-0.9519E-14	
	0.1919E+00	0.2274E+00	0.9722E-01	0.7543E+02
	0.5525E+00	0.8397E+01	0.1276E+00	
8.	0.1667E+00	0.3414E-05	0.3496E-06	0.4344E-08
	0.1575E-15	0.5821E-11	0.6202E-16	
	0.2286E+00	0.5449E+00	0.3296E+00	0.2267E+03
	0.3670E+00	0.2503E+02	0.2287E+00	
9.	0.1669E+00	0.6814E-04	0.6546E-07	0.2172E-08
	0.2417E-16	0.7486E-11	-0.9315E-17	
	0.1250E-01	0.2812E+00	0.3687E-01	0.5001E+02
	0.7688E-01	0.5557E+01	0.2910E-01	
10.	0.1985E+00	0.2350E-02	0.4758E-02	0.5624E-04
	-0.2897E-07	-0.2718E-05	0.2722E-08	
	0.5415E+01	0.2720E+00	0.6496E+01	0.6145E+02
	0.1219E+02	0.1820E+01	0.1233E+01	

Table 15 continued on next page.



Table 15, continued.

11.	0.2237E+00 0.1427E-05 0.3984E-01 0.4367E+00	0.1304E-01 0.7971E-04 0.3206E+00 0.2673E-01	0.5988E-02 0.1063E-07 0.3405E+00 0.1527E+00	0.6982E-03 0.2941E+01
12.	0.2506E+00 0.1110E-07 0.1336E+00 0.4586E+00	0.2574E-01 -0.2207E-04 0.7797E-01 0.1491E+00	0.6464E-02 -0.9558E-06 0.3482E-01 0.1228E+00	0.5209E-03 0.2178E+01
13.	0.3742E+00 0.3971E-05 0.1424E-01 0.9870E-01	0.1030E+00 0.6319E-04 0.1819E+00 0.1685E+00	0.1066E-01 0.6785E-05 0.2078E+00 0.4252E-01	0.1797E-02 0.1671E+01
14.	0.3018E+00 -0.3622E-06 0.2690E-01 0.3867E+00	0.5401E-01 -0.8273E-04 0.2086E+00 0.2667E+00	0.7947E-02 0.1933E-05 0.1533E+00 0.5400E-01	0.7866E-03 0.2181E+01
15.	0.4276E+00 0.4411E-03 0.1483E-01 0.1732E+00	0.1458E+00 0.7204E-02 0.1874E+00 0.2899E-01	0.2898E-01 0.1543E-06 0.9012E-01 0.8651E-01	0.1886E-01 0.1561E+01
16.	0.5818E+00 0.1923E-02 0.3804E-01 0.1511E+00	0.3016E+00 0.2207E-01 0.1584E+00 0.4982E-01	0.5434E-01 -0.4708E-04 0.7824E-01 0.7519E-01	0.4084E-01 0.1411E+01
17.	0.1592E+00 0.9415E-20 0.3721E+02 0.2775E+01	0.9736E-06 0.2932E-14 0.1127E+01 0.3638E+02	0.7497E-06 0.2009E-19 0.3853E+00 0.3649E+02	0.8691E-11 0.5721E+03
18.	0.1592E+00 0.9157E-20 0.1862E+02 0.5529E+00	0.6580E-06 0.8620E-14 0.4319E+00 0.1931E+02	0.8963E-07 0.5512E-20 0.1623E+01 0.1821E+02	0.1084E-10 0.2793E+03
19.	0.1593E+00 -0.8507E-17 0.4260E+01 0.8479E+00	0.1173E-04 -0.7453E-12 0.4084E+00 0.4145E+01	0.4778E-05 0.1442E-16 0.3645E+00 0.4173E+01	0.3886E-09 0.6448E+02
20.	0.1592E+00 0.8391E-20 0.6618E+01 0.2201E+00	0.6620E-05 0.1616E-13 0.7103E+00 0.6712E+01	0.3712E-06 -0.9347E-20 0.5757E+00 0.6666E+01	0.7519E-11 0.1113E+03

Table 16. Fuzzy clustering results for all 4 basic shapes; invariant moments and Fourier descriptors.

Membership distribution of  
each shape in each cluster

Shape	Cluster 1	Cluster 2	Cluster 3	Cluster 4
1.	0.613	0.374	0.005	0.008
2.	0.210	0.780	0.004	0.006
3.	0.802	0.187	0.005	0.007
4.	0.752	0.236	0.005	0.007
5.	0.288	0.564	0.079	0.069
6.	0.924	0.070	0.002	0.004
7.	0.890	0.100	0.003	0.006
8.	0.564	0.295	0.034	0.106
9.	0.932	0.063	0.002	0.003
10.	0.390	0.369	0.090	0.151
11.	0.661	0.326	0.005	0.008
12.	0.404	0.578	0.008	0.011
13.	0.107	0.879	0.006	0.007
14.	0.112	0.882	0.003	0.003
15.	0.288	0.541	0.112	0.059
16.	0.001	0.001	0.998	0.000
17.	0.007	0.006	0.003	0.984
18.	0.402	0.265	0.042	0.292
19.	0.907	0.084	0.003	0.006
20.	0.685	0.248	0.017	0.049

Table 17. Fuzzy clustering results for all 4 basic shapes;  
invariant moments only.

Membership distribution of  
each shape in each cluster

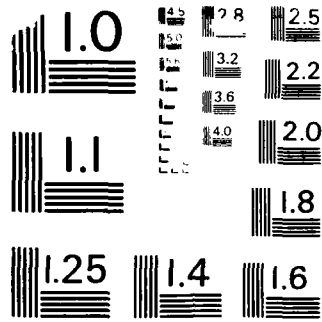
Shape	Cluster 1	Cluster 2	Cluster 3	Cluster 4
1.	0.140	0.845	0.002	0.013
2.	0.655	0.301	0.004	0.039
3.	0.002	0.997	0.000	0.000
4.	0.019	0.978	0.000	0.002
5.	0.008	0.004	0.001	0.987
6.	0.009	0.990	0.000	0.001
7.	0.009	0.990	0.000	0.001
8.	0.009	0.990	0.000	0.001
9.	0.009	0.990	0.000	0.001
10.	0.039	0.955	0.001	0.004
11.	0.126	0.861	0.002	0.011
12.	0.295	0.682	0.003	0.020
13.	0.893	0.066	0.003	0.038
14.	0.844	0.138	0.002	0.016
15.	0.430	0.215	0.077	0.278
16.	0.000	0.000	1.000	0.000
17.	0.013	0.985	0.000	0.002
18.	0.013	0.985	0.000	0.002
19.	0.013	0.985	0.000	0.002
20.	0.013	0.985	0.000	0.002

Table 18. Fuzzy clustering results for all 4 basic shapes;  
Fourier descriptors only.

Membership distribution of  
each shape in each cluster

Shape	Cluster 1	Cluster 2	Cluster 3	Cluster 4
1.	0.970	0.001	0.025	0.004
2.	0.985	0.000	0.012	0.002
3.	0.951	0.001	0.041	0.007
4.	0.963	0.001	0.031	0.005
5.	0.981	0.001	0.015	0.003
6.	0.687	0.006	0.270	0.037
7.	0.600	0.008	0.339	0.052
8.	0.206	0.050	0.445	0.300
9.	0.789	0.004	0.182	0.025
10.	0.275	0.101	0.326	0.298
11.	0.931	0.002	0.059	0.009
12.	0.903	0.004	0.074	0.019
13.	0.979	0.001	0.017	0.004
14.	0.990	0.000	0.008	0.002
15.	0.981	0.001	0.015	0.003
16.	0.966	0.001	0.027	0.006
17.	0.000	0.999	0.000	0.000
18.	0.024	0.018	0.048	0.909
19.	0.272	0.005	0.685	0.038
20.	0.141	0.015	0.731	0.112





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

Table 19. Fuzzy clustering results for circles and squares;  
invariant moments and Fourier descriptors.

Shape	Membership distribution of each shape in each cluster	
	Cluster 1	Cluster 2
6.	0.912	0.088
7.	0.587	0.413
8.	0.557	0.443
9.	0.844	0.156
17.	0.249	0.751
18.	0.211	0.789
19.	0.654	0.346
20.	0.609	0.391

Table 20. Fuzzy clustering results for circles and squares;  
invariant moments only.

Shape	Membership distribution of each shape in each cluster	
	Cluster 1	Cluster 2
6.	0.063	0.937
7.	0.998	0.002
8.	0.017	0.983
9.	0.107	0.893
17.	0.018	0.982
18.	0.023	0.977
19.	0.156	0.844
20.	0.011	0.989

Table 21. Fuzzy clustering results for circles and squares;  
Fourier descriptors only.

Membership distribution of  
each shape in each cluster

Shape	Cluster 1	Cluster 2
6.	0.013	0.987
7.	0.017	0.983
8.	0.080	0.920
9.	0.022	0.978
17.	0.989	0.011
18.	0.367	0.633
19.	0.018	0.982
20.	0.040	0.960



Table 22. The input data for blob1.

Chain code data for file: BLOB1.chn

-----  
 No. of links = 775  
 Starting (x,y) = ( 38, 57)

```

0070000000000012223232343333434434343434344444443444343332
232222121111101101001001000000000000070700700707077777777
707776777676766767767767767776707000000001001010101010101
21101011101010011000000007070707077767766766666766666665666
666566656656565554545454544454445444544454444544444445444444544
44444444545444444454454454545454445444545565556565566566666766
6777777677070707007000000000000000010000010000001000000
00001000070000000070777707767666665666555555555455454544544
45454545445444545455554554555555565656565565656556565656656
55556545454555455454545454445444444443444334323432322321
2222121212112121121122121211121122122121221222222222232232
22322323223223233322223232322323232232232232323222322322
322322323222222322222212221221221121111111011011121100
  
```

Mask data for file: BLOB1.msk

-----  
 No. of mask elements = 255

Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl
13	31	45	14	28	47	15	25	50	16	23	53
17	22	55	18	20	57	19	19	58	20	18	59
21	17	60	22	16	61	23	15	62	24	14	63
25	14	64	26	13	65	27	13	66	28	13	68
29	13	69	30	13	70	31	14	71	32	14	71
33	14	72	34	15	73	35	16	74	36	17	74
37	19	75	38	23	75	39	30	76	40	33	76
41	35	76	41	132	140	42	37	77	42	131	142
43	39	77	43	128	144	44	41	78	44	126	146
45	44	79	45	124	148	46	46	80	46	123	149
47	47	80	47	122	150	48	48	81	48	120	151
49	49	82	49	118	151	50	51	82	50	117	152
51	51	83	51	116	153	52	52	84	52	116	153
53	52	84	53	114	153	54	53	85	54	112	154
55	53	86	55	110	154	56	53	87	56	108	154
57	36	40	57	53	88	57	106	154	58	35	88
58	104	154	59	34	90	59	101	154	60	34	155
61	33	155	62	32	155	63	30	155	64	29	155
65	27	155	66	26	155	67	25	155	68	24	155
69	23	154	70	22	154	71	21	154	72	20	154
73	19	154	74	19	154	75	18	154	76	17	153

Table 22 continued on next page.

Table 22, continued.

Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl
77	17	153	78	17	153	79	16	153	80	16	152
81	16	152	82	15	152	83	15	151	84	15	151
85	15	150	86	14	150	87	14	149	88	14	148
89	14	147	90	14	145	91	14	143	92	14	141
93	15	139	94	15	135	95	15	131	96	15	127
97	15	122	98	15	114	99	15	106	100	16	95
101	16	93	102	17	85	103	17	82	104	17	79
105	18	77	106	18	75	107	18	74	108	19	72
109	19	68	110	19	66	111	20	65	112	20	65
113	20	64	114	21	63	115	21	62	116	21	62
117	21	61	118	22	61	119	22	60	120	23	59
121	23	59	122	24	59	123	24	58	124	25	58
125	25	58	126	25	58	127	26	58	128	26	58
129	26	58	130	27	59	131	27	59	132	27	59
133	28	59	134	28	60	135	29	61	136	29	62
137	30	63	138	30	64	139	30	65	140	31	66
141	31	66	142	32	67	143	32	69	143	122	126
144	33	71	144	110	135	145	33	73	145	103	137
146	34	76	146	97	138	147	34	139	148	34	140
149	34	142	150	34	143	151	35	144	152	36	144
153	37	145	154	37	145	155	38	145	156	38	145
157	38	145	158	39	145	159	39	144	160	39	144
161	40	144	162	40	144	163	41	143	164	41	142
165	41	141	166	42	140	167	42	139	168	42	138
169	42	137	170	43	136	171	43	135	172	43	133
173	44	132	174	44	131	175	44	129	176	44	127
177	44	124	178	44	120	179	44	118	180	44	116
181	44	114	182	44	111	183	44	107	184	44	105
185	44	103	186	43	102	187	43	101	188	43	100
189	42	98	190	42	97	191	41	95	192	41	94
193	41	93	194	40	92	195	40	91	196	40	90
197	39	89	198	38	88	199	38	88	200	37	87
201	36	87	202	35	86	203	35	86	204	34	85
205	34	84	206	33	84	207	33	83	208	33	82
209	32	82	210	31	81	211	31	81	212	30	80
213	29	79	214	29	79	215	28	78	216	28	77
217	27	77	218	26	76	219	26	76	220	25	75
221	25	75	222	24	75	223	24	74	224	23	74
225	23	73	226	23	72	227	23	71	228	23	70
229	22	70	230	22	69	231	23	67	232	23	65
233	23	63	234	23	62	235	24	61	236	24	59
237	25	58	238	27	56	239	27	54	240	28	52
241	30	50	242	31	48	243	35	44			

Table 23. The input data for blob2.

Chain code data for file: BLOB2.chn

-----  
 No. of links = 641  
 Starting (x,y) = ( 23, 113)

```

111010101010010100010000010000000010001001000100010010000100
001000100101101111121121122122121212122122212122212222122212
1221212112121111112110101010101010100010000010000000000007
00070007070707077707707077777677676767676767676767676766676666766667666
666666666666666666666666566566565656556554555555455454545454545454545
454454454454544545444544445444545445444544544544545454545455554545
556556666666666666676676766766766766766766676667666766676667666766666666
566666665656656656565655555656555554555454554545444444544
444444444344444443434343333333332323232223222222222222221221
221222212222122212222122222222222322322322323233333434334334
3434434344343333323223222212122112111101
  
```

Mask data for file: BLOB2.msk

-----  
 No. of mask elements = 231

Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl
13	157	169	14	151	173	15	147	177	16	145	179
17	143	181	18	141	183	19	139	184	20	137	185
21	135	187	22	133	188	23	131	190	24	130	192
25	129	193	26	129	194	27	128	195	28	127	196
29	126	197	30	125	197	31	124	198	32	123	199
33	123	199	34	122	200	35	122	200	36	121	201
37	120	202	38	120	202	39	119	203	40	119	203
41	118	204	42	118	205	43	118	205	44	117	206
45	117	206	46	116	207	47	116	207	48	116	207
49	116	208	50	115	208	51	115	208	52	115	208
53	115	208	54	115	209	55	114	209	56	114	209
57	114	209	58	114	209	59	113	210	60	113	210
61	112	210	62	112	210	63	112	210	64	112	210
65	111	210	66	111	210	67	111	210	68	110	210
69	110	210	70	109	210	71	109	210	72	108	210
73	108	210	74	107	210	75	107	210	76	107	210
77	106	210	78	106	210	79	106	210	80	105	210
81	104	209	82	104	209	83	103	209	84	102	208
85	102	208	86	101	208	87	100	207	88	99	207
89	98	206	90	96	206	91	95	205	92	93	204
93	90	204	94	86	203	95	81	202	96	76	200

Table 23 continued on next page.

Table 23, continued.

Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl
97	73	199	98	69	198	99	65	197	100	62	196
101	58	195	102	49	193	103	43	192	104	39	190
105	37	188	106	34	186	107	32	184	108	30	181
109	28	179	110	26	177	111	25	175	112	24	173
113	23	170	114	21	167	115	20	164	116	19	162
117	18	159	118	17	157	119	17	153	120	16	148
121	15	144	122	15	142	123	15	139	124	14	135
125	14	133	126	13	130	127	13	128	128	13	126
129	13	124	130	13	122	131	13	121	132	14	120
133	14	119	134	14	117	135	15	115	136	15	114
137	16	113	138	17	113	139	18	112	140	19	111
141	20	111	142	22	111	143	25	111	144	27	111
145	30	111	146	32	111	147	34	111	148	35	111
149	37	111	150	38	111	151	40	111	152	42	111
153	43	112	154	44	112	155	45	112	156	46	113
157	46	113	158	47	114	159	47	114	160	48	114
161	48	115	162	48	115	163	49	115	164	49	116
165	49	116	166	50	116	167	50	117	168	50	117
169	51	118	170	51	118	171	51	118	172	51	119
173	51	119	174	51	119	175	51	120	176	51	120
177	51	120	178	51	120	179	51	121	180	51	121
181	50	121	182	50	121	183	50	122	184	50	122
185	50	122	186	49	122	187	49	123	188	49	123
189	49	123	190	48	123	191	48	123	192	48	123
193	48	123	194	48	123	195	47	123	196	47	122
197	47	122	198	47	122	199	47	122	200	46	122
201	46	122	202	46	122	203	45	122	204	45	121
205	45	121	206	44	120	207	44	120	208	44	120
209	44	119	210	44	119	211	44	119	212	44	118
213	44	118	214	44	117	215	44	117	216	44	116
217	44	115	218	44	115	219	44	114	220	45	114
221	45	113	222	45	112	223	45	111	224	46	110
225	46	109	226	47	108	227	48	108	228	48	107
229	49	107	230	49	106	231	50	105	232	51	104
233	52	103	234	53	102	235	54	100	236	55	99
237	56	98	238	57	96	239	58	94	240	60	93
241	62	91	242	64	89	243	72	83			

Table 24. The input data for blob3.

Chain code data for file: BLOB3.chn

-----  
 No. of links = 741  
 Starting (x,y) = ( 37, 57)

```

00000700000000112223232333434343434343443434443444344434443
32222211212111111001001001000000000007000000707077077077
7770777676776767676767676767677677707007000000001001010010010
101111111011011010100000007077077677676767666676666666665665
6666665666565555454545454454445444544454445444544444444454444454
44444454444445444454445455454545455555656566656666666666767
777777760700700000000000001000100000001000000000007707
767667667666666656566665556545545454545455545454555454545444
6556565556565656565656565656565655545555545545444544454445444
45444444443433434322322222222212122121212121212121212121212
1212121221222222222322232223232232322322232232323232323232
22322323232232323223223223222322232232232232212222212221212
121111111111111121002
  
```

Mask data for file: BLOB3.msk

-----  
 No. of mask elements = 254

Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl
13	31	43	14	28	50	15	25	52	16	22	54
17	21	55	18	20	57	19	19	58	20	18	60
21	17	61	22	16	63	23	16	64	24	15	65
25	15	66	26	14	67	27	13	69	28	13	70
29	13	71	30	13	72	31	13	72	32	13	73
33	14	73	34	15	74	35	19	75	36	23	75
37	27	76	38	31	76	39	33	77	40	36	77
40	133	140	41	38	78	41	131	142	42	39	78
42	129	143	43	41	79	43	128	145	44	43	79
44	126	146	45	45	80	45	125	147	46	47	80
46	123	147	47	49	81	47	122	148	48	50	81
48	121	149	49	51	82	49	120	149	50	51	82
50	119	150	51	52	83	51	118	150	52	52	84
52	117	151	53	53	84	53	115	152	54	53	85
54	113	152	55	53	86	55	110	153	56	53	87
56	107	153	57	37	42	57	52	89	57	105	153
58	35	92	58	102	153	59	34	153	60	34	154
61	33	154	62	32	154	63	31	154	64	30	154
65	29	154	66	28	154	67	27	154	68	26	154
69	25	153	70	24	153	71	23	153	72	22	152
73	21	152	74	20	152	75	20	152	76	19	152

Table 24 continued on next page.

Table 24, continued.

Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl
77	19	152	78	18	152	79	18	151	80	17	151
81	17	151	82	17	151	83	17	150	84	16	150
85	16	149	86	16	148	87	16	147	88	16	146
89	16	144	90	15	142	91	15	140	92	15	138
93	16	135	94	16	133	95	16	129	96	17	126
97	17	122	98	17	118	99	18	109	100	18	103
101	18	95	102	18	88	103	19	83	104	19	79
105	19	77	106	19	76	107	20	74	108	20	72
109	20	70	110	20	68	111	21	67	112	21	66
113	21	65	114	22	64	115	22	63	116	22	63
117	22	62	118	23	62	119	23	61	120	24	61
121	24	61	122	25	61	123	25	60	124	26	60
125	26	60	126	26	60	127	27	60	128	27	60
129	28	60	130	28	60	131	29	60	132	29	60
133	29	60	134	30	61	135	30	61	136	30	62
137	30	63	138	31	64	139	31	65	140	31	66
141	32	67	142	32	68	143	33	69	144	33	70
145	34	71	145	106	117	146	34	74	146	98	118
147	35	77	147	94	120	148	35	121	149	36	122
150	36	122	151	37	123	152	37	123	153	37	123
154	38	124	155	38	124	156	38	124	157	38	125
158	39	125	159	39	125	160	39	125	161	40	125
162	40	125	163	41	125	164	41	125	165	41	125
166	42	124	167	42	124	168	43	123	169	43	123
170	43	123	171	43	123	172	44	123	173	44	122
174	44	121	175	44	120	176	45	120	177	45	119
178	45	117	179	45	116	180	45	114	181	45	112
182	45	110	183	45	108	184	45	106	185	45	104
186	45	103	187	44	102	188	44	100	189	44	98
190	43	98	191	43	97	192	42	96	193	42	94
194	41	92	195	41	91	196	40	91	197	40	90
198	39	90	199	39	89	200	38	88	201	38	88
202	37	87	203	37	87	204	36	86	205	35	85
206	35	84	207	34	84	208	33	83	209	33	82
210	32	82	211	31	81	212	31	81	213	30	80
214	30	80	215	29	79	216	29	79	217	28	78
218	28	78	219	27	77	220	27	77	221	27	76
222	26	76	223	26	75	224	25	75	225	25	74
226	25	74	227	25	73	228	25	73	229	25	72
230	25	71	231	25	69	232	25	68	233	25	67
234	25	66	235	25	65	236	26	63	237	26	62
238	26	60	239	27	58	240	29	54	241	31	51
242	32	47	243	34	42						

Table 25. The input data for blob4.

Chain code data for file: BLOB4.chn

-----  
 No. of links = 615  
 Starting (x,y) = ( 18, 117)

```

12110110101010100100101000010001001010010010110110110110111
11111010111111111111111112111211112121121211121211211212121
111101101010101010001001001000000000700000070070707070707
7707077777767766767676767667676666676666667666666666666666666
66565656565555555655455455455454544544544544544544544544544
54544544454544454445445445445445454554555555656565666666766666
766766766767676767667676666667666666665666666566656665656556
55565656556565556554554545445454444444444443444444443443
433343333323323232322222232221222122212221222122212222212221
2212212222232222232323233333333343333343434434344343433432322
232222121212211
    
```

Mask data for file: BLOB4.msk

-----  
 No. of mask elements = 231

Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl
13	153	163	14	150	170	15	147	173	16	143	175
17	141	177	18	139	179	19	138	180	20	136	182
21	134	184	22	132	185	23	131	186	24	129	188
25	128	190	26	127	191	27	126	192	28	125	193
29	124	194	30	124	195	31	123	196	32	123	196
33	122	197	34	122	198	35	121	198	36	120	198
37	119	199	38	119	199	39	118	200	40	117	200
41	117	201	42	116	201	43	116	202	44	115	202
45	114	203	46	113	203	47	113	203	48	112	204
49	112	204	50	111	205	51	110	205	52	110	205
53	109	205	54	109	205	55	108	205	56	107	206
57	106	206	58	105	206	59	105	206	60	104	206
61	103	206	62	102	206	63	102	207	64	101	207
65	100	207	66	99	207	67	98	207	68	97	207
69	96	207	70	95	207	71	94	207	72	93	207
73	92	207	74	91	207	75	90	207	76	89	207
77	88	207	78	87	206	79	86	206	80	84	206
81	82	206	82	81	206	83	80	205	84	79	205
85	78	204	86	77	204	87	76	203	88	75	203
89	73	202	90	72	202	91	70	201	92	69	200
93	68	199	94	66	198	95	65	197	96	63	196

Table 25 continued on next page.

Table 25, continued.

Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl	Row	Bcl	Ecl
97	62	195	98	60	195	99	57	194	100	54	193
101	52	191	102	49	190	103	45	188	104	40	187
105	38	185	106	35	184	107	32	182	108	30	180
109	28	177	110	26	174	111	24	171	112	23	168
113	21	166	114	20	164	115	19	161	116	19	159
117	18	155	118	17	152	119	16	150	120	16	147
121	16	143	122	15	141	123	15	138	124	14	134
125	14	132	126	13	129	127	13	126	128	13	123
129	13	121	130	13	120	131	14	118	132	14	117
133	14	116	134	14	115	135	15	114	136	15	113
137	16	112	138	16	112	139	17	111	140	19	111
141	20	110	142	22	110	143	25	110	144	27	109
145	30	109	146	32	109	147	34	109	148	35	109
149	36	109	150	37	109	151	38	110	152	40	110
153	41	110	154	42	110	155	43	110	156	44	110
157	45	111	158	46	111	159	47	111	160	47	112
161	47	112	162	48	112	163	48	113	164	49	113
165	50	113	166	50	114	167	50	114	168	50	115
169	50	115	170	50	116	171	51	116	172	51	117
173	51	117	174	51	117	175	51	118	176	51	118
177	50	119	178	50	119	179	50	120	180	49	120
181	49	120	182	49	120	183	48	120	184	48	120
185	48	120	186	48	121	187	47	121	188	47	121
189	47	121	190	47	121	191	47	121	192	47	121
193	46	121	194	46	121	195	46	120	196	46	120
197	45	120	198	45	120	199	45	120	200	45	120
201	45	120	202	44	119	203	44	119	204	44	119
205	44	119	206	43	118	207	43	118	208	43	118
209	43	118	210	42	117	211	42	117	212	42	116
213	42	116	214	43	115	215	43	114	216	43	114
217	43	113	218	43	112	219	43	111	220	43	111
221	44	110	222	44	110	223	45	109	224	45	109
225	46	108	226	47	107	227	47	107	228	48	106
229	48	106	230	49	105	231	50	104	232	50	103
233	51	103	234	52	102	235	53	101	236	54	99
237	55	98	238	57	96	239	58	94	240	59	91
241	61	89	242	64	87	243	73	83			



Table 26. A comparison of the invariant attributes of the noisy blobs to the non-noisy blob.

Invariant moments:		Noise = 12.5		Noise = 25	
Index	Original	Data	(% diff)	Data	(% diff)
(2,1)	0.2933E+00	0.2979E+00	( 1.6)	0.3014E+00	( 2.8)
(2,2)	0.2159E-01	0.2253E-01	( 4.4)	0.2317E-01	( 7.3)
(2,3)	0.3489E-04	0.3664E-04	( 5.0)	0.3941E-04	( 13.0)
(3,1)	0.5684E-02	0.5927E-02	( 4.3)	0.6072E-02	( 6.8)
(3,2)	0.2457E-03	0.2526E-03	( 2.8)	0.2632E-03	( 7.1)
(3,3)	0.2878E-06	0.3063E-06	( 6.4)	0.3239E-06	( 12.5)
(3,4)	-0.3866E-07	-0.4189E-07	( 8.4)	-0.7623E-07	( 97.2)
(4,1)	0.4293E-03	0.4549E-03	( 6.0)	0.4826E-03	( 12.4)
(4,2)	0.6595E-02	0.7053E-02	( 6.9)	0.7429E-02	( 12.6)
(4,3)	0.1768E-01	0.1881E-01	( 6.4)	0.1972E-01	( 11.5)
(4,4)	0.4094E-03	0.4421E-03	( 8.0)	0.4689E-03	( 14.5)
(4,5)	0.4094E-03	0.4421E-03	( 8.0)	0.4689E-03	( 14.5)
(5,1)	0.5064E-02	0.5520E-02	( 9.0)	0.5939E-02	( 17.3)
(5,2)	0.2843E-02	0.3073E-02	( 8.1)	0.3230E-02	( 13.6)
(5,3)	0.3804E-03	0.4040E-03	( 6.2)	0.4259E-03	( 12.0)
(5,4)	0.4346E-04	0.4649E-04	( 7.0)	0.5181E-04	( 19.2)
(5,5)	0.4346E-04	0.4649E-04	( 7.0)	0.5181E-04	( 19.2)
Average  %diff		( 6.2)		( 17.3)	

Fourier Descriptors:		Noise = 12.5		Noise = 25	
Index	Original	Data	(% diff)	Data	(% diff)
-4	0.2394E+00	0.2428E+00	( 1.4)	0.2435E+00	( 1.7)
-3	0.7122E+00	0.6897E+00	( -2.1)	0.7113E+00	( 1.0)
-2	0.8878E+00	0.8543E+00	( -3.8)	0.8367E+00	( -5.8)
-1	0.3573E+01	0.3545E+01	( -0.8)	0.3442E+01	( -3.7)
2	0.4383E+00	0.4339E+00	( -1.0)	0.4789E+00	( 9.3)
3	0.7544E+00	0.7391E+00	( -2.0)	0.7199E+00	( -4.6)
4	0.5964E+00	0.6095E+00	( 2.2)	0.5878E+00	( -1.4)
Average  %diff		( 1.9)		( 3.9)	

Table 26 continued on next page.

Table 26, continued.

Invariant moments:		Noise = 50		Noise = 75	
Index	Original	Data	(% diff)	Data	(% diff)
(2,1)	0.2933E+00	0.3059E+00	( 4.3)	0.3208E+00	( 9.4)
(2,2)	0.2159E-01	0.2387E-01	( 10.6)	0.2617E-01	( 21.2)
(2,3)	0.3489E-04	0.4523E-04	( 29.6)	0.6045E-04	( 73.3)
(3,1)	0.5684E-02	0.6409E-02	( 12.8)	0.7729E-02	( 36.0)
(3,2)	0.2457E-03	0.2960E-03	( 20.5)	0.3773E-03	( 53.6)
(3,3)	0.2878E-06	0.3915E-06	( 36.0)	0.6201E-06	( 115.5)
(3,4)	-0.3866E-07	-0.1139E-06	( 194.6)	-0.1749E-06	( 352.4)
(4,1)	0.4293E-03	0.5673E-03	( 32.1)	0.6632E-03	( 54.5)
(4,2)	0.6595E-02	0.8025E-02	( 21.7)	0.9530E-02	( 44.5)
(4,3)	0.1768E-01	0.2116E-01	( 19.7)	0.2539E-01	( 43.6)
(4,4)	0.4094E-03	0.5381E-03	( 31.4)	0.6018E-03	( 47.0)
(4,5)	0.4094E-03	0.5381E-03	( 31.4)	0.6018E-03	( 47.0)
(5,1)	0.5064E-02	0.6658E-02	( 31.5)	0.8590E-02	( 69.6)
(5,2)	0.2843E-02	0.3601E-02	( 26.7)	0.4678E-02	( 64.5)
(5,3)	0.3804E-03	0.4953E-03	( 30.2)	0.6515E-03	( 71.3)
(5,4)	0.4346E-04	0.6182E-04	( 42.2)	0.8573E-04	( 97.3)
(5,5)	0.4346E-04	0.6182E-04	( 42.2)	0.8573E-04	( 97.3)
Average  %diff			( 36.3)	( 76.3)	

Fourier Descriptors:		Noise = 50		Noise = 75	
Index	Original	Data	(% diff)	Data	(% diff)
-4	0.2394E+00	0.2118E+00	( -11.5)	0.1851E+00	( -22.7)
-3	0.7042E+00	0.7131E+00	( 1.3)	0.7137E+00	( 1.3)
-2	0.8878E+00	0.7124E+00	( -19.8)	0.7393E+00	( -16.7)
-1	0.3573E+01	0.3270E+01	( -8.5)	0.3058E+01	( -14.4)
2	0.4383E+00	0.4524E+00	( 3.2)	0.4444E+00	( 1.4)
3	0.7544E+00	0.6513E+00	( -13.7)	0.5873E+00	( -22.2)
4	0.5964E+00	0.5540E+00	( -7.1)	0.5713E+00	( -4.2)
Average  %diff			( 9.3)	( 11.8)	

Table 27. A comparison of the invariant attributes of the noisy blob2s to the non-noisy blob2.

Invariant moments:		Noise = 12.5		Noise = 25	
Index	Original	Data	(% diff)	Data	(% diff)
(2,1)	0.2843E+00	0.2865E+00	( 0.8)	0.2881E+00	( 1.3)
(2,2)	0.3760E-01	0.3843E-01	( 2.2)	0.3922E-01	( 4.3)
(2,3)	-0.2565E-04	-0.2621E-04	( 2.2)	-0.2757E-04	( 7.5)
(3,1)	0.2727E-02	0.2768E-02	( 1.5)	0.2808E-02	( 3.0)
(3,2)	0.1531E-03	0.1547E-03	( 1.0)	0.1600E-03	( 4.5)
(3,3)	-0.9408E-07	-0.9641E-07	( 2.5)	-0.1025E-06	( 8.9)
(3,4)	0.3058E-07	0.3074E-07	( 0.5)	0.3131E-07	( 2.4)
(4,1)	0.2538E-02	0.2658E-02	( 4.7)	0.2739E-02	( 7.9)
(4,2)	0.8763E-02	0.9056E-02	( 3.3)	0.9314E-02	( 6.3)
(4,3)	0.1521E-01	0.1565E-01	( 2.9)	0.1600E-01	( 5.2)
(4,4)	0.1648E-02	0.1722E-02	( 4.5)	0.1790E-02	( 8.6)
(4,5)	0.1648E-02	0.1722E-02	( 4.5)	0.1790E-02	( 8.6)
(5,1)	0.1065E-02	0.1107E-02	( 3.9)	0.1148E-02	( 7.8)
(5,2)	0.8143E-03	0.8370E-03	( 2.8)	0.8639E-03	( 6.1)
(5,3)	0.8217E-04	0.8384E-04	( 2.0)	0.8751E-04	( 6.5)
(5,4)	-0.1481E-04	-0.1530E-04	( 3.3)	-0.1619E-04	( 9.3)
(5,5)	-0.1481E-04	-0.1530E-04	( 3.3)	-0.1619E-04	( 9.3)
Average  %diff		( 2.7)		( 6.3)	

Fourier Descriptors:		Noise = 12.5		Noise = 25	
Index	Original	Data	(% diff)	Data	(% diff)
-4	0.2388E+00	0.2314E+00	( -3.1)	0.2241E+00	( -6.2)
-3	0.5042E+00	0.5101E+00	( 1.2)	0.4969E+00	( -1.4)
-2	0.1831E+00	0.1873E+00	( 2.3)	0.1825E+00	( -0.3)
-1	0.2926E+01	0.2897E+01	( -1.0)	0.2887E+01	( -1.3)
2	0.3686E+00	0.3727E+00	( 1.1)	0.3787E+00	( 2.7)
3	0.2255E+00	0.2291E+00	( 1.6)	0.2470E+00	( 9.5)
4	0.2003E+00	0.1925E+00	( -3.9)	0.1801E+00	( -10.1)
Average  %diff		( 2.0)		( 4.5)	

Table 27 continued on next page.

Table 27, continued.

Invariant moments:		Noise = 50		Noise = 100	
Index	Original	Data	(% diff)	Data	(% diff)
(2,1)	0.2843E+00	0.2934E+00	( 3.2)	0.3039E+00	( 6.9)
(2,2)	0.3760E-01	0.4121E-01	( 9.6)	0.4525E-01	( 20.3)
(2,3)	-0.2565E-04	-0.3179E-04	( 23.9)	-0.3547E-04	( 38.3)
(3,1)	0.2727E-02	0.3085E-02	( 13.1)	0.3321E-02	( 21.8)
(3,2)	0.1531E-03	0.1769E-03	( 15.5)	0.1837E-03	( 20.0)
(3,3)	-0.9408E-07	-0.1268E-06	( 34.8)	-0.1418E-06	( 50.7)
(3,4)	0.3058E-07	0.3192E-07	( 4.4)	0.2183E-07	( -28.6)
(4,1)	0.2538E-02	0.3003E-02	( 18.3)	0.3588E-02	( 41.4)
(4,2)	0.8763E-02	0.1010E-01	( 15.3)	0.1171E-01	( 33.6)
(4,3)	0.1521E-01	0.1722E-01	( 13.2)	0.1967E-01	( 29.3)
(4,4)	0.1648E-02	0.1975E-02	( 19.8)	0.2366E-02	( 43.6)
(4,5)	0.1648E-02	0.1975E-02	( 19.8)	0.2366E-02	( 43.6)
(5,1)	0.1065E-02	0.1341E-02	( 25.9)	0.1647E-02	( 54.6)
(5,2)	0.8143E-03	0.9869E-03	( 21.2)	0.1131E-02	( 38.9)
(5,3)	0.8217E-04	0.9993E-04	( 21.6)	0.1101E-03	( 34.0)
(5,4)	-0.1481E-04	-0.1908E-04	( 28.8)	-0.2250E-04	( 51.9)
(5,5)	-0.1481E-04	-0.1908E-04	( 28.8)	-0.2250E-04	( 51.9)
Average  %diff			( 18.7)	( 35.9)	

Fourier Descriptors:		Noise = 50		Noise = 100	
Index	Original	Data	(% diff)	Data	(% diff)
-4	0.2388E+00	0.3234E+00	( 35.4)	0.2480E+00	( 3.9)
-3	0.5042E+00	0.4960E+00	( -1.6)	0.4621E+00	( -8.3)
-2	0.1831E+00	0.1467E+00	( -19.9)	0.2083E+00	( 13.8)
-1	0.2926E+01	0.2925E+01	( -0.0)	0.2777E+01	( -5.1)
2	0.3686E+00	0.4543E+00	( 23.3)	0.4585E+00	( 24.4)
3	0.2255E+00	0.2492E+00	( 10.5)	0.2784E+00	( 23.5)
4	0.2003E+00	0.1903E+00	( -5.0)	0.1954E+00	( -2.4)
Average  %diff			( 13.7)	( 11.6)	

Table 28. A comparison of the invariant attributes of blob3 to blob1.

Invariant moments:		Blob3	
Index	Original	Data	(% diff)
(2,1)	0.2933E+00	0.3051E+00	( 4.0)
(2,2)	0.2159E-01	0.2927E-01	( 35.6)
(2,3)	0.3489E-04	0.8068E-04	( 131.2)
(3,1)	0.5684E-02	0.7736E-02	( 36.1)
(3,2)	0.2457E-03	0.4716E-03	( 91.9)
(3,3)	0.2878E-06	0.8649E-06	( 200.5)
(3,4)	-0.3866E-07	-0.2516E-06	( 550.8)
(4,1)	0.4293E-03	0.1053E-02	( 145.3)
(4,2)	0.6595E-02	0.9563E-02	( 45.0)
(4,3)	0.1768E-01	0.2165E-01	( 22.5)
(4,4)	0.4094E-03	0.9466E-03	( 131.2)
(4,5)	0.4094E-03	0.9466E-03	( 131.2)
(5,1)	0.5064E-02	0.6352E-02	( 25.4)
(5,2)	0.2843E-02	0.3995E-02	( 40.5)
(5,3)	0.3804E-03	0.7097E-03	( 86.6)
(5,4)	0.4346E-04	0.1132E-03	( 160.5)
(5,5)	0.4346E-04	0.1132E-03	( 160.5)
Average  %diff			( 117.6)

Fourier Descriptors:		Blob3	
Index	Original	Data	(% diff)
-4	0.2394E+00	0.2281E+00	( -4.7)
-3	0.7042E+00	0.5937E+00	( -15.7)
-2	0.8878E+00	0.6040E+00	( -32.0)
-1	0.3573E+01	0.3142E+01	( -12.1)
2	0.4383E+00	0.5547E+00	( 26.6)
3	0.7544E+00	0.4735E+00	( -37.2)
4	0.5964E+00	0.4653E+00	( -22.0)
Average  %diff			( 21.5)

Table 29. A comparison of the invariant attributes of blob4 to blob2.

Invariant moments:		Blob4	
Index	Original	Data	(% diff)
(2,1)	0.2843E+00	0.2708E+00	( -4.7)
(2,2)	0.3760E-01	0.3306E-01	( -12.1)
(2,3)	-0.2565E-04	-0.3341E-04	( 30.3)
(3,1)	0.2727E-02	0.2681E-02	( -1.7)
(3,2)	0.1531E-03	0.1840E-03	( 20.2)
(3,3)	-0.9408E-07	-0.1181E-06	( 25.5)
(3,4)	0.3058E-07	-0.5238E-07	(-271.3)
(4,1)	0.2538E-02	0.2081E-02	( -18.0)
(4,2)	0.8763E-02	0.7425E-02	( -15.3)
(4,3)	0.1521E-01	0.1293E-01	( -15.0)
(4,4)	0.1648E-02	0.1325E-02	( -19.6)
(4,5)	0.1648E-02	0.1325E-02	( -19.6)
(5,1)	0.1065E-02	0.1011E-02	( -5.1)
(5,2)	0.8143E-03	0.7946E-03	( -2.4)
(5,3)	0.8217E-04	0.9525E-04	( 15.9)
(5,4)	-0.1481E-04	-0.1662E-04	( 12.2)
(5,5)	-0.1481E-04	-0.1662E-04	( 12.2)
Average  %diff			( 29.5)

Fourier Descriptors:		Blob4	
Index	Original	Data	(% diff)
-4	0.2388E+00	0.2103E+00	( -11.9)
-3	0.5042E+00	0.5265E+00	( 4.4)
-2	0.1831E+00	0.2853E+00	( 55.8)
-1	0.2926E+01	0.3127E+01	( 6.9)
2	0.3686E+00	0.4492E+00	( 21.9)
3	0.2255E+00	0.1765E+00	( -21.7)
4	0.2003E+00	0.1681E+00	( -16.1)
Average  %diff			( 19.8)

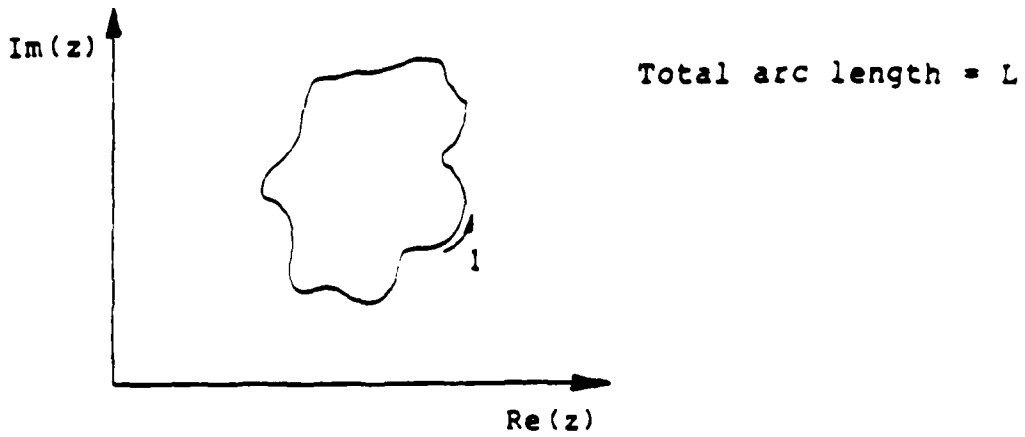
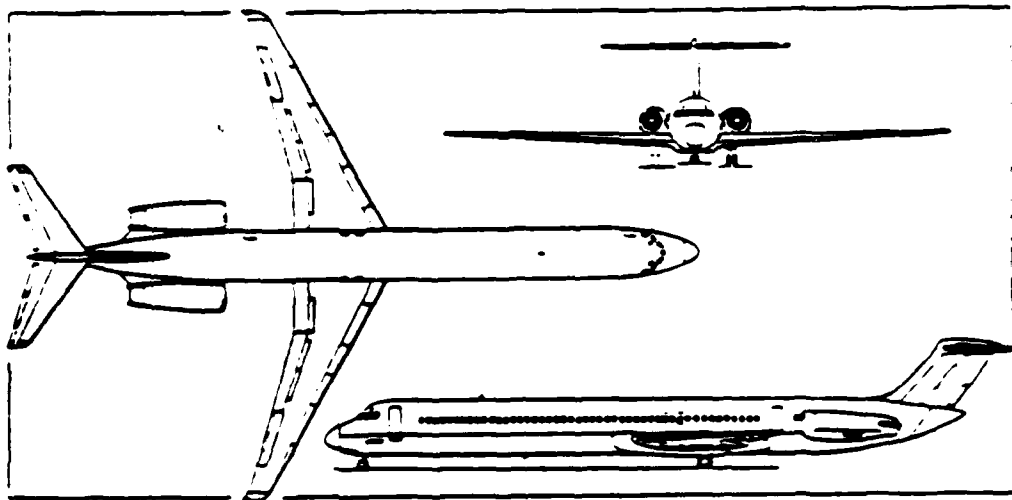


Figure 1. The complex function description of a closed boundary.



McDonnell Douglas DC-8 Super 80 'stretched' version of this non-turboprop transport (Pilot Press)

Figure 2. Example original object outline.

	3	4	5	6	7	8	9	10	11	12
10	x	x	x							x
11	x	-	x						x	x
12	x	x	-	x	x	x			x	x
13			x	-	-	-	x	x	x	x
14		x	x	x	-	-	-	-	-	x
15		x	-	-	-	-	-	-	x	
16			x	-	-	x	x	x	x	
17			x	x	x					

(a) The digitized outline; x = boundary point, - = internal point.

	row	bcol	ecol		row	bcol	ecol
	m	m	m		m	m	m
k	1k	2k	3k	k	1k	2k	3k
-	---	---	---	-	---	---	---
1	10	4	6	7	13	6	12
2	10	12	12	8	14	4	12
3	11	4	6	9	15	4	11
4	11	11	12	10	16	5	11
5	12	4	9	11	17	5	7
6	12	11	12				

(b) Mask description

Figure 3. Mask description of an object.



3	2	1
4	*	0
5	6	7

Figure 4. The 8 direction chain code.

File pos.	No. bits	Data type	Contents
0	16	I*2	Starting x coordinate of chain
1	16	I*2	Starting y coordinate of chain
2	16	I*2	Ending x coordinate of chain
3	16	I*2	Ending y coordinate of chain
4	32	I*4	No. of codes stored in this file
6	16	Encoded	Up to 5 codes stored as follows:
.			
.			

bit: |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|  
 | | | | | | | | | | | | | | | | |

bit 15: if = 0, this word contains 5 codes stored one for each 3 bits, first starting from bits 14 to 12;  
 if = 1, this is the last word in the chain indicating that:  
 (a) bits 14-12 contain the number (0 to 4) of codes stored in this last word,  
 (b) bits 11-0 contain up to 4 codes as indicated by bits 14-12.

Figure 5. Format of the chain code data file.

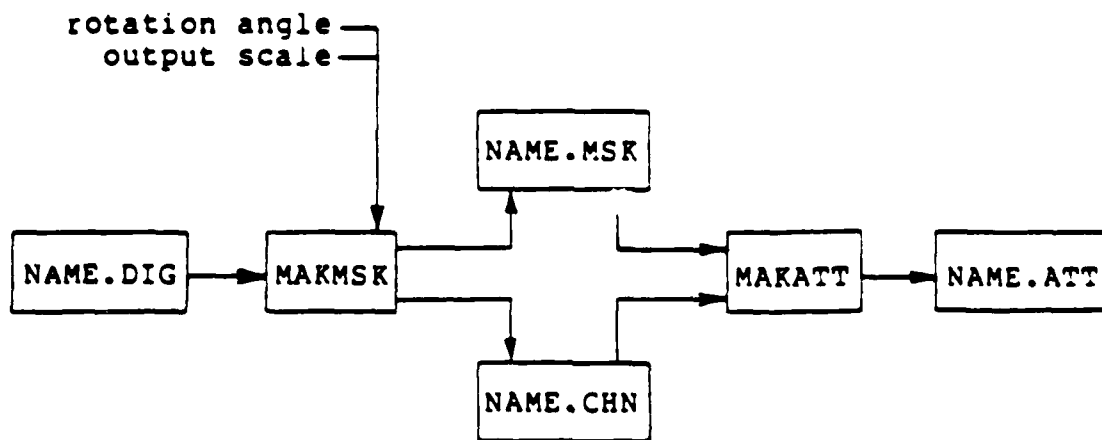
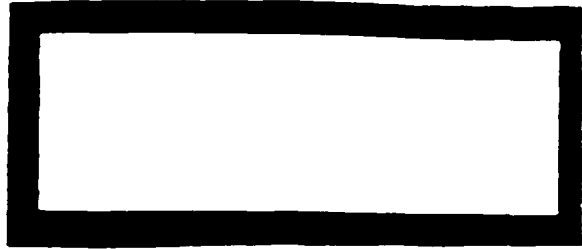


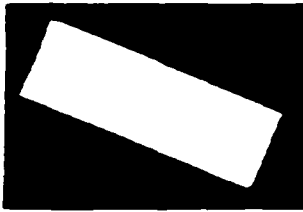
Figure 6. Flowchart of one computer run to generate an object's attributes.



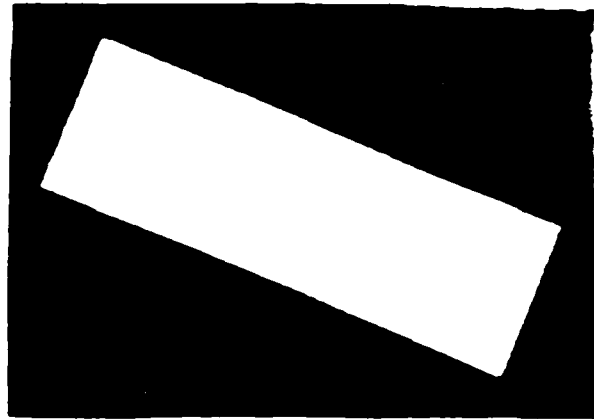
(a) 128 by 128, 0 deg.



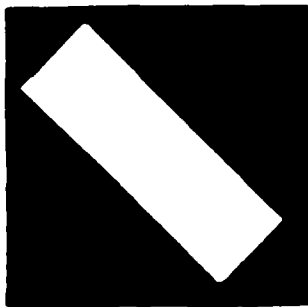
(d) 256 by 256, 0 deg.



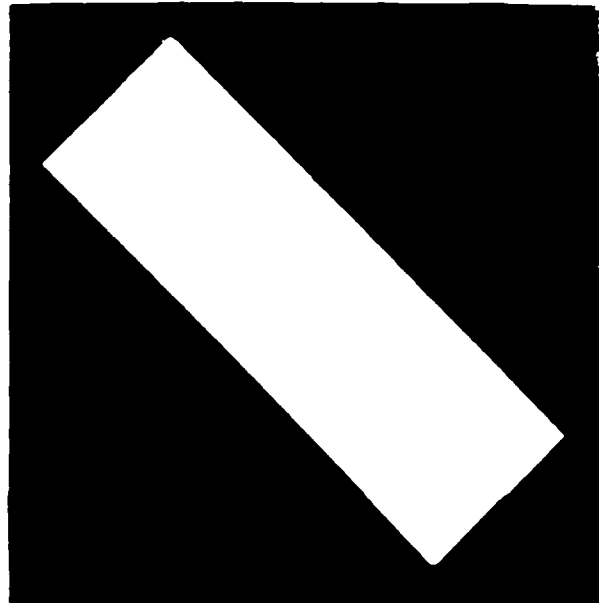
(b) 128 by 128, 22 deg.



(e) 256 by 256, 22 deg.

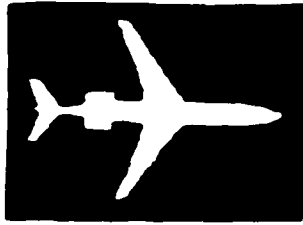


(c) 128 by 128, 45 deg.

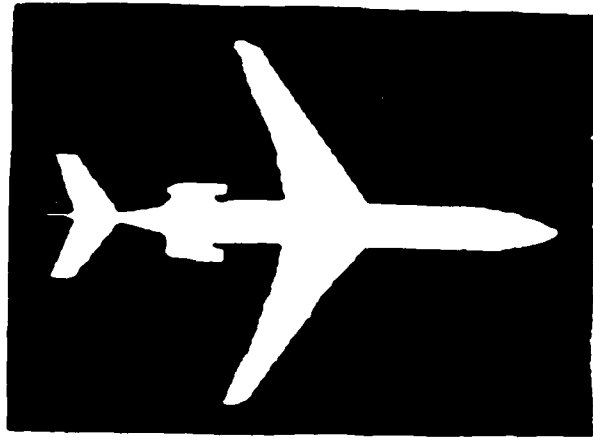


(f) 256 by 256, 45 deg.

Figure 7. Six transformations of a rectangle.



(a) 128 by 128, 0 deg.



(d) 256 by 256, 0 deg.



(b) 128 by 128, 22 deg.



(e) 256 by 256, 22 deg.



(c) 128 by 128, 45 deg.



(f) 256 by 256, 45 deg.

Figure 8. Six transformations of a digitized Boeing 727 outline.

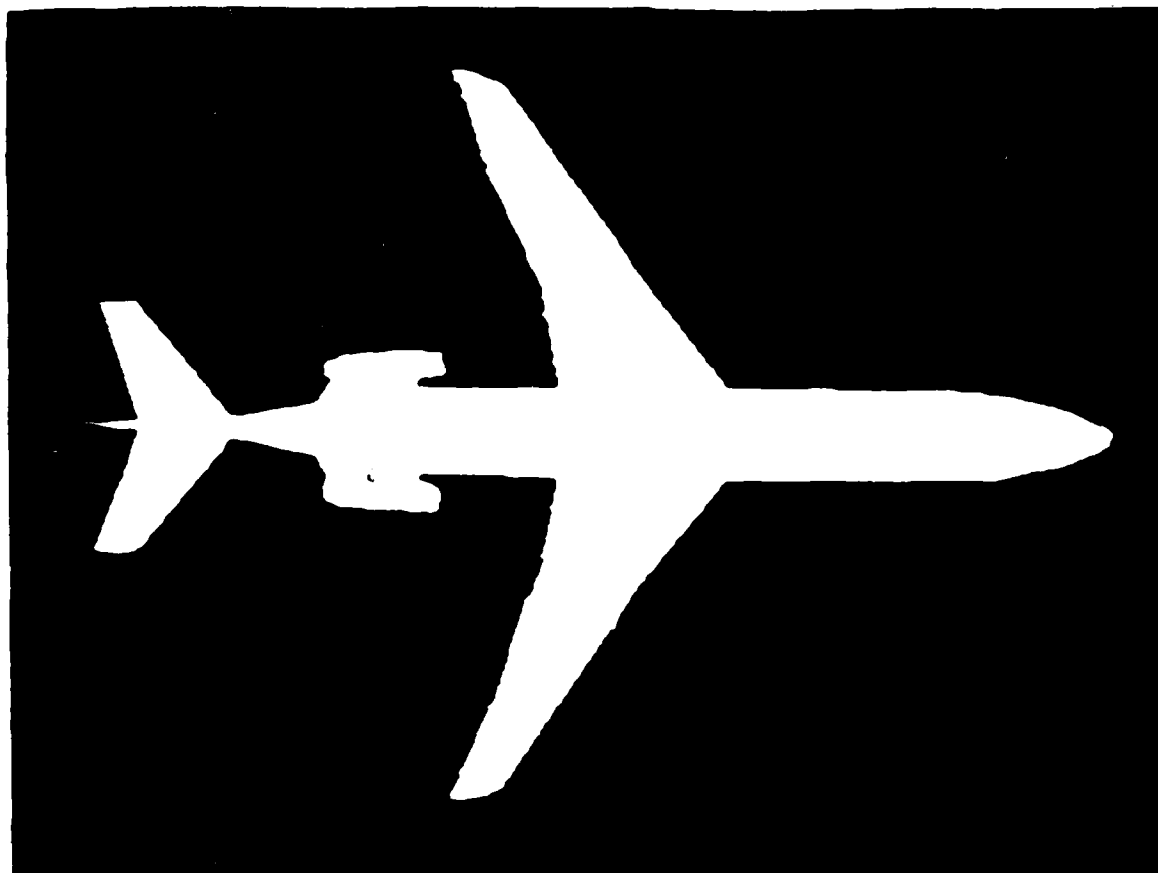


Figure 9. Plot of the 512 by 512 pixel space representation of a Boeing 727.

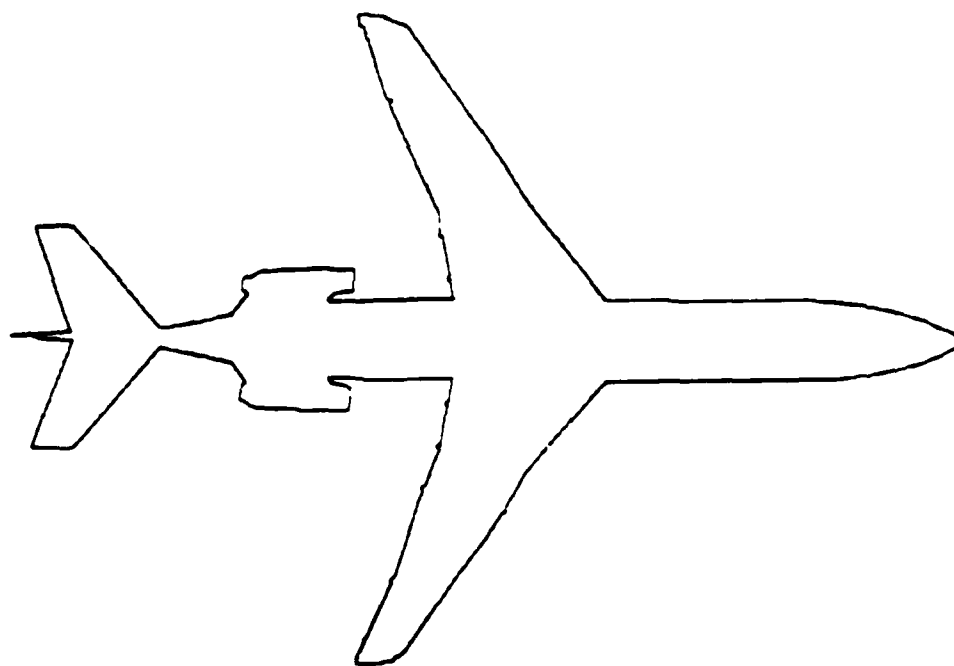


Figure 10. Boeing 727-200 (Boe727)

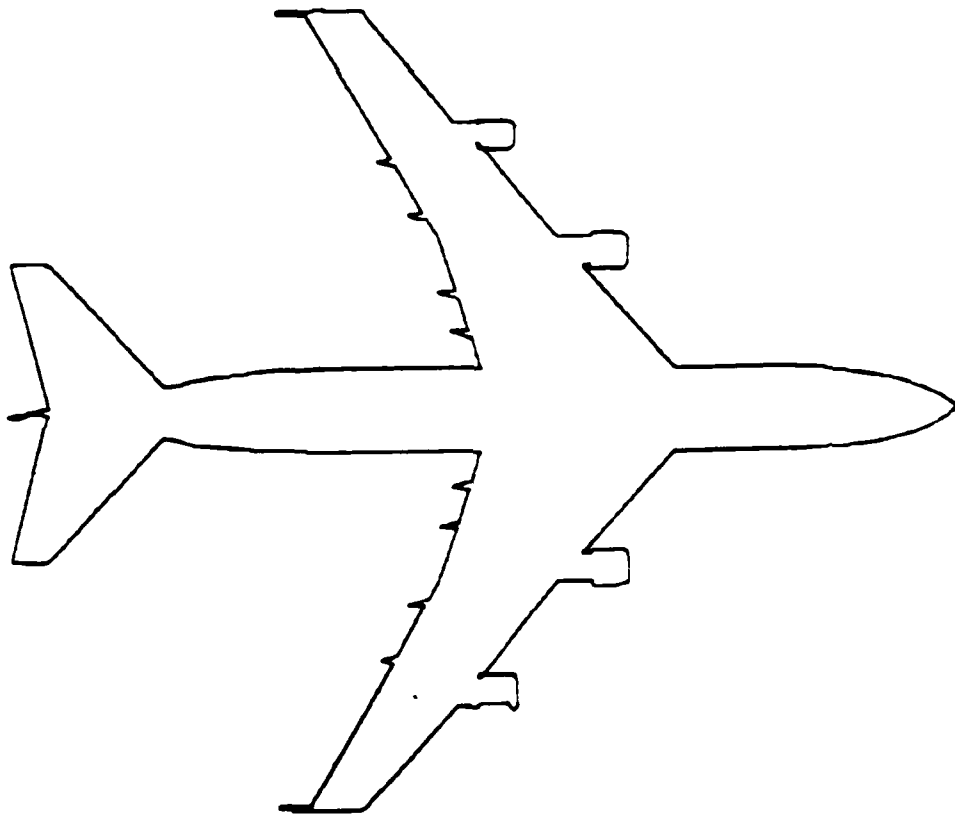


Figure 11. Boeing 747-200B (Boe747)

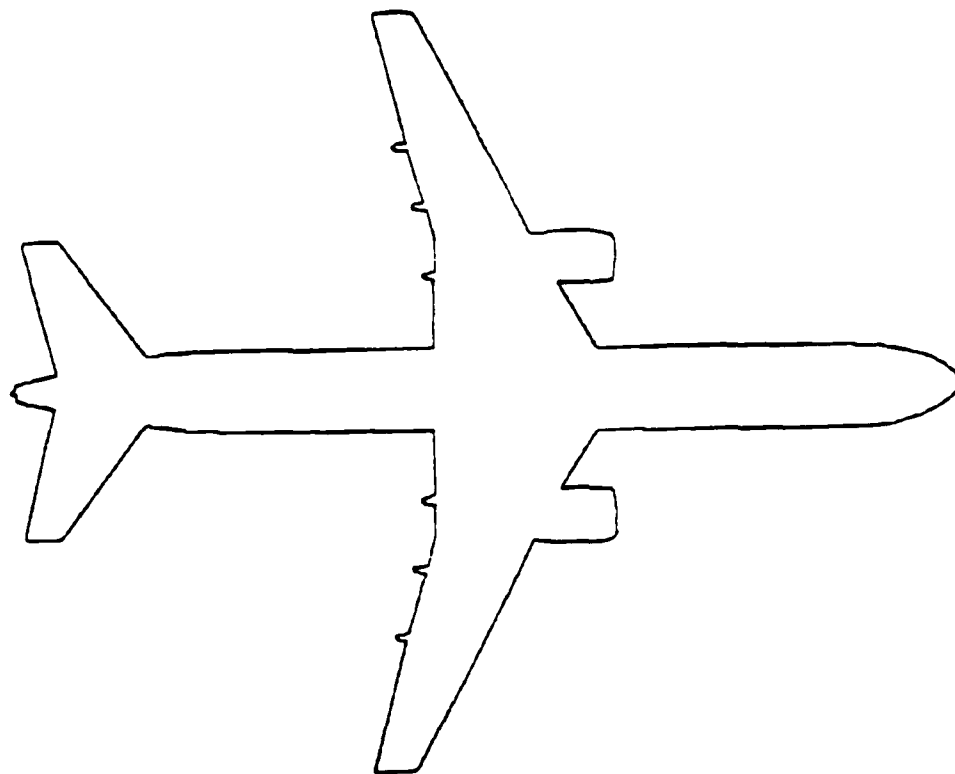


Figure 12. Boeing 757-200 (Boe757)



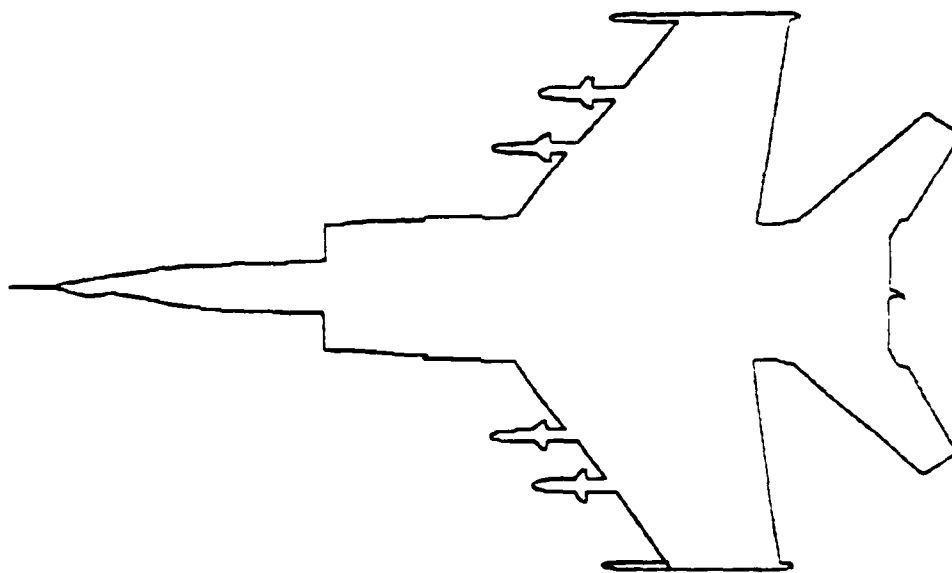


Figure 13. Mikoyan MiG-25 (MiG-25)

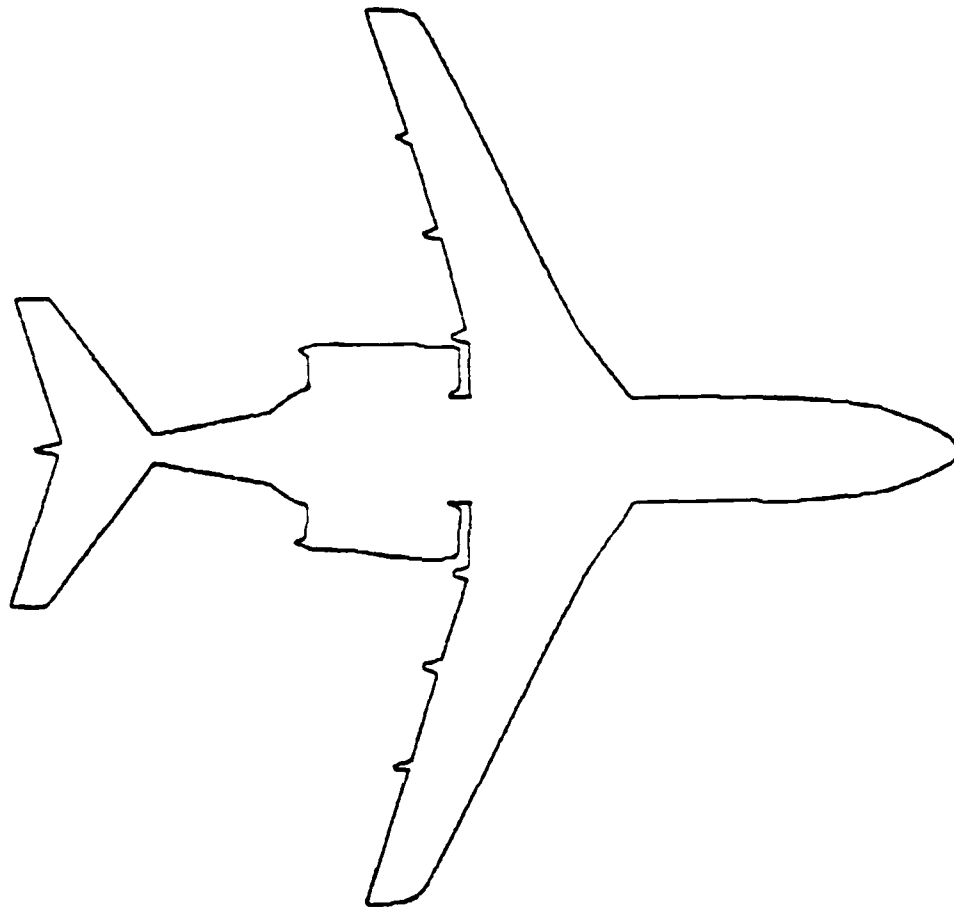


Figure 14. Cessna Citation III (Ccit3)

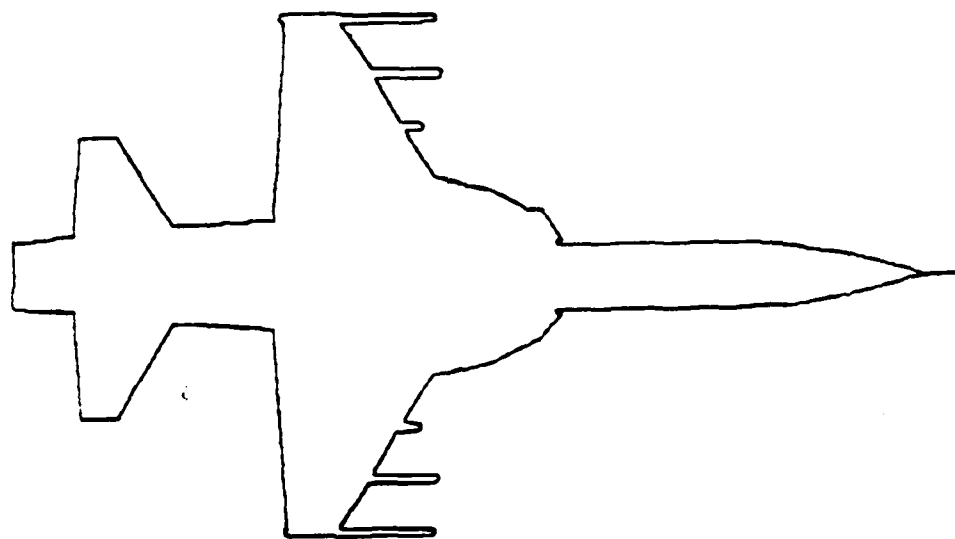


Figure 15. Northrop F-5E Tiger II (F-5E)

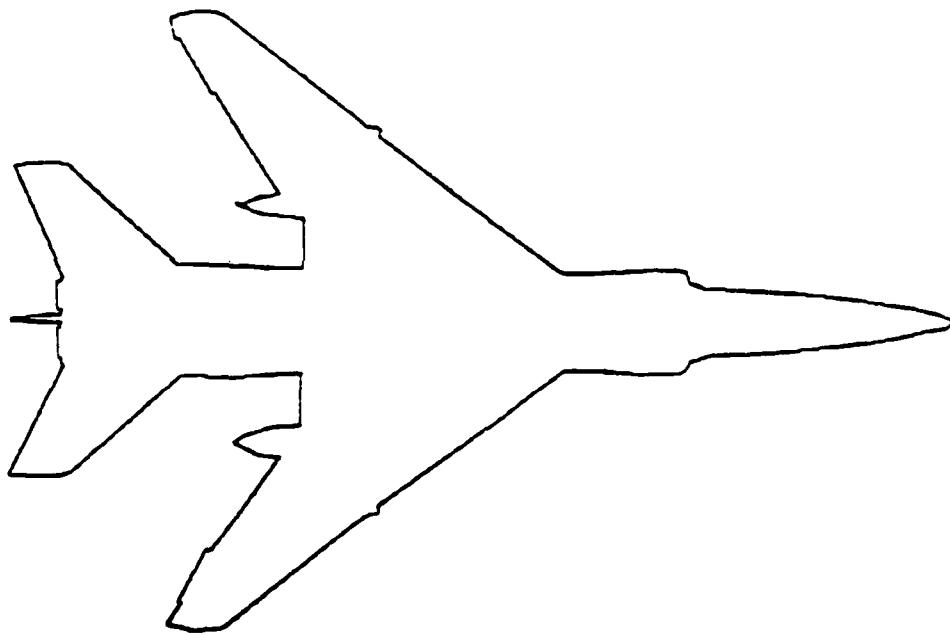


Figure 16. Tupolev Tu-28P (Tu-28P)

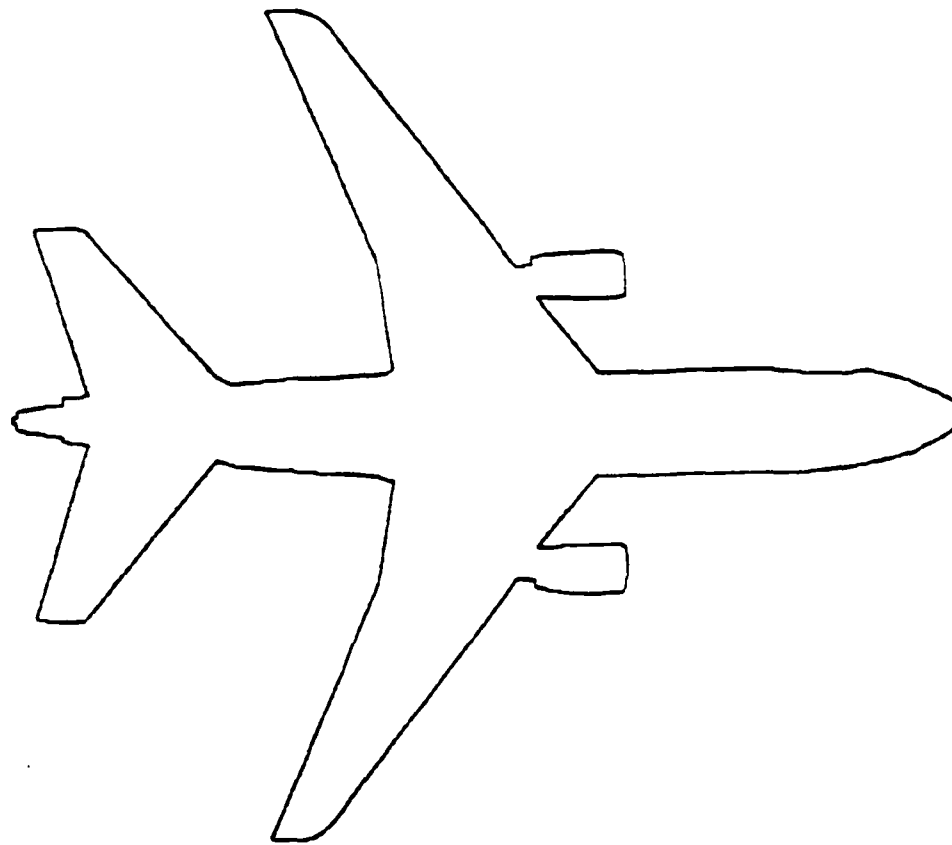


Figure 17. McDonnell Douglas DC-10 (DC-10)

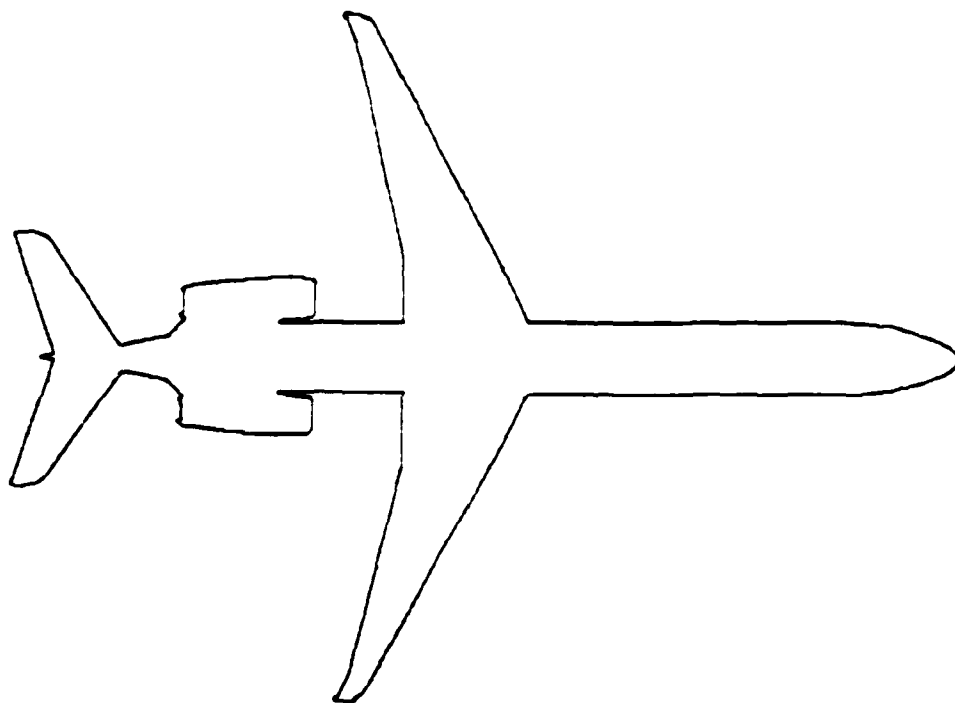


Figure 18. McDonnell Douglas DC-9 (DC-9)

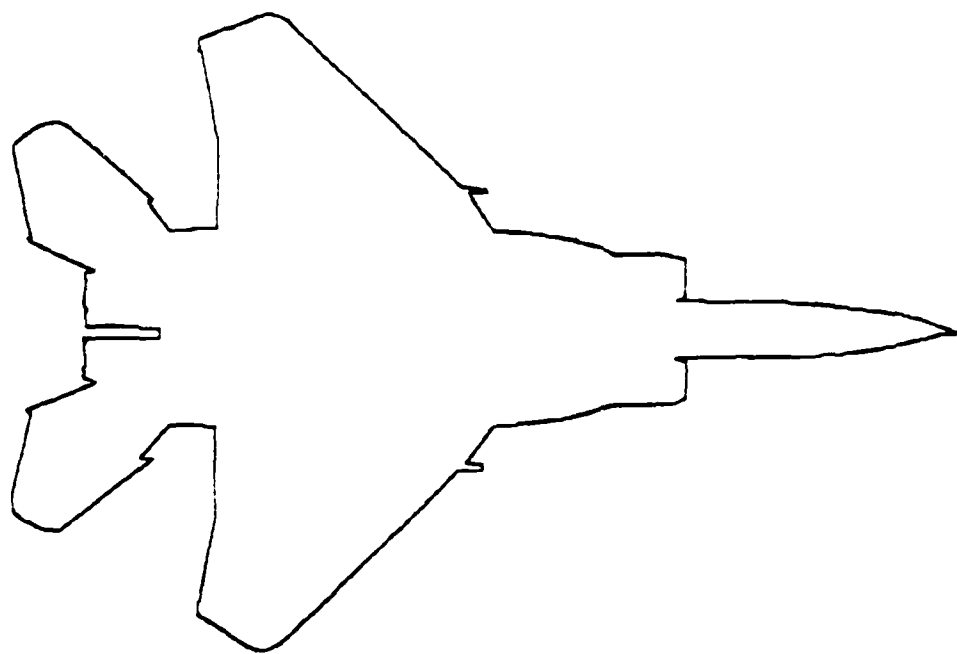


Figure 19. McDonnell Douglas F-15C Eagle (F-15C)

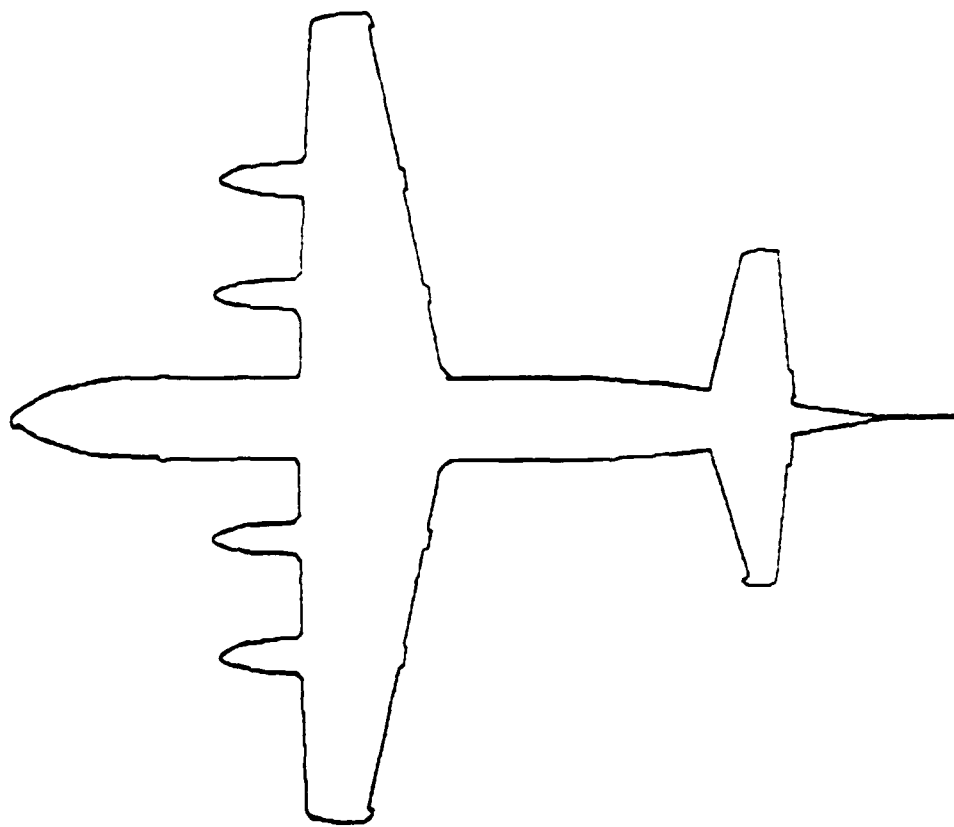


Figure 20. Lockheed P-3C Orion (LP-3C)



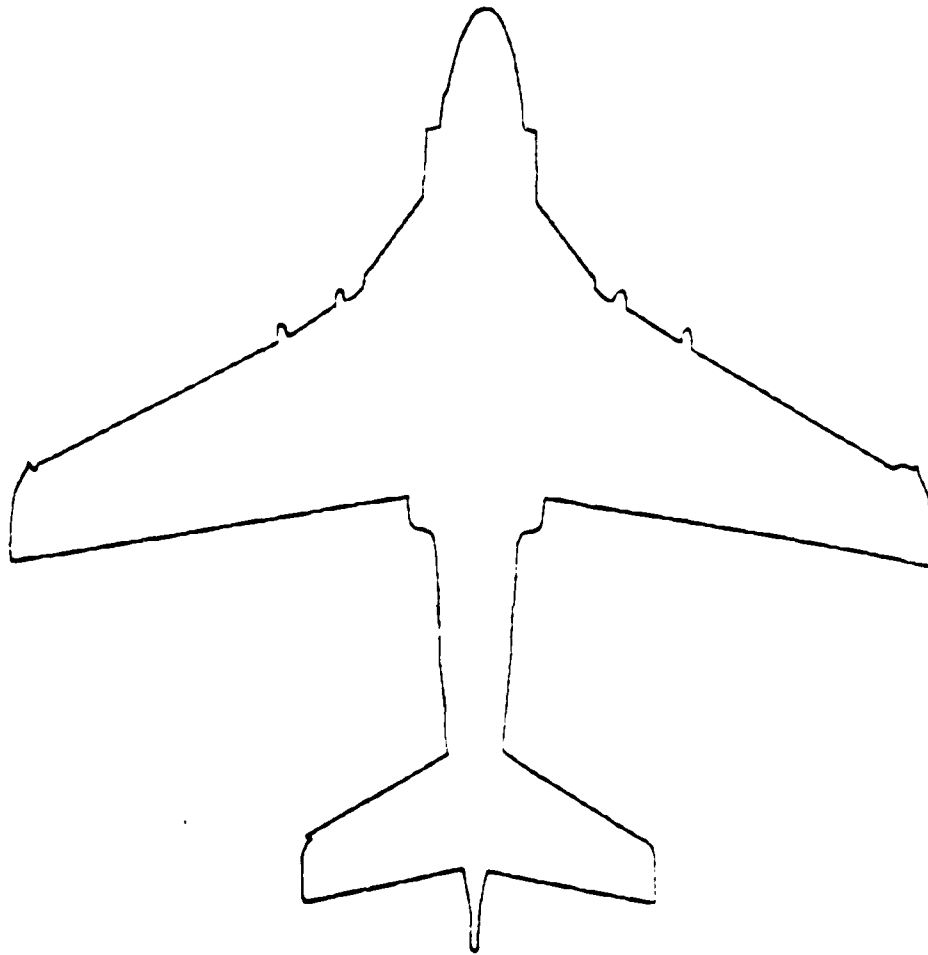


Figure 21. Grumman A6-E/TRAM (A6-E)

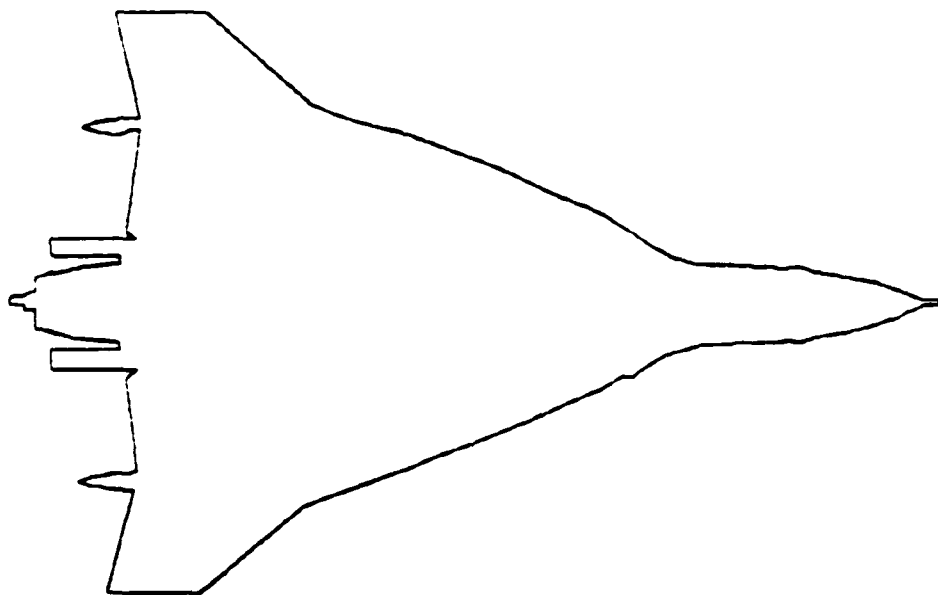


Figure 22. General Dynamics F-16XL (F-16XL)

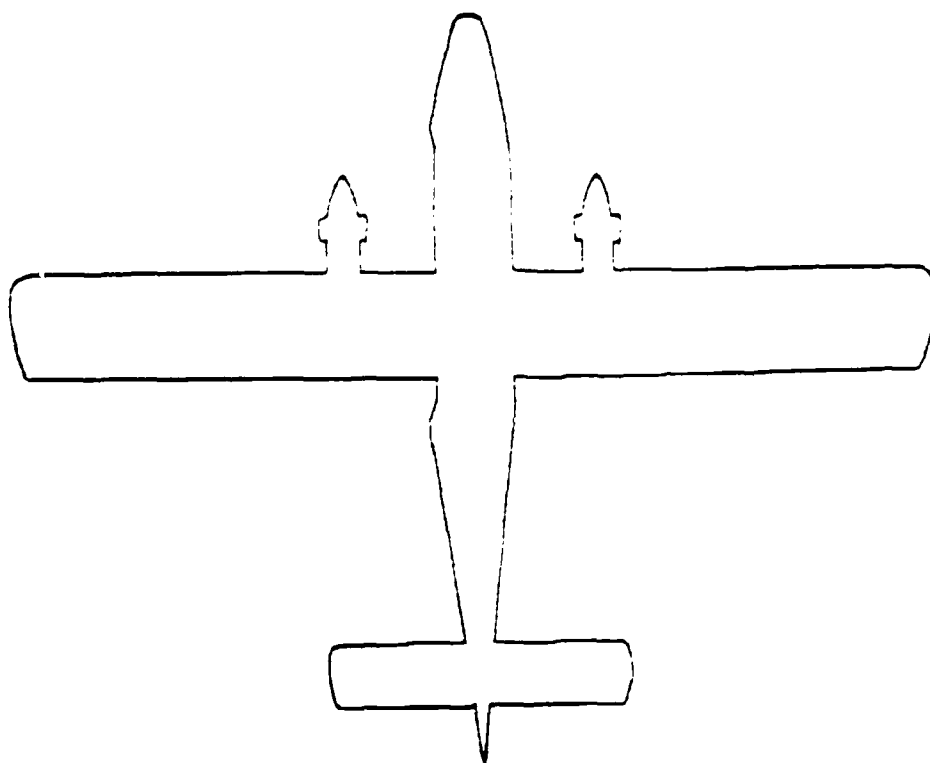


Figure 23. de Havilland DHC-6 Twin Otter (DHC-6)

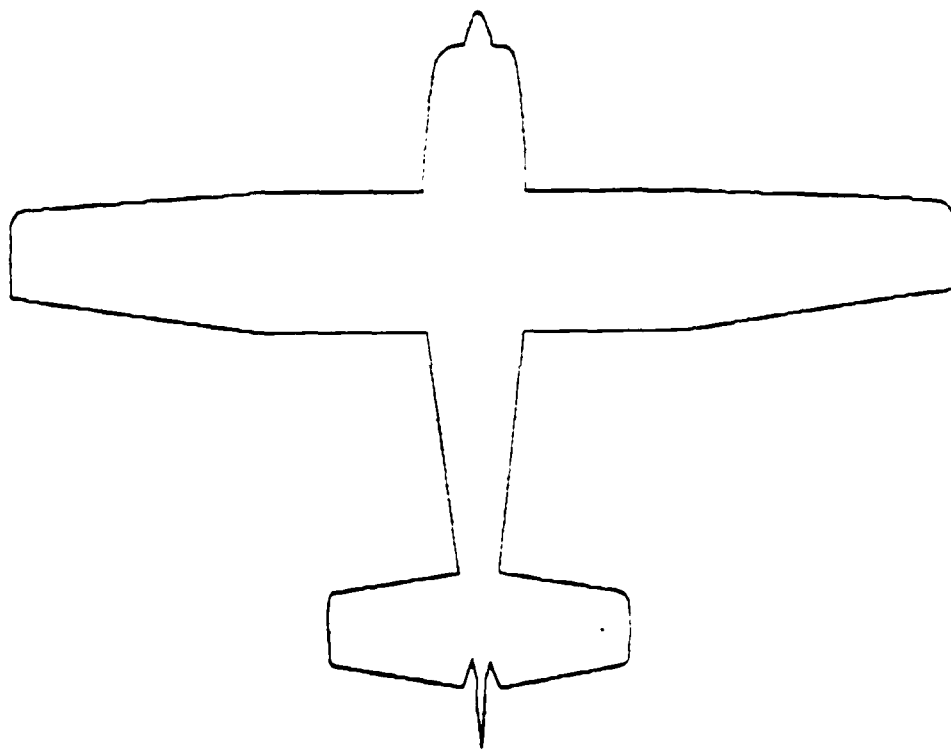


Figure 24. Cessna Skylane RG (CRG)

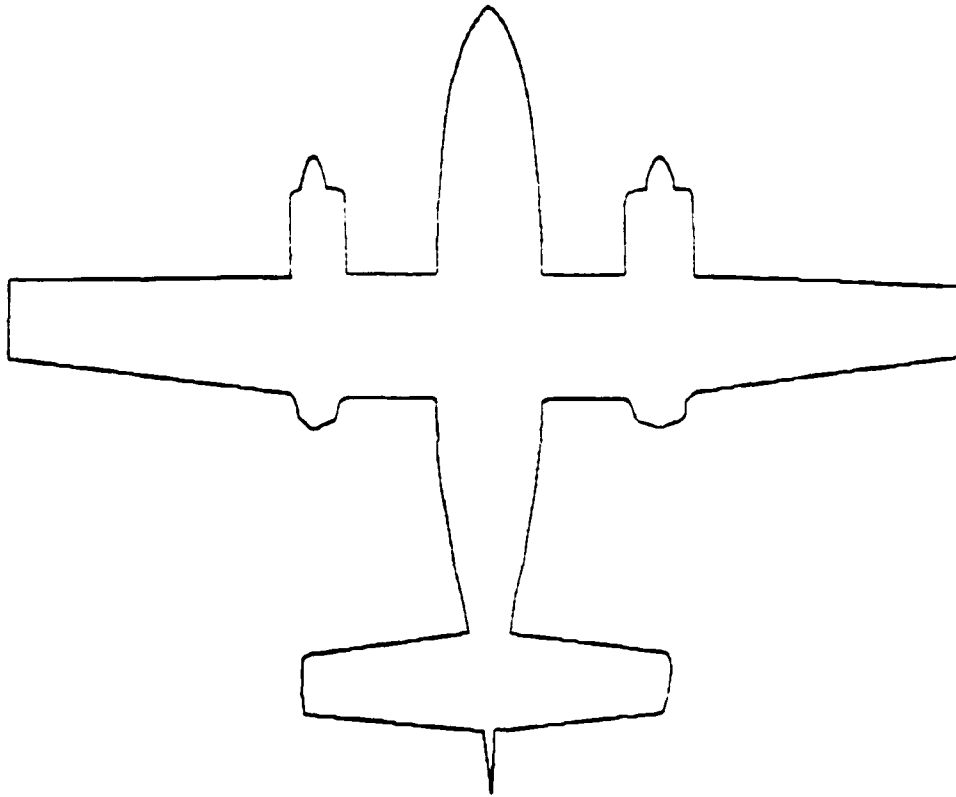


Figure 25. Cessna 402C (C402C)

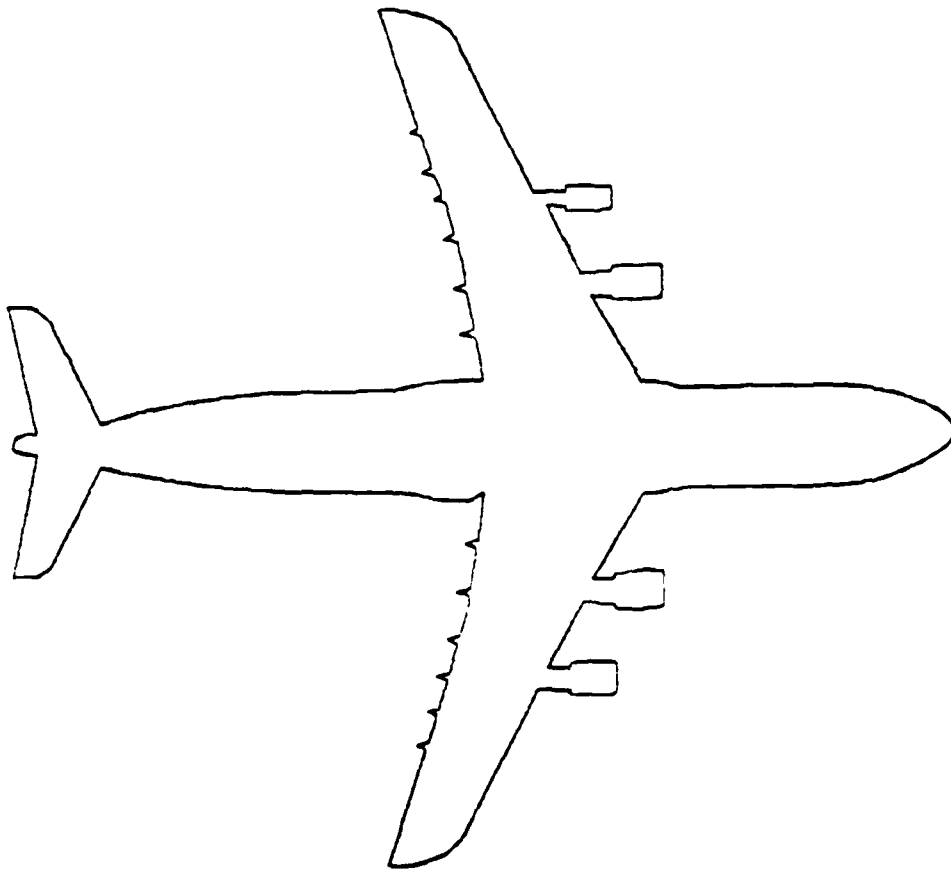


Figure 26. Lockheed C-5B Galaxy (C-5B)

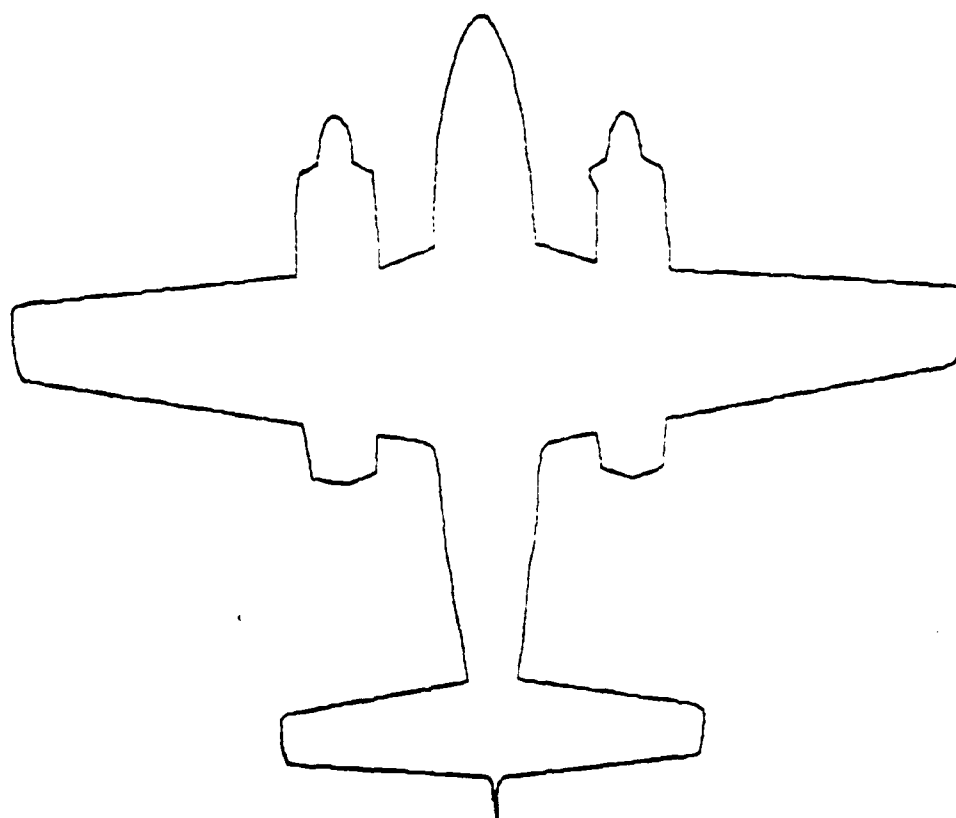


Figure 27. Piper Chieftain (Piper)

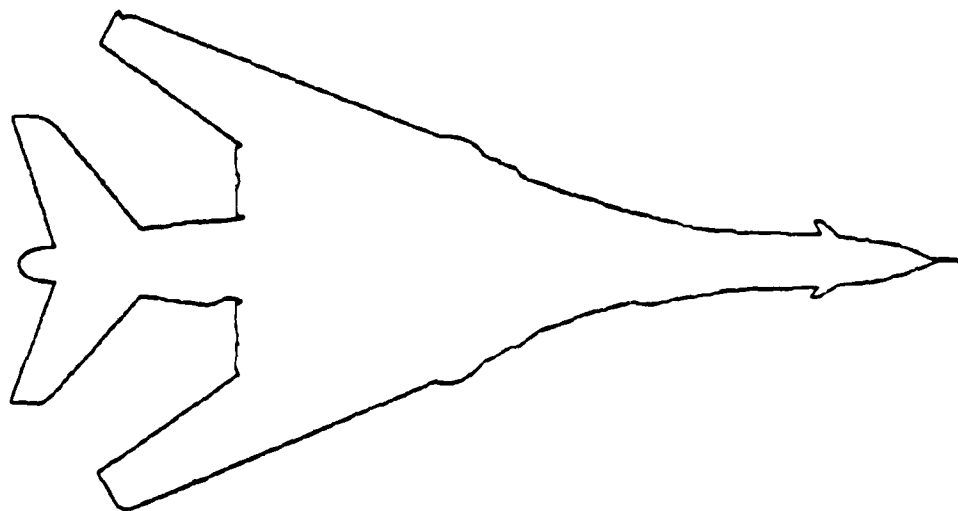


Figure 28. Rockwell International B-1B (B-1B)



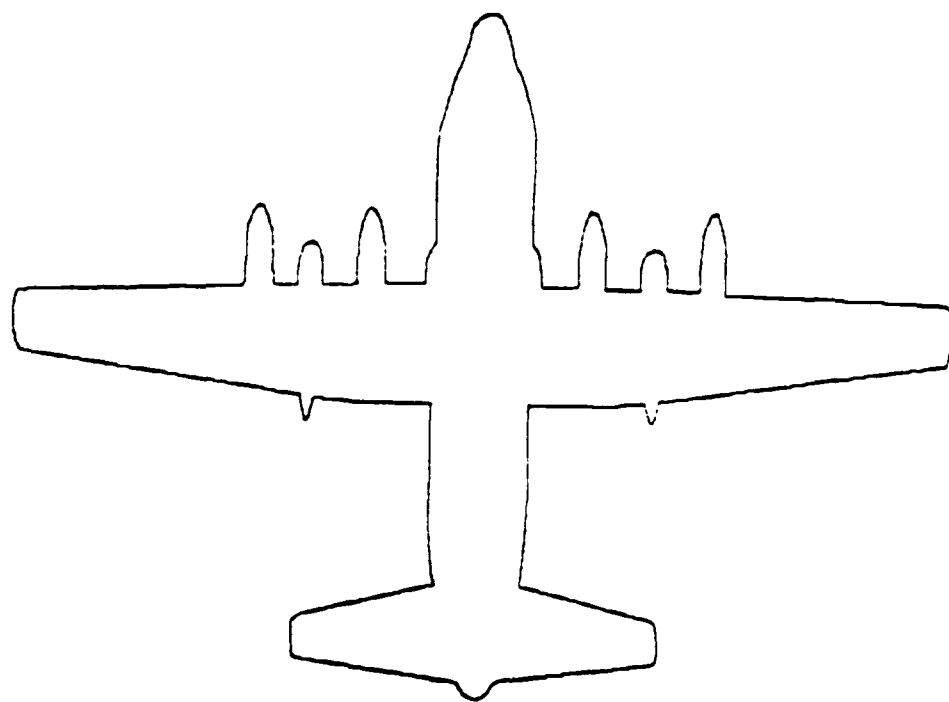


Figure 29. Lockheed C-130E Hercules (Herc)

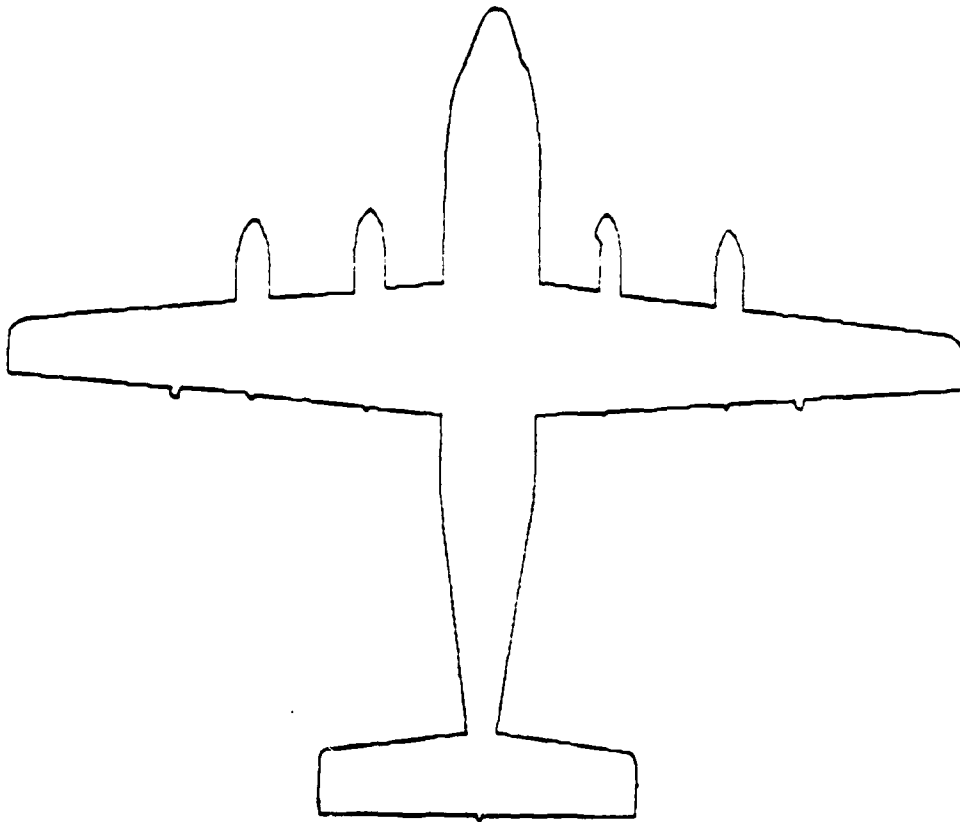


Figure 30. de Havilland DHC-7 Dash 7

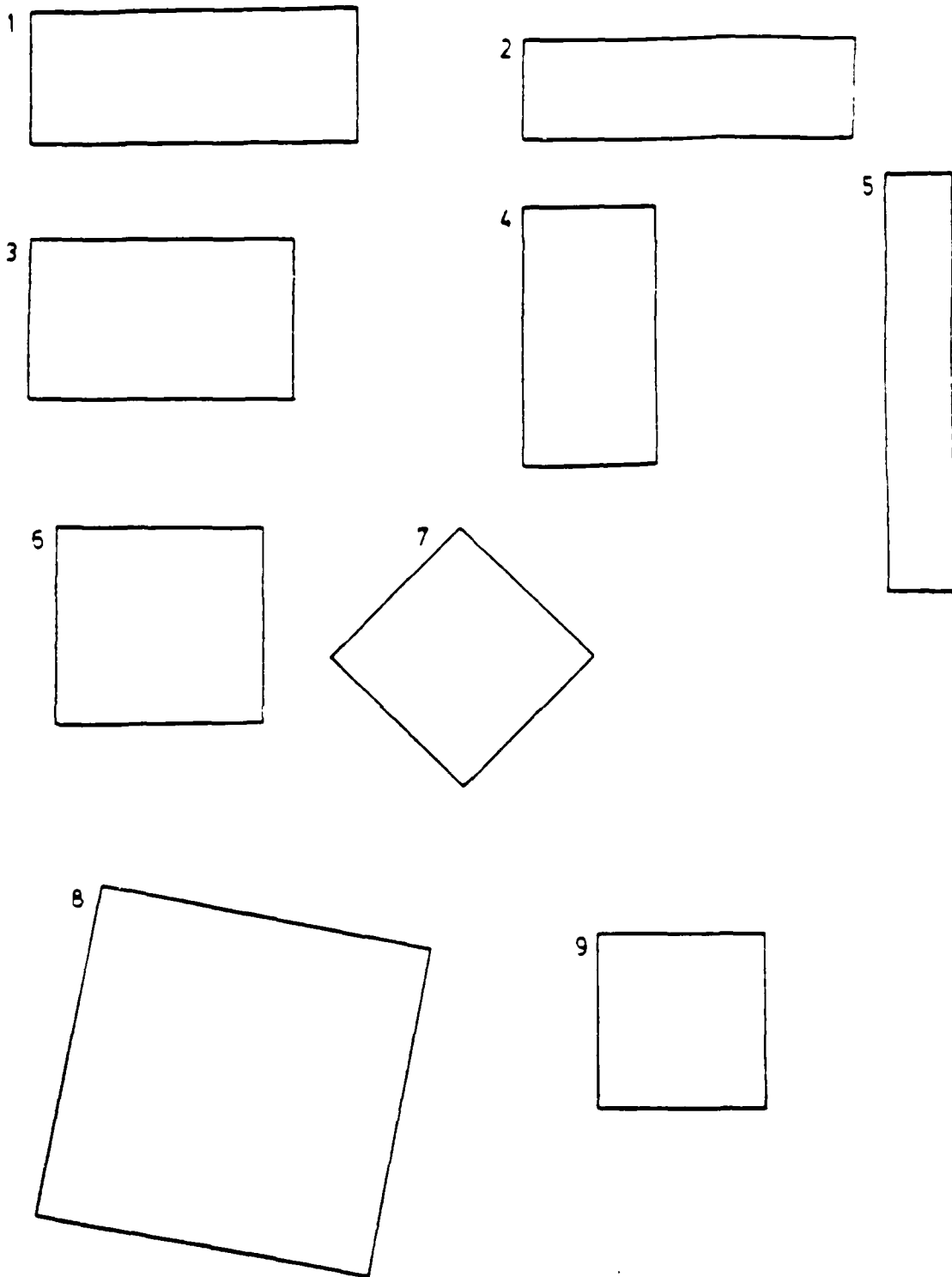


Figure 31. Basic shapes used to test the fuzzy clustering of invariant attributes.

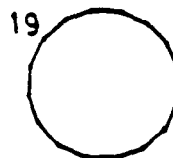
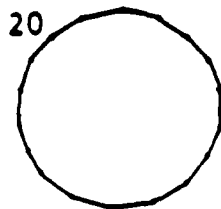
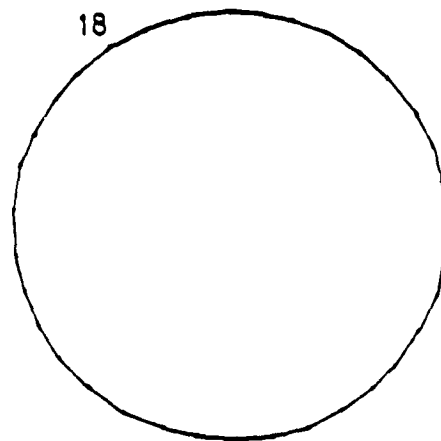
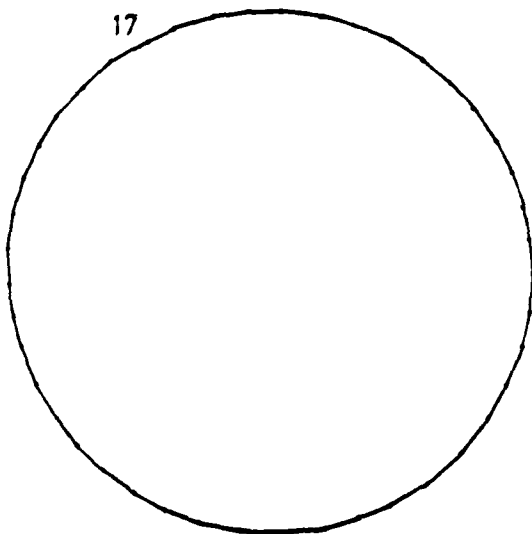
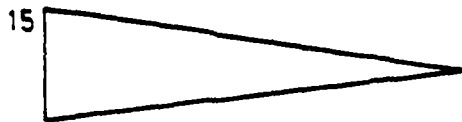
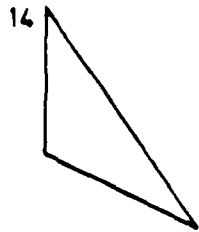
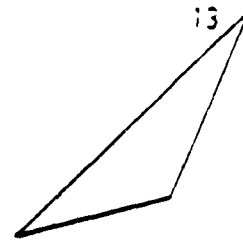
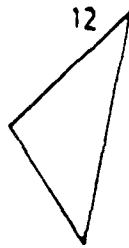
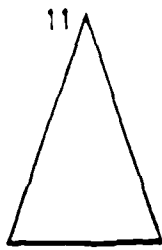
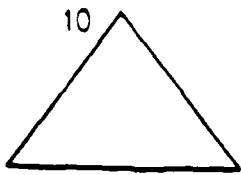


Figure 31, continued



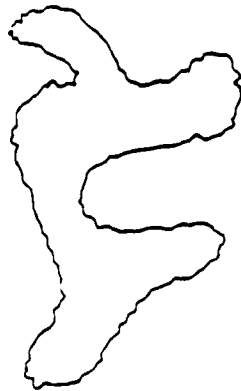
(a) Original blob



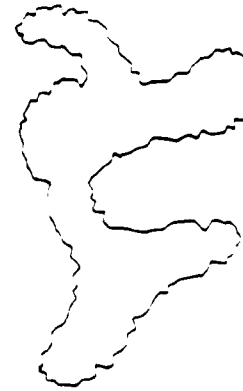
(b) Noise = 12.5



(c) Noise = 25



(d) Noise = 50



(e) Noise = 75

Figure 32. Blob1 with random noise added.



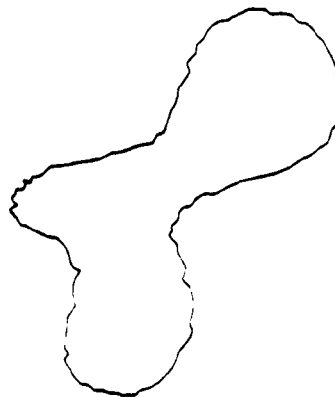
(a) Original blob2



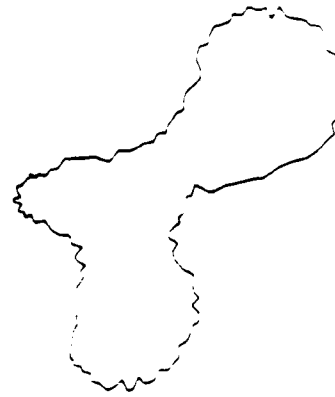
(b) Noise = 12.5



(c) Noise = 25



(d) Noise = 50

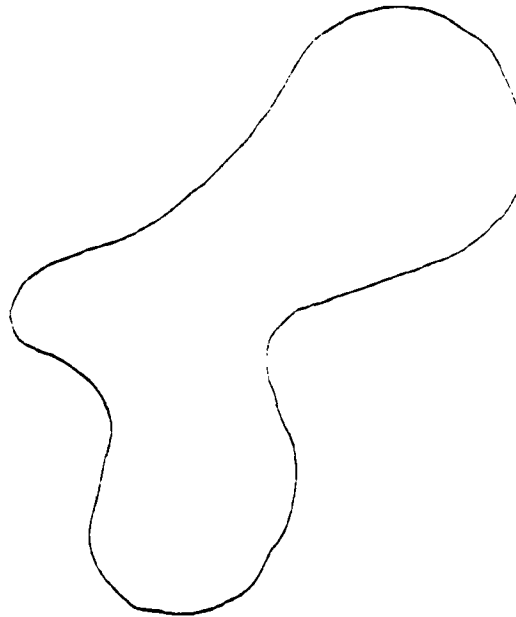


(e) Noise = 100

Figure 33. Blob2 with random noise added.



(a) Blob3



(b) Blob4

Figure 34. Blob3 and blob4 (slightly changed versions of blob1 and blob2).

## 7 SPEECH UNDERSTANDING RESEARCH

Report submitted by:  
Harvey Rhody

RIT Research Corporation  
75 Highpower Road  
Rochester, NY 14623

### TABLE OF CONTENTS

I.	Introduction .....	702
A.	Research activities .....	702
II.	Research Activities .....	703
A.	Application of expert systems methods .....	706
III.	Speech Scientist's Workbench .....	708
IV.	Tasks for the Coming Year .....	710
A.	Tasks .....	711
	Bibliography .....	712



## I. INTRODUCTION

The purpose of the speech understanding project is to develop methods for a large-vocabulary, speaker independent continuous time speech understanding system. The project is to be carried out over a period of five years covering fiscal years 1985 through 1989. A related project, "Speech Analysis Based on a Model of the Auditory system" was carried out under the RADC Post-Doctoral program in FY-1984.

The Post-Doctoral research program produced a signal processing model of the auditory system. It was shown that speech processing algorithms based on this model are capable of resolving speech parameters in both the time and frequency domains with greater accuracy than was heretofore possible.

The goals of the work for FY 1985 have been to:

1. Develop the signal processing algorithms, based on the auditory system model, which can be used to extract descriptive parameters from speech.
2. To develop a paradigm for phoneme segmentation and preliminary identification.
3. To carry out a literature search and plan a study of pattern-matching architecture, to be carried out in FY 1986.

At the time of this writing, the FY 1985 project has been approximately 75% completed. The tasks listed above are nearly complete, as described below. Their completion is anticipated in the near future.

## II. RESEARCH ACTIVITIES

It has been found that the auditory system model can be used as the basis of a speech analysis tool which has superior resolution of both time and frequency events. The system provides an accurate pitch-synchronous time indication, which is the key to selecting the times at which to measure the speech formant frequencies.

It has been found that a pitch-synchronous analysis method can be used to resolve formant locations with a resolution that is five times greater than is available with traditional formant extraction techniques. It is believed that this additional resolution will lead to improved speech segmentation and phoneme identification.

The purpose of phoneme identification is to reduce the dimensionality of the speech signal to manageable proportions. The data rate needed to represent the phoneme sequence can be estimated as 6 bits per phoneme at an average of 20 phonemes per second, for a total of approximately 120 bits per second. The actual data rate may be four or five times as high because of the impossibility of identifying the correct phoneme with certainty at this level of the system.

The general nature of the decision-making system has been investigated. The current plan is to use a fuzzy-logic system. Such systems are characterized by the maintenance of lists of elements for each point in the sequence. These lists are searched systematically under higher-level control. In this case, the higher-level process would be seeking to construct reasonable words and phrases. The sequence of fuzzy phoneme sets would provide the raw material for the construction. A major goal, therefore, is to not reject correct phonemes while minimizing the size of each set. It is not necessary to rank-order the sets by probabilities (which are probably not available to the process). Word recognition is carried out by

combining phoneme possibilities from adjacent time segments and comparing the partial strings to phoneme-word dictionaries. It would be most helpful if this matching process could simply proceed left to right as the phoneme lists are presented. However, it is evident that the process must proceed in both directions in time. The two-directional process is imposed by the fact that words are of differing lengths and recognition in some regions will be more reliable than in others.

The word-sequences selection must be under the control of a process which contains a knowledge base of the structure of utterances for the given context. This knowledge base can make use of an appropriate grammar and lexicon for the context. The content and structure of the knowledge base may be tailored according to application.

The structure of particular phonemes does not depend upon the higher-level structure of the utterance. However, the structure in which they are embedded can, and should, be exploited in the decoding process. The spoken expression imposes a structure that must be observed by the phoneme string over an extended interval, much as a convolutional code imposes a structure on a sequence of binary digits.

The set of phonemes that are used to build the utterance are largely invariant from one person to another. In large measure, the phoneme set does not even depend on the language of the speaker. It is expected that the effect of language will be second-order, in which one or two phonemes are added to or deleted from the set and the use frequencies are changed somewhat. However, word recognition will be highly dependent on nationality, region, language and context.

Phoneme recognition in continuous speech must take account of the inter-phoneme interactions that occur in the speech production process. The templates of isolated phonemes that have been studied in classical speech science are only of limited use because of the changes that take place in a dynamic system. A phoneme identification system for continuous speech must be based on a dynamic model. This means that the characterization of phonemes by signal parameters that are extracted from the speech will be dependent on the surrounding phonemes and their parameters. Once the phonemes have been identified, their labels will be the same as if they were spoken in isolation; however, the process of extracting them from a background of dynamic parameter variations is far more complicated than extracting them from a static parameter background.

A dynamic model of speech production views the production process as one of continuous motion of the vocal-tract articulators. This produces an organized interaction of the phonemic elements in which the parameters of adjacent elements are systematically blended across the phoneme boundaries. Thus, no static representation will be adequate for describing the phonemes by their parameter values.

The difficulty that is presented is not one of measuring the values of the phoneme parameters. Such parameters as the formant frequencies, intensity, duration, pitch and voice/unvoice can be measured. However, the phonemes must be identified in the context of other phonemes because the values of the parameters will be dependent on the surroundings. Thus, it is the interpretation which must be dynamic.

A substantial body of research exists on the characterization of phonemes in static situations. However, characterization in terms of dynamic movement of vocal-tract articulators is largely unexplored. A beginning point is presented in the article by Browman and Goldstein (1985). It is expected that the speech scientist's workbench, developed on the basis of the auditory model, will provide a tool which will allow this important investigation to be carried out expeditiously.

#### A. Application of Expert System Methods

It is generally recognized that speech recognition can be carried out successfully if a reasonably accurate phoneme string can be extracted from the voice waveform. The analysis system that is based on the auditory model provides an analytical tool for accurately computing the values of the important parameters of speech and tracking them continuously as they vary. In a sense, these parameters are the coordinate values that describe the speech in a specialized state space. What is missing is the conversion from the continuous parameter variations in this state space to a reliable segmentation into discrete time space and a reliable mapping of the segments into a phoneme sequence.

The general procedure is to separate the segmentation and identification problems. However, it is our view that this is not necessary. Moreover, we believe, this division will lead to suboptimal performance. Thus, we seek to map from a state space of continually varying speech parameters to the discrete phoneme sequence. It is our view that this discrete phoneme sequence preserves the essential information for speech recognition.

The basic problem, then, is to find a suitable speech state space and a mapping from that space to the phoneme sequence. The parameters that must be preserved are generally recognized. These are the speech formants, the pitch, the energy levels in various frequency bands, and certain binary information such as voice/unvoice indications. What is missing is the mapping from

combinations of parameter values and trajectories into phoneme sequences. The literature contains experimental and speculative information about parameter combinations that identify certain phonemes and discriminate them from other phonemes. This is mostly related to the identification of phonemes spoken in isolation. We have been unable to identify any work, be it authoritative or speculative, for the mapping from continuous speech into a discrete phoneme sequence. What is available in the literature are papers on the study of parameter variations that are useful in identifying particular phonemes.

A difficulty in pursuing this task is in evaluating the efficacy of various rules for identifying phonemes. Beyond that is the problem of collecting a large enough body of rules that are internally consistent to meet essentially all of the parameter combinations that are observed in continuous speech. This problem is compounded by our desire to make the identification procedure applicable to a wide range of speakers.

It is our intention to apply knowledge engineering tools too in this investigation. We plan to build a rule-based expert system for phoneme identification on the basis of speech parameter measurements. The system will make use of rules that are found scattered through the literature as well as rules that we create in the course of our investigation. This approach has a number of advantages.

1. It provides a structure for organizing the knowledge scattered through the literature on phoneme identification.
2. It works naturally with heuristic rules. Most of the rules found in the literature are heuristic.
3. It provides a tool which can be applied and evaluated in a convenient manner.

4. If the expert works well in any particular situation, then it can be asked which particular rules it used to make a decision. Thereby, the effective rules can be gleaned from the knowledge base.
5. Rules can easily be modified.
6. New rules can easily be incorporated.
7. The knowledge base is separated from the speech data base.
8. High-level tools for building the knowledge base are readily available.
9. Rules for speech segmentation will examine the same parameters that are used for phoneme identification. Thus, segmentation and identification can be made to work together in a coordinated manner.

The phoneme expert system will therefore be a research tool which will be used to find a useful rule base. The rule base that is constructed can then be incorporated in a special computational structure for real-time operation.

It is important to emphasize that this kind of knowledge base can be used with uncertain and ambiguous information and that it can provide phoneme identification with indications of the uncertainty involved. This is a suitable match to a fuzzy-logic methodology.

It is anticipated that the same methodology can be extended to cover word identification from fuzzy phoneme strings.

### III. SPEECH SCIENTIST'S WORKBENCH

A close association with Dr. Robert Houde of Speech Recognition Systems, Inc. of Rochester, New York has led to the acquisition of speech processing software to be used for the speech analysis portion of this project. This is needed for the analytical front end of the system. It does not involve any of the tools necessary for phoneme identification.

This analysis software is proprietary and is not available for

distribution or resale. It was developed by Speech Recognition Systems, Inc. for their own commercial applications. It is provided at no cost with the understanding that it will be used only for the research of this project. The software has been developed by a team of speech scientists and programmers at Speech Recognition Systems, Inc. with a budget and effort that is at least an order of magnitude beyond what is possible within this project. Their willingness to allow it to be used for this research helps us to overcome a very large obstacle.

The software is written in "C" to run on a Sun Microsystems, Inc. computer. This brand of computer has been purchased by RIT Research Corporation, and has been made available for use on this project.

The plan is to run the expert system engineering tools on the same system so that all of the software tools are available on the same system.

The Speech Scientist's Workbench contains the following software modules:

bargraph.c	Plots bar graphs.
client.c	Used for recording speech to a digital file and playing back speech from a digital file.
default.c	Default grapher, labeller, input and output function routine.
do_menu.c	Menu handler.
do_mouse.c	Handles mouse selection in option and graph windows.
do_opt.c	Routines to handle selected options.
filetype.c	Holds information used by graphers and measurers.
gcreate.c	Routines for creating and updating graphs.
gr_fif2.c	Draws an xy graph from a data file.
gr_form.c	Draws the form graph.



gram.c	Draws the spectrogram graph.
gram40.c	Calls gram.c with certain parameters.
gram_l.c	Draws linear predictive coder graph.
gramgen2.c	Generic spectrogram grapher; called by a few routines.
graph.c	Generic grapher, called by a few routines.
grfft.c	FFT grapher.
grfsw.c	Handles actions in the graph subwindow of speechtool.
intercept.c	Used for handling filtering parameter lists.
inval_sig.c	Handles the loading and updating of data.
label.c	Used for labeling graphs.
opt.c	Sets up all available options for speechtool.
point.c	Point grapher.
prtsw.c	Handles all events in the feedback window of speechtool.
rec.c	Does the recording of the original signal.
redraw.c	Performs refreshing and updating of displayed records.
sumdif.c	Line grapher.
ticks.c	Ticks grapher (the time axis).
tool.c	Main routine. From this routine, one accesses the analysis algorithms.

#### IV. TASKS FOR THE COMING YEAR

The tasks for the coming year are focused upon the completion of the speech science needed for this project and the initiation of the system studies. The speech science studies will utilize the

expert system modeling approach described above. The system studies will concentrate on the structure of the phoneme data base and the pattern search procedure to be used in phoneme identification. The design of a structure which will allow real-time phoneme matching is critical to the success of the recognition process. The system studies will also include an examination of system control structures, such as Hearsay II, with the view of making a control system selection in the following year of the project.

#### A. Tasks

1. Develop a set of characteristic parameter relationships for each phoneme.
2. Develop a dynamic production model based upon the dynamic articulator target model of the speech process.
3. Develop a speech phoneme segmentation algorithm.
4. Develop a phoneme-phoneme psycho-distance measure.
5. Examine candidate data-base structures and search algorithms for use in dynamic phoneme identification.
6. Examine system control structures for the integration of multi-level knowledge bases.

## BIBLIOGRAPHY

- Abramson, A.S. and Lisker, L., (1985), "Relative Power of Cues: F0 Shift Versus Voice Timing", *Phonetic Linguistics*, pp. 25-33.
- Baker, J.K., (1975), "The Dragon System: An Overview", *IEEE Trans. ASSP-23*, pp. 24-29.
- Bates, M., (1975), "The Use of Syntax in a Speech Understanding System", *IEEE Trans. ASSP-23*, pp. 112-117.
- Becker, R. and Poza, F., (1975), "Acoustic Processing in the SRI Speech Understanding System", *IEEE, Trans., ASSP-23*, pp. 416-426.
- Bedzek, J., (1981), Pattern Recognition with Fuzzy Objective Function Algorithms, Plenum.
- Blunstein, S.E. and Stevens, K.N., (1979), "Acoustic Invariance in Speech Production: Evidence From Meas. of Spectral Char.", *JASA*, Vol. 66, No. 4, pp. 1001-1017.
- Bobrow, D.G. and Collins, A., (1975), Representation and Understanding, Academic Press, New York.
- Bondarko, L.V., (1969), "The Syllable Structure of Speech and Distinctive Features of Phonemes", *Phonetica*, Vol. 20, pp. 1-40.
- Browman, C.P. and Goldstein, L.M., (1984), "Towards an Articulatory Phonology", Unpublished Manuscript.
- Browman, C.P. and Goldstein, L.M., (1985), "Dynamic Modeling of Phonetic Structure", *Phonetic Linguistics*, pp. 35-53.
- Chomsky, N. and Halle, M., (1968), The Sound Pattern of English, Harper & Row.
- De Mori, R., (1983), Computer Models of Speech Using Fuzzy Algorithms, Plenum.
- Eimas, P.D., (1985), "The Perception of Speech in Early Infancy", *Scientific American*, pp. 46-52.
- Flanagan, J.L., (1972), Speech Analysis, Synthesis and Perception, (Second Edition), Springer-Verlag.
- Flanagan, J.L. and Rabiner, L.R., (1973), Speech Synthesis-Benchmark Papers, Dowden, Hutchinson & Ross.
- Fowler, C.A., (1983), "Converging Sources of Evidence on Spoken and Perceived Rhythms of Speech...", *J. Exp. Psy.-General* V. 112, pp. 386-412.
- Fromkin, V.A., (1985), Phonetic Linguistics, Essays in Honor of Peter Ladefoged, Academic Press, New York.

- Hawley, M.E., (1977), Speech Intelligibility and Speaker Recognition -- Benchmark Papers, Dowden, Hutchinson & Ross.
- Jakobson, R., Fant and Halle, (1969), Preliminaries to Speech Analysis: The Distinctive Features and Their Correlates, MIT Press.
- Judson, L. and Weaver, A., (1965), Voice Science, Appleton-Century Crofts.
- Kates, James M., (1983), "An Auditory Spectral Analysis Model Using the Chirp Z-Transform", IEEE ASSP-31, No. 1, pp. 148-156.
- Keidel, Wolf D., et al., (1983), The Physiological Basis of Hearing, Thieme-Stratton.
- Kelso, J.A.S., et al., (1985), "A Qualitative Dynamic Analysis of Reiterant Speech Production: Phase Portraits...", JASA, Vol. 77, pp. 266-280.
- Malmberg, Bertil, (1968), Manual of Phonetics, American Elsevier.
- Newell, Allen (1975), "A Tutorial on Speech Understanding Systems", Speech Recognition (Reddy), pp. 3-54.
- Parkins, Charles W. and Anderson, Samuel W., (1983), Cochlear Prostheses: An International Symposium, New York Academy of Sciences.
- Parkins, Charles W., et al., (1983), "A Fiber Sum Modulation Code for a Cochlear Prosthesis", Cochlear Prostheses: Int. Symp., 22. 490-501.
- Parush, A., et al., (1983), "A Kinematic Study of Lingual Coarticulation in VCV Sequences", JASA, Vol. 74, pp. 1115-1125.
- Potter, Ralph K., Kopp and Kopp, (1966), Visible Speech, Dover.
- Reddy, D.R., (1976), "Speech Recognition by Machine: A Review", IEEE Proc., Vol. 64, pp. 501-531.
- Reddy, D. Raj, (1975), Speech Recognition, Academic Press, New York.
- Ruske, G., (1982), "Auditory Perception and Its Application to Computer Analysis of Speech", Computer Models for Percept...
- Saltzman, E.L. and Kelso, J.A.S., (1983), "Skilled Actions: A Task Dynamic Approach", Haskins Labs, SR-76, pp. 3-50.
- Seneff, Stephanie, (1985), "Pitch and Spectral Analysis of Speech Based on an Auditory Synchrony Model", MIT-Res. Lab. of Electronics.
- Sussman, H.M., et al., (1973), "Labial and Mandibular Dynamics During the Production of Labial Consonants...", JSHR, Vol. 16, pp. 397-420.

Tuller, B., et al., (1982), "Interarticulator Phasing as an Index of Temporal Regularity in Speech", *J. Exp. Psy: Human Perception*, pp. 460-472.

Umeda, N., (1975), "Vowel Duration in American English", *JASA*, Vol. 58, pp. 434-455.

Zwicker, E., et al., (1979), "Automatic Speech Recognition Using Psychoacoustic Models", *JASA*, Vol. 65, No. 2 (1979), pp. 433-455.

Zwicker, E. and Terhardt, E., (1974), Facts and Models in Hearing, Springer-Verlag.

## 8 TIME-ORIENTED PROBLEM SOLVING

Report submitted by:  
James F. Allen

Computer Science Department  
University of Rochester  
Rochester, NY 14627

### TABLE OF CONTENTS

Summary .....	716
Description of Research Accomplished .....	716
1) A Concise Interval-Based Theory of Time .....	716
2) Planning in Uncertain Worlds .....	716
3) A Theory of Plan Recognition .....	717
Future Research .....	717
Appendix 8-A: A Common Sense Theory of Time .....	719
Appendix 8-B: A Model of Naive Temporal Reasoning .....	731
Appendix 8-C: Toward a Theory of Plan Recognition .....	749
Appendix 8-D: A Formal Logic that Supports Planning with a Partial Description of the Future .....	817

# Annual Report of Project: Time-Oriented Problem Solving

Air Force Contract No F30602-85-C-0008

PI: James F. Allen, Computer Science Department, University of Rochester

November 1985

## Summary

In the last year we have made progress in our research on time-oriented problem solving in several areas. We developed a new theory of time based on temporal intervals that is considerably simpler and more elegant than our previous theory, and which subsumes and extends the latter. We also have made considerable progress on the development of a formal model for problem solving in temporally rich, uncertain worlds based on the combination of our temporal logic and existing logics of counterfactuals. We have also shown how plan recognition in simple worlds (e.g., the blocks world) can be formally related to and derived from a theory of planning in those worlds, thereby relating planning and plan recognition in an intuitively satisfying way. Finally, we are close to completing the conversion of our HORNE system, the general inference engine upon which our planning and plan recognition systems are built, into Common Lisp on the Symbolics machines from Franz Lisp on a VAX. Testing of the new system is expected to be complete on December 1st.

## Description of Research Accomplished

A detailed description of the research completed is described in the attached papers and technical reports. Here we give a short description of each of the major results.

### 1) A Concise Interval-Based Theory of Time

The literature on the nature and representation of time is full of disputes and contradictory theories. This is surprising since the nature of time does not cause any worry for people in their everyday coping with the world. What this suggests is that there is some form of common sense knowledge about time that is rich enough to enable people to deal with the world, and universal enough to enable cooperation and communication between people. In the last year, we have developed such a theory. We have an axiomatic theory of time in terms of intervals and the single relation MEET. We have shown that this axiomatization subsumes Allen's interval-based theory, and have extended the theory by formally defining the beginnings and endings of intervals, which are shown to have the properties we normally would associate with points. We distinguished between these point-like objects and the concept of *moment*, i.e., non-decomposable intervals, as hypothesized in discrete time models. Finally, we have examined the theory in terms of each of several different models, including continuous and discrete models, and shown that our logic is consistent with both.

### 2) Planning in Uncertain Worlds

In this project, Allen's interval-based temporal logic has been extended with a counterfactual-like modality, called IFTRIED that can be used to describe what can

and cannot be done by the planning agent. This allows us to represent planning problems where the robot has a practical description of the past, present, and expected future, and its goal is to bring about a set of desired future conditions. We show how the IFTRIED modality can be used to represent knowledge about what one can and cannot do. It also can be used to reason about the interaction of two concurrent actions, and to specify conditions under which the two actions can be executed together. A formal semantics for this logic is in progress.

### 3) A Theory of Plan Recognition

The problem of recognizing an agent's plans arises in many contexts in work in artificial intelligence. The plan recognition techniques suggested in the literature are rarely formally justified. We have developed a theory of plan recognition as a special kind of non-monotonic reasoning, and have demonstrated how formal techniques developed for such reasoning--namely, circumscription and minimal entailment--can be used in plan recognition. In this way, a theory of plan recognition has been derived directly from a theory of planning. This development suggests that the processes of planning and of plan recognition are actually two different viewpoints of the same problem, namely, reasoning about actions and causality.

### Future Research

In the next year we plan to complete our formal theory of planning in uncertain worlds and use it to guide the development of a planning system on the Symbolics that can construct conditional plans to deal with uncertainty. This system will be able to reason about simple events in the future that are not actions of the planner.

Our work on a general action reasoner, subsuming both planning and plan recognition, will continue as well. We intend to develop the theory in a simple world where two agents are constructing and executing plans. We will develop a model of one of these agents that can construct plans, observe the actions of the other agent and infer the other's plans, and then use this information to modify the initial plan. We will examine situations where the agents must compete, must cooperate, or are indifferent to each other's goals.

Our initial domain will probably be a route planning world where one agent may plan to avoid or meet the other agent. Initially, we will keep the temporal and spatial aspects of this world to a minimum, and enrich the domain as the work progresses. An initial implementation of this system will be begun, and will incorporate as much of our previous planning system as is possible.



# A Common-Sense Theory of Time

James F. Allen and Patrick J. Hayes  
Departments of Computer Science and Philosophy  
University of Rochester  
Rochester, NY 14627

December 1984

## Abstract

The literature on the nature and representation of time is full of disputes and contradictory theories. This is surprising since the nature of time does not cause any worry for people in their everyday coping with the world. What this suggests is that there is some form of common sense knowledge about time that is rich enough to enable people to deal with the world, and which is universal enough to enable cooperation and communication between people. In this paper, we propose such a theory and defend it in two different ways. We axiomatize a theory of time in terms of intervals and the single relation MEET. We then show that this axiomatization subsumes Allen's interval-based theory. We then extend the theory by formally defining the beginnings and endings of intervals and show that these have the properties we normally would associate with points. We distinguish between these point-like objects and the concept of *moment* as hypothesized in discrete time models. Finally, we examine the theory in terms of each of several different models.

## Introduction

The literature on the nature and representation of time is full of disputes and contradictory theories. This is surprising since the nature of time does not cause any worry for people in their everyday coping with the world. What this suggests is that there is some form of common sense knowledge about time that is rich enough to enable people to deal with the world, and which is universal enough to enable cooperation and communication between people. In this paper, we propose such a theory and defend it in two different ways.

First, the theory is powerful enough to include the distinction between "intervals" (i.e., times corresponding to events with duration), and "points" (i.e., times corresponding to instantaneous events), as well as allowing substantial reasoning about temporal ordering relations (including the abilities described in [Allen, 1984]). In addition, it includes a formalization of the beginning and ending of events by introducing the corresponding beginning and endings of times. We show that beginnings and endings act in many ways like "points," yet can be distinguished from them.

Second, the theory has as allowable models a number of the temporal models that are suggested in the literature. This includes models that equate time with intervals and points on the real number line, models that hypothesize discrete time,

and any model which mixes real points and intervals. Our claim is that if our common-sense theory of time excluded any one of these models, then there would be no debate as to whether that model was valid, since in that case our own primitive intuitions on the matter would be extremely clear. We do make one restriction on the models considered: they must allow the possibility that two intervals *MEET*, which is defined as the situation where there is no time between the two intervals, and no time that the intervals share. The importance of this relationship for naive theories of time has been argued elsewhere (e.g., [Allen, 1983; 1984]), and so will not be defended again here. Even with this requirement, we shall see that substantially different models are possible.

One important intuition which guides us is that time is occupied by events. If the universe did not change, there would be no time. Any sort of event or happening which can be described or thought of has a corresponding time, and the universe of times consists of these. We will often appeal to this intuition, which notoriously sometimes indicates continuity and sometimes discreteness. (In particular, it is the source of the need to allow time intervals to be able to *MEET*.)

In Section I, we axiomatize a theory of time in terms of intervals and the single relation *MEETS*. It is then shown in Section II that this axiomatization subsumes the interval-based theory proposed in [Allen, 1983; 1984].

We then extend the theory in Section III by formally defining the beginnings and endings of intervals and show that these have the properties we normally would associate with points. In Section IV, a distinction is made between these point-like objects and the concept of *moment* as hypothesized in discrete time models. Finally, in Section V, we examine the theory in terms of each of several different models.

The proofs of the theorems are not included in this paper. Most of them are fairly straightforward. Where this is not the case, we try to give an intuitive argument of why it is true. All the proofs are included in a longer version of this paper to appear as a forthcoming technical report.

## I. An Axiomatization of Interval Time

We start the formal development by positing a class of objects in our ontology that we shall call *TIMES*. These are intended to correspond to our intuitive notion of when some event occurs. We do not, at this early stage, make any commitment as to whether all times are decomposable or not.

The essential requirement of our intuition above is that two time intervals can *MEET*. We will take *MEET* as our primitive relationship between times and show that we can constructively define the complete set of possible relationships between intervals in terms of *MEETS*. It can easily be shown that the only other of the thirteen relationships in terms of which all might be defined is *OVERLAPS*. (We are not yet sure whether *OVERLAPS* would do; certainly, if so, the reduction would be far more complex and not directly constructive. Other sets of relations can be defined and reduced to one or two; for example, Hamblin [1972] uses a relation we could define as *less-than-or-MEETS*.)

For example, we can define a relationship BEFORE to hold between intervals only if there exists an interval that spans some time between them. Thus

$$I \text{ BEFORE } J \iff \text{EXISTS } k . I \text{ MEETS } k \ \& \ k \text{ MEETS } J.$$

As a notational convenience, we shall abbreviate conjunctions such as the above into a chain, i.e.,  $I \text{ MEETS } k \text{ MEETS } J$ . We shall also use the abbreviations used in [Allen, 1983] for disjunctions between pairs of intervals. Thus " $J$  (o o i s f d)  $I$ " is shorthand for the formula

$$(J \text{ OVERLAPS } I) \text{ OR } (J \text{ OVERLAPPED-BY } I) \text{ OR } (J \text{ STARTS } I) \\ \text{OR } (J \text{ FINISHES } I) \text{ OR } (J \text{ DURING } I).$$

These relation names are all defined in Figure 1, below.

As another example,  $I$  equals  $J$  only if there are two intervals, one that meets both (at their beginning), and one that both meet (at their ending), i.e.,

$$I = J \iff \text{EXISTS } k, l . k \text{ MEETS } I \text{ MEETS } l \ \& \\ k \text{ MEETS } J \text{ MEETS } l$$

The other possible relationships are defined in Figure 1. By including the inverses of these relations in the obvious way, we have all thirteen relationships defined constructively in terms of MEET. Each entry defines the ordered relation between  $I$  and  $J$  ( $I$  BEFORE  $J$ ,  $I$  OVERLAPS  $J$ , etc.). The inverses are also between  $I$  and  $J$  and are equivalent to the original relationship between  $J$  and  $I$  (e.g.,  $I$  BEFORE  $J \iff J$  AFTER  $I$ , etc.). The small letters listed with each give the abbreviation for the relation that will be used later in some examples.

<u>Relation</u>	<u>Inverse</u>	<u>Definition</u>
BEFORE. b	AFTER. a	EXISTS k . I MEETS k MEETS J
=	=	EXISTS k,j . k MEETS I MEETS j & k MEETS J MEETS i
OVERLAPS. o	OVERLAPPED-BY. oi	EXISTS a,b,c,d,e . a MEETS I MEETS d MEETS e & a MEETS b MEETS J MEETS e & b MEETS c MEETS d
STARTS. s	STARTED-BY. si	EXISTS a,b,c . a MEETS I MEETS b MEETS c & a MEETS J MEETS c
FINISHES. f	FINISHED-BY. fi	EXISTS a,b,c . a MEETS b MEETS I MEETS c & a MEETS J MEETS c
DURING. d	CONTAINS. di	EXISTS a,b,c,d a MEETS b MEETS I MEETS c MEETS d & a MEETS J MEETS d

Figure 1: The Relationships Between I and J in terms of the MEET Relation

With this reduction, we can axiomatize the interval logic entirely in terms of the one relation, as follows.

The first two axioms are based on the intuition that each interval may MEET any other interval at a single "place." Intuitively, these axioms simply state that intervals have a unique beginning position and a unique ending position. As a consequence of this, if two intervals both MEET a third interval, then any interval that one MEETS, the other MEETS as well.

Axioms for Uniqueness of "Meeting Places":

(M1) ALL ij .  
(EXISTS k . I MEETS k & J MEETS k) =>  
(ALL l . I MEETS l <=> J MEETS l)

(M2) ALL ij .  
(EXISTS k . k MEETS I & k MEETS J) =>  
(ALL l . I MEETS l <=> j MEETS l)

The third axiom captures the notion of ordering. It simply states that given two "places" where two intervals meet, then these places are either equal or one precedes the other. This is axiomatized without referring to places as follows:

Ordering Axiom:

(M3) ALL  $i, j, k, l$ .  
 $(i \text{ MEETS } j \ \& \ k \text{ MEETS } l) \Rightarrow$

- 1)  $(i \text{ MEETS } l) \text{ XOR}$
- 2)  $(\text{EXISTS } m . i \text{ MEETS } m \text{ MEETS } l) \text{ XOR}$
- 3)  $(\text{EXISTS } n . k \text{ MEETS } n \text{ MEETS } j)$

In other words, we have exactly three possible cases, shown in Figure 2, for any four intervals  $i, j, k$ , and  $l$ .

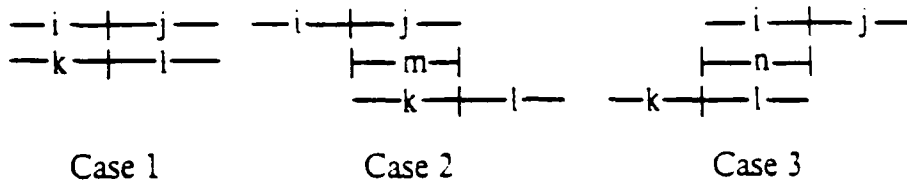


Figure 2: The Three Possible Orderings of  $i, j, k$ , and  $l$  in Axiom M3

Finally, we need some existence axioms. First, given any interval, there exists an interval that meets it, and an interval that it meets, i.e.,

(M4) ALL  $i$ . EXISTS  $j, k$   $j \text{ MEETS } i \text{ MEETS } k$   
 i.e.,  $j \mid i \mid k$

A consequence of this axiom is that no infinite time intervals are allowed in our theory. One further existence axiom is needed. It says that, given two intervals that MEET, there is an interval that consists of the two intervals concatenated, or merged, together. To define this precisely we need to introduce a "union" operator on intervals.

Defn-3:  $I+J$  is the ordered union of  $I$  and  $J$ , defined by

(M5) ALL  $I, J$ .  $I \text{ MEETS } J \Rightarrow$   
 EXISTS  $(I+J)$  such that EXISTS  $a, b$ .  
 $a \text{ MEETS } I \text{ MEETS } J \text{ MEETS } b \ \& \ a \text{ MEETS } (I+J) \text{ MEETS } b$   
 i.e.,  $\begin{array}{c} \text{---} a \text{---} | \text{---} I \text{---} | \text{---} J \text{---} | \text{---} b \text{---} \\ \text{---} I+J \text{---} \end{array}$

Using the defined relations above, this axiom can be restated as

ALL  $I, J$ .  $I \text{ MEETS } J \Rightarrow$  EXISTS  $(I+J)$  such that  
 $I \text{ STARTS } (I+J) \ \& \ J \text{ FINISHES } (I+J)$

We can prove that when  $I+J$  exists it is unique, and that  $+$  is associative.

An intersection operator on intervals also proves useful throughout in the proofs. Let  $I!J$  be the intersection of  $I$  and  $J$ , which is defined as follows:

$I!J$  is the interval such that

The intersection contains all intervals in both  $I$  and  $J$

(I1) All  $i$  .  $(i \text{ IN } I) \text{ AND } (i \text{ IN } J) \implies i \text{ IN } I!J$

The intersection, if it exists, is in both  $I$  and  $J$

(I2) EXISTS  $i$  .  $(i \text{ IN } I) \ \& \ (i \text{ IN } J) \implies (I!J \text{ IN } I) \ \& \ (I!J \text{ IN } J)$

We can prove that  $I!J$  is unique if it exists, and it exists whenever  $I$  and  $J$  overlap in the intuitive sense of the word.

i.e.,  $I \text{ (o o l s s i f f i d d i } =) J$ .

Since this is all derivable from axioms M1-M5, axioms I1 and I2 can be regarded as a definition of this notation.

## II. Capturing the Behavior of an Interval-Based Temporal Reasoner

The question now arises as to whether the above axiomatization of MEET and the definitions of the other interval relationships totally captures the behavior of the interval logic in [Allen, 1983]. This turns out to be the case, although it is tedious to show. We can prove that, for any two intervals  $I$  and  $J$ , then exactly one of the thirteen interval relationships possible holds between them. We can also show that the transitivity table in [Allen, 1983] is a result of the above axiomatization. This had to be shown entry by entry through the table, following the intuitive reasoning by possible cases which was used to construct the table originally. The proof, while long, is simple, as it only involves the repeated application of the ordering axiom M3. For example, given  $I$ ,  $J$ , and  $K$  such that  $I \text{ OVERLAPS } J \ \& \ J \text{ DURING } K$ , we know

EXISTS a.b.c.d.e . a MEETS I MEETS d MEETS e &  
 a MEETS b MEETS J MEETS c &  
 b MEETS c MEETS d

EXISTS f.g.h.j . f MEETS g MEETS J MEETS h MEETS j &  
 f MEETS K MEETS j

These facts can be presented pictorially as in Figure 3.

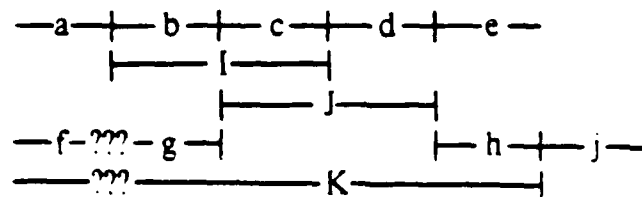


Figure 3:  $I \text{ OVERLAPS } J \ \& \ J \text{ DURING } K$

Using Axiom M3, and the facts  $a$  MEETS  $b$  &  $f$  MEETS  $g$ , we have three cases:

- 1)  $a$  MEETS  $g$ , and hence  $b = g$  (since  $a$  MEETS  $b$  MEETS  $J$  &  $a$  MEETS  $g$  MEETS  $J$ ): but then we have

$a$  MEETS  $I$  MEETS  $d+h$  MEETS  $j$  &  
 $a$  MEETS  $K$  MEETS  $j$ .

which by definition entails  $I$  STARTS  $K$ :

- 2) EXISTS  $m$  .  $a$  MEETS  $m$  MEETS  $g$ ; in which case we have

$a$  MEETS  $m$  MEETS  $g+c$  MEETS  $d-h$  MEETS  $j$  &  
 $a$  MEETS  $I$  MEETS  $d-h$  MEETS  $j$  &  
 $a$  MEETS  $m$  MEETS  $K$  MEETS  $j$

which by definition entails  $I$  OVERLAPS  $K$ :

- 3) EXISTS  $n$  .  $f$  MEETS  $n$  MEETS  $b$ ; in which case we have

$f$  MEETS  $n$  MEETS  $I$  MEETS  $h$  MEETS  $j$  &  
 $f$  MEETS  $K$  MEETS  $j$

which by definition entails  $I$  DURING  $K$ .

Thus, we have the fact that

$(I$  OVERLAPS  $J$  &  $J$  DURING  $K) \Rightarrow I$  (s o d)  $J$

which is one of the entries in the transitivity table in [Allen, 1983].

This set of five axioms is of a manageable size for comparing different theories and for theoretical proofs. This is not to say, of course, that the system should be re-implemented solely in terms of the MEET relation. There are important efficiency gains from using the larger set of primitives, as already described in [Allen, 1983].

### III. Nests: Beginnings and Endings

There are classes of events described in English that cannot be associated with a temporal duration. These are often called "instantaneous" events, or "accomplishments" (e.g., [Mourelatos, 1978]). Such events cannot be qualified by a duration. Thus, we can say "I closed the door," but if we say "I closed the door for three hours," it means we are repeatedly performing the action (contrast "I sat on the floor."). Some argue that this is because the closing the door describes the accomplishment of some state by the performance of some action. The time of this event is the time when the door changed state from being open to being closed.

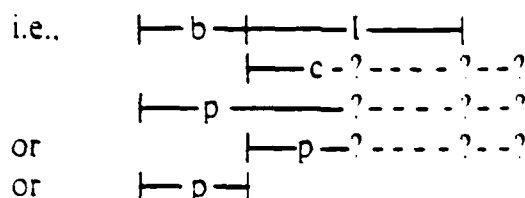
Other examples involve events that are considered instantaneous in the same way, yet the world is essentially identical after the event and before it. For example, a click, or the flash of a strobe, cannot be qualified by a duration. Yet the world after a click, or flash, could be essentially the same as before it. One common approach to

handling the times for such events is to model them as points (real points in the *continuous* model; integers in the *discrete* model). In this section and the following one, we shall develop two distinct notions of points from our interval logic. These will be compared in the final section.

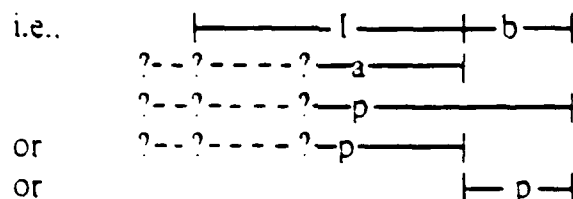
In this section we shall construct the equivalent of points within the interval logic defined in Section II by adopting a variant of filters, one of the standard mathematical constructions of points from intervals.

In particular, we define the beginning of an interval to be the set of all intervals that "touch the beginning" in any way, and the end similarly.

$$\text{BEGIN}(I) = \{p \mid \text{EXISTS } b, c . b \text{ MEETS } I \ \& \ b \text{ MEETS } c \ \& \\ p = b+c \text{ OR } p = c \text{ OR } p = b\}$$



$$\text{END}(I) = \{p \mid \text{EXISTS } a, b . I \text{ MEETS } b \ \& \ a \text{ MEETS } b \ \& \\ p = a+b \text{ OR } p = a \text{ OR } p = b\}$$



These sets are always non-null. These could also be defined by the following:

$$\text{BEGIN}(I) = \{ p \mid p \text{ (o s m f i d i e s i) } I \}$$

$$\text{END}(I) = \{ p \mid p \text{ (o i f m i f i d i e s i) } I \}$$

For convenience, we can define a *nest* as a beginning or an ending:

$$\text{NEST}(p) \iff \text{EXISTS } I . p = \text{BEGIN}(I) \text{ OR } p = \text{END}(I)$$

Given this definition of nests, we can now define relations over the set of nests which gives them the properties of points. To see this, we need to define an ordering relation on nests. We shall say a nest  $N$  is *before* a nest  $M$  iff there is at least one interval in  $N$  that is before some interval in  $M$ .

for any two NESTS,  $N$  and  $M$

$$N < M \iff \text{EXISTS } n, m . n \text{ is in } N \ \& \ m \text{ is in } M \ \& \ n < m$$



Now we can show some important properties about nests. First, nests are rather like filters. In particular, using the intersection and union operations defined above, we can show that nests are closed under intersections whenever they exist, and that if we take any interval  $n$  in a nest  $N$ , then  $n - m$  for any interval  $m$  (such that their union exists) is also in  $N$ . More formally, we have the lemmas:

Lemma 5: If  $P$  and  $Q$  are in a nest  $N$ , then  $P \cap Q$  exists and is in  $N$

Lemma 6A: If  $P$  is in a nest  $N$ , then  $P + Q$  is in  $N$  for any  $Q$  such that  $P$  MEETS  $Q$

Lemma 6B: If  $P$  is in a nest  $N$ , then  $Q + P$  is in  $N$ , for any  $Q$  such that  $Q$  MEETS  $P$

Another crucial property involves the union operation: if an interval  $n - m$  is in a nest  $N$ , then either  $n$  or  $m$  is also in  $N$ :

Lemma 7: If  $n + m$  is in  $N$  for any NEST, then either  $n$  is in  $N$  or  $m$  is in  $N$

The main result is that, given any two nests, and the ordering relationships between nests defined above, the nests must either be equal, or one is before the other. Furthermore, these possibilities are mutually exclusive.

Theorem 8: For any two nests  $N$  and  $M$ , either  $N < M$ ,  $M < N$  or  $N = M$

We can also show that the intuitive definitions of the interval relations in terms of nests are theorems. For example, we have

$$\begin{aligned} \text{BEGIN}(I) < \text{END}(I) \\ I \text{ MEETS } J < \Rightarrow \text{END}(I) = \text{BEGIN}(J) \\ I \text{ OVERLAPS } J < \Rightarrow \text{BEGIN}(I) < \text{BEGIN}(J) \ \& \\ & \text{BEGIN}(J) < \text{END}(I) \ \& \\ & \text{END}(I) < \text{END}(J) \end{aligned}$$

The second of these is especially important, as it shows that there is only one "place" where two meeting intervals actually meet. This is, perhaps surprisingly, a delicate matter. Very small changes in the definitions of *BEGIN* and *END* fail to achieve this. It is perilously easy to get a point structure, which distinguishes two "sides" of a single point, and other oddities, as discussed in [Van Benthem, 1982]. (We are grateful to Professor Dana Scott for bringing this and Hamblin's work to our attention, and emphasizing some of these subtleties.) We will discuss this at greater length in a forthcoming technical report.

#### IV. Discrete Time and Time Points

We can now show that discrete time models introduce a different kind of "point" than the points that are defined above. In particular, discrete time hypothesizes times that are not decomposable. Let us introduce a distinction between *true-intervals* and *moments* as follows:

ALL I . TRUE-INTERVAL(I)  $\Leftrightarrow$   
 EXISTS a,b,c,d . a MEETS I MEETS d  
 & a MEETS b MEETS c MEETS d

ALL I . MOMENT(I)  $\Leftrightarrow$  ~TRUE-INTERVAL(I)

Thus, a true-interval has at least two sub-intervals (which might in turn be moments or true-intervals)--one that STARTS it and one that FINISHES. Another way of stating the definition of a true-interval would be that it is the result of some union operation, i.e.,

ALL I . TRUE-INTERVAL(I)  $\Leftrightarrow$  EXISTS a,b . I = a-b

Before we continue, it is important to remember that all of the earlier theorems were proven before any distinction was made between moments and true-intervals, so they all hold for both classes: none of the proofs ever depended on the decomposability of an interval. These definitions allow us to prove that two moments cannot overlap in any sense of the term, yet they can MEET each other. More precisely,

ALL I,J . MOMENT(I) & MOMENT(J)  $\Rightarrow$  I ( $\ll$  m = m1  $\gg$ ) J

Let us now consider the relationship between nests and moments. The definition of nests did not exclude nests defined at the beginning or ending of moments. In fact, we can show that the beginning of a moment is less than the ending of that same moment! Thus, although a moment cannot be decomposed, we can distinguish its beginning from its ending.

We can also show that moments and nests cannot be considered to be isomorphic to each other. This is easily seen from the observation that moments can MEET each other, whereas nests cannot. Intuitively, a moment is a time *during which* some event (a flash, a bang) occurs, while a nest defines an abstract "position" in the sequence of times.

## V. Discussion

It is interesting to interpret these axioms in various possible models. The simplest one is discrete time: intervals are pairs of integers  $\langle n,m \rangle$  with  $n < m$ , and  $\langle n,m \rangle$  MEETS  $\langle m,k \rangle$ . Then a moment is a nondecomposable interval  $\langle n,n+1 \rangle$ , and nests pick out integers, the places "between" moments. In this model there is a clear distinction between moments and points. We can also define several models based on the real line. For example, time intervals can be mapped into open or closed real intervals: however, then times can never MEET. A simpler continuous model, based on the integer model above, defines time intervals as *pairs* of reals  $\langle a,b \rangle$ , with  $\langle a,b \rangle$  MEETS  $\langle b,c \rangle$ . Following through the axiomatic definitions with this as a basis makes nests define points on the real line, as expected, but now there are no moments at all, since even the smallest interval is decomposable. We might try to extend the model to allow intervals of the form  $\langle a,a \rangle$ , which would qualify as moments, but now consider  $\langle a,b \rangle$ ,  $\langle b,b \rangle$  and  $\langle b,c \rangle$ . By our definitions, the first MEETS the last, yet they have the second between them, so the first is BEFORE the last, violating the

ordering axiom. We have tried to fit real, substantial--though very small--time intervals into merely mathematical "places," and they don't fit.

However, another possible model is one which mixes these, using the same definitions of interval and MEET (from which all else follows) but allowing parts of the time line to be discrete and parts to be continuous. Intuitively, if we have only coarse time measuring tools available, then we can treat time as discrete, but the possibility always remains of turning up the temporal magnification arbitrarily far, if we have access to events which can make the finer distinctions, distinctions which can split "moments" into smaller and smaller parts.

Our axiomatic theory allows all of these models and others; it is uncommitted as to continuity or discreteness of the sequence of times, yet is powerful enough to support a great deal of the temporal reasoning of common sense.

### References

- Allen, J.F., "Maintaining knowledge about temporal intervals." TR 86, Computer Science Dept., U. Rochester, January 1981; *Communications of the ACM* 26, 11, 832-843, November 1983.
- Allen, J.F., "Towards a general model of action and time." *Artificial Intelligence* 23, 2, July 1984.
- Hamblin, C.L., "Instants and intervals," in J.T. Fraser, F.C. Haber, and G.H. Muller (Eds). *The Study of Time*. New York: Springer-Verlag, 1972.
- Mourelatos, A.P.D., "Events, processes, and states." *Linguistics and Philosophy* 2, 415-434, 1978.
- Van Benthem, J.F.A.K. *The logic of time*. Reidel, 1982.

# 7 A Model of Naive Temporal Reasoning

James F. Allen  
Henry A. Kautz

Department of Computer Science  
University of Rochester  
Rochester, New York

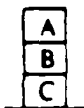
Temporal reasoning is an essential part of most tasks considered as intelligent behavior. In fact, it is so prevalent that it is often not recognized explicitly. In this paper, we shall consider naive temporal reasoning as it is required within two areas: natural language comprehension and problem solving.

For example, consider the following story:

Ernie entered the room and picked up a cup in each hand from the table. He drank from the one in his right hand, put the cups back on the table, and left the room.

This story contains many explicit temporal relationships. For instance, we are told that Ernie picked up the cups after he entered the room, and that the cups were picked up more or less simultaneously. After they were picked up, he drank from one, and later still he put the cups down more or less simultaneously. There are temporal relationships that are obvious from this story that are not explicitly mentioned as well. For example, we know that he held the cups for more or less the same span of time, and that while he drank from one cup, he was holding the other cup in his left hand.

In problem-solving situations, we see more evidence of temporal reasoning. Consider the simple blocks world in which there is one action, namely picking up a block and moving it to a new location. Assume we are given an initial situation with three blocks on a table and want to construct a tower:



We must reason that putting B on C must precede putting A on B. Looking deeper, however, the above temporal constraint is only valid in a domain in which only one put action can occur at a time. In a domain with two arms, for instance, we should

know that the action of putting B on C must complete no later than the action of putting A on B completes, but otherwise they may overlap or be performed simultaneously.

In more complicated domains, temporal reasoning becomes crucial. For instance, assume block A is on a rotating table and hence can only be reached periodically as it passes close to the arm. A plan to put A on B involves waiting for block A to appear and then picking it up while it is available.

All of these examples may seem obvious, but it is not obvious how to explain even such simple examples within existing models in artificial intelligence. This paper outlines a theory of time that accounts for many of the foregoing examples and that appears to be computationally effective. We hope to provide motivation and background for our research effort, leaving the actual details for other papers (Allen, 1981a, 1981b). In the first section, we shall discuss basic issues on the nature of temporal representations and argue for an interval-based approach rather than a point-based approach (i.e., in which time is viewed as analogous to the real line). We shall then provide a brief description of a temporal reasoner we have built and demonstrate it by some examples involving story comprehension. Finally, we shall describe how the model may be applied to problem solving.

### *1 A Theory of Time Based on Intervals*

Let us consider some (generally accepted) intuitions about time. The primary intuition is that times and events are intimately connected. Our perception of time is intimately connected (or identical to) our perception of events. Thus, in the discussion below, we will be discussing the nature of events as much as the nature of time.

1. Most of our temporal knowledge is introduced without explicit reference to a date. By date we mean not only calendar dates (e.g., March 25, 1950), but also times of day (e.g., 12 noon). Temporal information in language is conveyed mostly by tense, order of presentation, and temporal connectives (e.g., "before," "during," "while"). Temporal information in plans is relative to the other actions in the plan rather than to a specific date (e.g., action A must occur before action B).

The same emphasis on relative information over precise quantization occurs in considering durations of time. It is more common to learn that event A took longer than event B, than that A took 35 minutes versus B taking 15.

2. Given that we know an event E occurred over time T, we believe that by considering the event more closely we could in fact break down E (and hence the time T) into subevents (and hence subtimes). For example, the event of walking to school can be decomposed into a series of events consisting of one step, each of which could be decomposed into moving a leg forward, which could be decomposed into lifting your foot off the ground, pushing it forward, etc. Even events that appear to complete other events (e.g., arriving at school)

- can be decomposed (entering the building, opening the door, etc.) So it appears we can always "increase the magnification" and find more structure.
3. Notwithstanding the decomposability of times discussed in 2, it appears that we also often consider time as indivisible. In other words, in a given situation, we often view time as being pointlike. This obviously varies. A historian interested in ancient history might not care ever to break down years into finer divisions (thus a year is "pointlike"); however, a computer engineer may want to consider times as small as or smaller than nanoseconds. To the engineer, a second might be like a century to the historian.
  4. Time (or events) appears to be hierarchically organized. For instance, we can detect a hierarchy of times by considering the ambiguous nature of the word 'now.' 'Now' may refer to the time of my writing this sentence, the time of my writing this page, or larger periods such as the time of research projects. The question "What are you working on now?" at a conference refers to a much larger time than the instant the question is asked. If it didn't, one would have to answer "nothing" every time one had an idle moment. Thus, 'now' appears to be ambiguous, and can refer to one of a hierarchy of times based on containment.

These intuitions strongly support the claim that times should be viewed as intervals. What this means simply is that all times can be decomposed into subtimes. This is actually a fairly nontrivial claim, in that it disallows models in which times may be points. Thus we cannot start, for instance, with the real line as a model of the time line and build time intervals from time points as in Bruce (1972) and McDermott (1982). We want to claim that there are no time points in such a strong sense. The major arguments for this are, first, that allowing time points presents us with certain technical difficulties, and second, that we can do without time points.

An annoying characteristic of allowing time points in the strong sense is that it presents difficulties in defining the semantics of our temporal logic. For example, consider the time of running a race (call it RR) and the time after the race (AR). These two events are intimately related by definition; the latter interval beings where the former ends. If we allow time points, we must consider whether time intervals are open or closed. We cannot pick one option uniformly. If intervals are open, then there is a time between RR and AR in which the race is neither being run nor is it over. If intervals are closed, then there is a time in which the race is both being run and is over. It has been suggested that this problem can be avoided by a convention that all time intervals are open at a lesser end, and closed at the later end, but this seems completely arbitrary and indicates the model is unintuitive, for each interval would only have one endpoint. In an appropriate model, such a question should never have arisen. Another solution is to claim that it does not make sense for predicates to hold at time points, but this is essentially eliminating time points as useful entities.

Addressing the second point, we don't need to introduce points to explain or elaborate on interval-based descriptions. In the next section we shall introduce

seven relations (and their inverses) that completely characterize how two time intervals could be related.

While we want to allow pointlike temporal reasoning, we should not equate this with allowing infinitesimal points in our model. For one thing, we may later reason about the same times that were previously considered to be points as though they are intervals with rich internal structure. This could not be done if the original times were represented as points. We view point-based reasoning as a special case of interval reasoning: by viewing a time as a point we (temporarily) eliminate the possibility that it could overlap another time also viewed as a point. As a consequence the reasoning task would be simplified, for we would be ignoring any internal structure of the time "points." This allows a notion of abstraction somewhat similar to that in hierarchical problem-solving systems such as Sacerdoti's (1977).

Let us move from consideration of the underlying model of time to consideration of models of temporal reasoning. The intuitions discussed above appear to eliminate techniques that depend on constructing dates for each event. In the domains we are considering, such information is not available and cannot be constructed in a reasonable manner. Even allowing "fuzzy dates" (e.g., Vere 1981) will not solve the problem. For instance, assume we know that events A and B did not occur over the same time interval, either A occurred entirely before B or vice versa. There is no way we can assign fuzzy dates to A and B that capture this knowledge. Thus we are left with relative reasoning schemes.

One possibility (even without allowing points) would be to have partial ordering of the start and finish of intervals. Thus, intervals might start at the same time, or one might start before the other. Similarly, we could analyze the endings of intervals. This scheme is theoretically adequate but makes it difficult to define reasoning techniques that reflect our intuitions. For instance, the most common type of reasoning we will perform is considering whether some property P holds at a certain time t. Typically, such information will not be explicitly stored, but will need to be inferred. The key inference rule we desire is:

If proposition P holds over interval T, and interval t is during T, then P holds over interval t.

Modeling the during relationship using a partial ordering of endpoints makes this inference more complicated. Thus we prefer a representation that explicitly captures such containment information. A further benefit of this organization is that it reflects our intuitions about the hierarchical organization of temporal knowledge. This representation is outlined in the next section.

## ***2 A Representation of Temporal Knowledge***

### ***2.1 Reasoning About Intervals***

A pair of time intervals can be related in only a small number of ways, such as by the "during" and "meets" relationships above. Thirteen primitive rela-

X equals (=) Y	XXXXXX YYYYYY	
X before (<) Y	XXXXXX	
Y after (>) X		YYYYYY
X meets (m) Y	XXXXXX	
Y met by (mi) X		YYYYYY
X overlaps (o) Y	XXXXXX	
Y overlapped by (oi) X		YYYYYY
X starts (s) Y	XXX	
Y started by (si) X	YYYYYYYYYY	
X during (d) Y	XXX	
Y contains (c) X	YYYYYYYYYY	
X finishes (f) Y		XXX
Y finished by (fi) X	YYYYYYYYYY	

Figure 1. The Thirteen Primitive Relations

tions, graphically illustrated below in Figure 1, form the basis for all knowledge relating two intervals.

Often the precise relationship between intervals is not known. A complex relation is a disjunction of primitive relations. For example, if we know only that no part of X occurred outside of Y, then X could have been equal to Y, or could have fallen at the start, middle, or end of Y. Thus

X entirely within Y

abbreviates the complex relationship  $X (= s d f) Y$ .

Similarly, one can assert that X and Y are disjoint by saying that X is related to Y by any of the primitive relations which have that property:  $X (< m mi >) Y$ . Finally, knowing nothing about the relationship between X and Y is equivalent to holding the disjunction of all primitive relations between the two:

$X (= < > s si d di f fi m mi o oi) Y$

This formulation naturally suggests that our temporal knowledge be organized in a *constraint network*. Such a network is a directed graph, where each node represents an interval. The arc between any two nodes is labeled with the set of primitive relations which are consistent with our knowledge of the relationship of the intervals.

In addition to constraints imposed by world knowledge, the semantics of the temporal representation impose certain binary and ternary constraints. These constraints can be used to complete the graph, as well as reduce the size of label sets on the given arcs.

The binary constraints merely state that the label on an arc from, say X to Y is the inverse of that from Y to X. These constraints are readily derived from the table above, e.g.,



The inverse of a set of primitives is the set of inverses of each primitive:

If  $X (a s d) Y$  then  $Y (oi si c) X$ .

More interesting are the ternary constraints, which encode the *transitive* properties of the primitive relations. For example, if  $X$  is during  $Y$ , and  $Y$  is during  $Z$ , then certainly  $X$  is during  $Z$ . Some other combinations of primitive relations yield only disjunctive information. Suppose  $X$  starts  $Y$ , and  $Z$  finishes  $Y$ . Then  $X$  could be before, meet, or overlap  $Z$ :

If  $X (s) Y$  and  $Y (fi) Z$  then  $X (< m o) Z$

or, expressed more briefly,

$(s) * (fi) \rightarrow (< m o)$

The transitive constraints can thus be considered to define a multiplication operation over interval relationships. A multiplication table for eight of the primitive relations, shown in Figure 2, is easily constructed. In this table, the three relations  $s$ ,  $f$  and  $d$  are collapsed into the one relation  $d$ , and likewise  $si$ ,  $fi$  and  $di$  are collapsed into the one relation  $di$ . (For the table for all thirteen relations, see Allen, 1983a.) The product of two complex relations is simply the disjunction of all products of primitives from the first complex relation with primitives from the second.

We now have the basic deductive tools for working with our temporal logic. The network may be built and updated in various ways (see Vilain, 1982 for an alternative treatment to the one presented here). An algorithm based on *incremental constraint propagation* is useful for maintaining an updated network as new constraints and queries are dynamically applied.

Briefly, when a new constraint is asserted, one checks whether it in fact shrinks the old label set on the specified arc. If so, it (and its inverse) is inserted in the net, and all transitive relations passing through the arc (and its inverse) are computed. The process is then applied recursively to the resulting new constraints. Queries about the relationship of one interval to another are simply answered by inspection.

The story fragment which began this paper serves to demonstrate interval reasoning. Imagine that temporal assertions are being derived from a text as it is sequentially processed:

Emie entered the room and picked up a cup in each hand.

ENTER (m) HOLDR (A1)

ENTER (m) HOLDL (A2)

...ing. relations marked with A1, A2, etc. are assertions, and those with < etc. are derived relations. Entering meets both holding events.

Ar1B Br2C	<	>	d	di	o	oi	m	mi
<	<	no info	< o d m	<	<	< o d m	<	< o d m
>	no info	>	> oi d mi	>	> oi d mi	>	> oi d mi	>
d	<	>	d	no info	< o d m	> oi d mi	< m	> mi
di	< o di m	> oi di mi	o oi d di	di	o di	oi di	o di m	oi di mi
o	<	> oi di mi	o d	< o di m	< o m	o oi d di	<	oi di
oi	< o di m	>	oi d	> oi di mi	o oi d di =	> oi mi	o di	>
m	<	> oi di mi	o d m	< m	<	o d	<	d di =
mi	< o di m	>	oi d mi	> mi	oi d	>	d di =	>

Figure 2. The Transitivity Table

Constraint propagation fills in the relationship between the holding events, by multiplying **HOLDR (mi) ENTER** and **ENTER (m) HOLDL**:

**HOLDR (= s si) HOLDL** (D1)

He drank from the one in his right hand—.

**DRINK (d) HOLDR** (A3)

Drinking is now known to have occurred after entering (by multiplying (A3) and (A1) inverse), but the relationship to holding the second cup is not fully known. We derive:

**DRINK (> mi) ENTER** (D2)

**DRINK (> d f mi oi) HOLDL** (D3)

--put the cups back on the table and left the room

From the first proposition deduced from this statement:

HOLDR (m) LEAVE (A4)

We infer that leaving occurred after entering and drinking:

DRINK (<) LEAVE (D4)

ENTER (<) LEAVE (D5)

HOLDL (< c fi m o) LEAVE (D6)

Then, adding the final assertion

HOLDL (m) LEAVE (A5)

lets us derive that the cups were held for the same period of time, and that John drank while holding the second cup as well:

HOLDR (=) HOLDL (D7)

HOLDL (c) DRINK (D8)

The constraint propagation algorithm, we see, only derives significant new assertions about the relationships between intervals, and automatically halts when no more can be drawn. Such nice behavior is rather more difficult to obtain when one tries to feed a collection of axioms about time to a general-purpose theorem prover!

The basic propagation algorithm can be elaborated in several ways. It is obviously costly and unnecessary to maintain a complete graph for the constraint network. For example, the system may deal with a number of intervals that occurred yesterday, and a number that occurred today. Rather than explicitly link every node in the first group via a "before" arc to every node in second, we introduce *reference intervals*. All of yesterday's intervals can be asserted to be during a certain reference interval, say YESTERDAY, and all of today's intervals to be during TODAY. Finally, YESTERDAY is asserted to be before TODAY. Constraint propagation is not carried through reference intervals. Instead, when a query comes in concerning intervals not directly related, one answers it by climbing up the reference hierarchy. A more thorough discussion of reference intervals (including their use in maintaining a notion of the present) is found in Allen (1981a).

Alternatively, one may desire to ensure that the propagation algorithm is *complete*. Although basic constraint propagation seems to capture most of the inferences that are "obvious" to humans, in certain cases it does not constrain all arcs as tightly as possible. A more complex algorithm that builds a constraint hierarchy (Freuder, 1978) can be shown to fully capture the temporal semantics.

## 2.2 Reasoning About Durations

An important aspect of time intervals is that they have *durations*. So far, there is no way to assert that interval X lasted for 45 minutes, or that X lasted for twice as long as Y. Such knowledge can have consequences for the interval logic discussed above; for instance, if X has a smaller duration than Y, then X is constrained not to contain Y. This section describes a logic for reasoning about durations, which is separate from, but integrates nicely with, the basic interval logic.

The simplest approach might be to choose a basic unit for durations, say seconds, and assign a real number of those units to every interval. This is clearly inadequate, since knowledge of durations is often approximate:

The trip lasted *several* hours.

The reaction takes *two to five* minutes to complete.

The next stage might be to use fuzzy numbers for durations. Thus one might encode the second statement above as

REACTION = [120 300] SECONDS

Note that it is somewhat clumsy to use the same units for all durations. The problem is exacerbated when the size of unit is completely inappropriate; for example, encoding knowledge about geological eras in microseconds! Even worse, much knowledge refers to *no fixed scale*:

Driving across town *takes longer* than walking.

John ran around the track *three times* while Mary played tennis.

Perhaps surprisingly, all these statements can be represented in a uniform manner. Our solution is to construct a constraint network for duration knowledge (see Davis, 1981, for a very similar treatment of *spatial* knowledge). The nodes are time intervals (as before) and the arcs are labeled with fuzzy numbers, where a fuzzy number is an open or closed interval over the positive real numbers together with infinity. An arc indicates that its source node is a fuzzy multiple of its destination. The example statements could be represented as:

TRIP [2 10] HOURS

REACTION [2 5] MINUTES

DRIVING (1  $\infty$ ) WALKING

MARY-PLAY-TENNIS [3 3] JOHN-RUN-AROUND-TRACK

Incremental constraint propagation, using standard fuzzy multiplication, can be used to update the constraint network as new information arrives.

The next brief example shows the integration of scalar and relative duration

knowledge. Suppose Ernie and Bert travel across town, the first by car, the latter by bus. The bus ride is known to take 30 to 60 minutes.

BUS [30 60] MINUTES (A6)

Other real world knowledge states that the car trip should be faster:

CAR (0 1) BUS (A7)

That is, the car trip is between 0 and 1 times as long. Thus the car trip is between 0 and 60 minutes, exclusive:

CAR (0 60) MINUTES (D9)

Now suppose we learn that the car trip took at least 45 minutes:

CAR [45 ∞) MINUTES (A8)

Propagation then narrows the constraint on the length of the bus ride:

BUS (45 60] MINUTES (D10)

In more complicated situations, certain units (such as the "standard" time units—minutes, hours, weeks, etc.) can be used as *reference durations* to keep the network of manageable size.

### 2.3 A Unified Implementation

A time interval logic system and a duration logic system as described above have been implemented in LISP. Since certain interval logic assertions yield duration information, and vice versa, an interface program links the two systems. The following example demonstrates constraints flowing back and forth between the interval and duration systems as the temporal information gleaned from a story fragment is processed.

Moe and Larry began reading *Principia Mathematica*.

The two reading events could be simultaneous, or one could last longer:

MOE-READ (= s si) LARRY-READ (A9)

Moe read for over an hour.

Now a duration assertion is made:

MOE-READ (1 ∞) HOUR (A10)

Larry stopped reading and fell asleep after 10 minutes

This statement contains both interval and duration information. The first is the assertion that Larry's sleep follows on the heels of his reading.

LARRY-READ (m) LARRY-SLEEP (A11)

Next comes the duration assertion:

LARRY-READ [10 15] MINUTE (A12)

Given an assertion relating hours to minutes,

MINUTE [60 60] HOUR (A13)

The duration reasoner first deduces Moe's reading time in minutes (or Larry's in hours), and the relationship between the two:

MOE-READ (60  $\infty$ ) MINUTE (D11)

MOE-READ (6  $\infty$ ) LARRY-READ (D12)

When (D12) is added to duration network, the interface program notes that it potentially constrains the interval network as well, since it implies that Moe's reading event could not be during Larry's reading event. So the interval assertion is made:

MOE-READ (< > c si fi m mi o oi) LARRY-READ (D13)

Combined with (A9) above, this means that Larry's reading must start Moe's.

MOE-READ (si) LARRY-READ (D14)

Finally, interval constraint propagation adds the fact that Larry's sleeping must have begun while Moe read, but it's not known whether it continued on afterward.

LARRY-SLEEP (d f oi) MOE-READ (D15)

More elaborate examples can be devised where a duration constraint triggers an interval constraint, which eventually triggers another duration constraint, and so on for several iterations.

### *3 Problem Solving with the Temporal Reasoner*

Given the model of temporal reasoning, we can consider how it affects our approaches to planning and problem-solving systems. Current problem solvers have a

quite crude model of time: the world is represented as sets of facts, each set describing an instantaneous slice of time (a state). An action is a function from one state to the next. There is no notion of an action taking any time. A goal description is simply another state. The planner attempts to find a sequence of actions that will transform the initial state into the goal state.

State-based models do not easily allow for the possibility of simultaneous actions, or actions or events not caused by the agent. In addition, the goals described are confined to one time instant. Thus we couldn't express a goal such as "Put block A on B, and then later move A to block C." Goals of this form are not as uncommon as they might seem. For instance, the above goal statement might be the description of how to signal something to another agent.

There are other problems that have been addressed by some systems. For instance, McDermott (1978) allows constraints on the solution to a problem such as, "Don't violate goal  $x$  during the solution." Vere (1981) allows events not caused by the agent provided that there is a reasonable estimation of the date at which the event will occur. These are both attempts to introduce more general world models into problem solving. The foregoing temporal reasoner seems to suggest a model of planning that may be able to incorporate all the above and more into one uniform framework.

The model of the world we suggest is that the current state consists of all the planner's knowledge of the present and past, and predictions about the future expressed in an interval-based temporal logic. Planning an action does not update the state of the world but updates the predictions about the world. An action might affect beliefs about the future, the past, or in fact the present. Thus the states in this model are states of the planner's knowledge and independent of the temporal aspects of the world being modeled. We could generalize this further by explicitly introducing a belief predicate and indexing it by time, but this is unnecessary for the present discussion.

In this view, a plan is a collection of assertions viewed as an abstract simulation of some future world, including actions by the agent, other events, actions, and states. Most of these actions, events, and states must be causally related if it is to be a reasonable plan, though it may be simply known that certain events (and states) will occur without any causal explanation.

A goal is a partial description of the world desired. This description is not confined to the world at some specific time. A goal may include sequences of states (get A on B, then later get A on C), restrictions throughout (never let ON(B,C) be true), or any other set of facts expressible in the temporal logic.

Problem solving can be approached along fairly traditional lines. We could use means and analysis, decomposition of actions, etc., (e.g., Ernst & Newell, 1969; Fikes & Nilsson, 1971; Sacerdoti, 1977]). Action descriptions may be quite standard, except that each part of it will be temporally qualified. For the example below, we will use a STRIPS-like action formalism. An interesting side effect of this approach is that the temporal reasoner may do a lot of the problem solving for

us. For instance, it will allow for nonlinear plans along the lines of Sacerdoti (1977), and in fact do all the bookkeeping for deriving ordering constraints between actions.

Consider a very simple example. We want to stack three blocks, A, B, and C, starting from a world in which each is clear on the table. While this example is trivial, it allows us to demonstrate our approach in a fairly short space. Thus at the initial time interval I, we have the facts

holds(CLEAR(A),I)  
holds(CLEAR(B),I)  
holds(CLEAR(C),I).

Our goal description is to build a tower that stands during time interval F. Thus we want

holds(ON(A,B),F)  
holds(ON(B,C),F).

Note that as this stands, it can be considered to be a (very abstract) plan. It, however, has no causal connections between the initial and final states, so is not considered to be a useful solution. As we go along, we will list the temporal constraints that are added. Our first constraint is that I is before F.

I (<) F (A13)

Let us assume Sacerdoti's strategy for conjunctive subgoals: we shall solve each independently and then check for interactions. Each ON(x,y) goal can be achieved by an action PUTON(x,y) that has preconditions that x and y are clear, and an effect that x is on y. With the temporal augmentation, we have

occurs(PUTON(x,y),t)  
only if, holds (CLEAR(x),t1) where t finishes t1  
holds (CLEAR(y),t2) where t finishes t2  
and the effect is holds (ON(x,y),t3) where t meets t3.

Applying this action description to our first subgoal holds (ON(A,B),F) we introduce the following assertions into the plan for some time TAB:

occurs(PUTON(A,B),TAB) where I before or meets TAB;  
holds(CLEAR(A),TABp1) where TAB finishes TABp1  
holds(CLEAR(B),TABp2) where TAB finishes TABp2  
holds(ON(A,B),TABc) where TAB meets TABc  
and F is during TABc



To summarize, the temporal constraints added are:

$I (< m) TAB$  (A14)

$TAB (f) TABp1$  (A15)

$TAB (f) TABp2$  (A16)

$TAB (m) TABc$  (A17)

$F (s f d) TABc$  (A18)

These relationships can be shown pictorially if we let Time increase from left to right, and use dotted lines to indicate uncertainty as to the position of the ends of intervals. Then the constraints (A14) to (A18) can be shown as in Figure 3. The constraint propagation algorithm infers all the obvious relationships (e.g.,  $TABp1 (m) TABc$ ) implicit in this diagram.

Do the same for an action  $PUTON(B,C)$  during time  $TBC$ , and we get the constraints:

$I (CV m) TBC$  (A19)

$TBC (f) TBCp1$  (A20)

$TBC (f) TBCp2$  (A21)

$TBC (m) TBCc$  (A22)

$F (s f d) TBCc$  (A23)

Now general world knowledge must come into play. We know that  $CLEAR(x)$  and  $ON(y,x)$  are mutually exclusive, so if

$holds(CLEAR(x),t1)$  and  
 $holds(ON(y,x),t2)$

are true, then it must be the case that  $t1$  and  $t2$  are disjoint, i.e.,

$t1 (< > m mi) t2.$

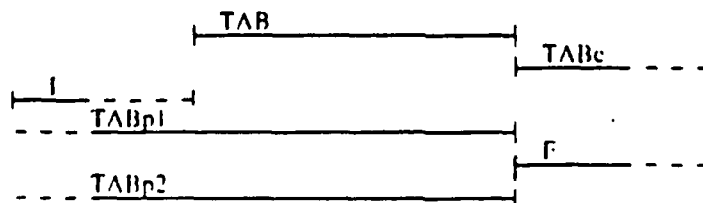


Figure 3. The Intervals Associated with  $PUTON(A,B)$

Examining the above plan, we find

holds(CLEAR(B),TABp2) and  
holds(ON(A,B),TABe),

and so could infer that

TABp2 (< > m mi) TABe, (D16)

but these two intervals are already further constrained to be TABp2 (m) TABe, which was derived from constraints in Axioms (A16) and (A17). As a consequence, this has no effect. A constraint that is not already known, however, arises from the facts

holds(ON(A,B),TABe) and  
holds(CLEAR(B),TBCp1),

giving us

TABe (< > m mi) TBCp1. (A24)

However, the constraint (A18) and the constraint TBCp1 (<) F inferred from Axioms (A20), (A22), and (A23) eliminate the possibility that TABe is less than or meets TBCp1. This situation is shown in Figure 4. Thus we are left with the conclusion that TABe is either after or met by TBCp1:

TABe (> mi) TBCp1. (D17)

This is shown in Figure 5. The relationship between TAB and TBC, the times of the two PUTON actions, is now constrained to be

TAB (e di si fi f > mi oi) TBC (D18)

Thus, PUTON(B,C) must be completed by the time PUTON(A,B) completes. The reason that this constraint is not stronger so far is that there is no implicit assumption that actions cannot be simultaneous in this model. If we add such a constraint

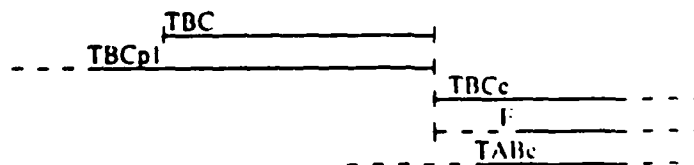


Figure 4. The Results of A18, A20, A22, and A23

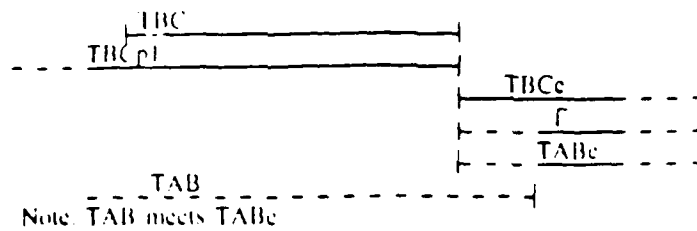


Figure 5. The Result of Adding A24 (Producing D17)

(bringing us in line with most current planning systems), then we have a new constraint that  $PUTON(A,B)$  and  $PUTON(B,C)$  must be disjoint:

$$TAB (< > m \text{ mi}) TBC \quad (A25)$$

Of these, only the ( $> \text{mi}$ ) relationships are possible; thus we derive

$$TAB (> \text{mi}) TBC. \quad (D19)$$

i.e.,  $PUTON(B,C)$  must occur before  $PUTON(A,B)$ . Thus, in effect, we have used the temporal inference machinery together with some general knowledge about the world to capture the action ordering as done by the resolve conflicts critic in *Sacerdoti*. This final configuration is shown in Figure 6.

As seen in the foregoing examples, actions may take time and may occur simultaneously. In a more complex example using hierarchical planning, as in *Sacerdoti*, we can model two actions that could occur simultaneously at one level of abstraction, but then when they are decomposed would have ordering constraints on their subparts. It is even possible that the subactions of two abstract actions could be interleaved. For more details on this, see Allen and Koomen (1983).

#### 4 Future Directions

We have presented a model of time based on hierarchically organized time intervals and specified an inference technique based on constraint propagation. This model appears to account for much of the temporal reasoning that is required for story comprehension and problem solving. There are cases where the technique, based on maintaining pairwise consistency between temporal relationships, is inadequate.

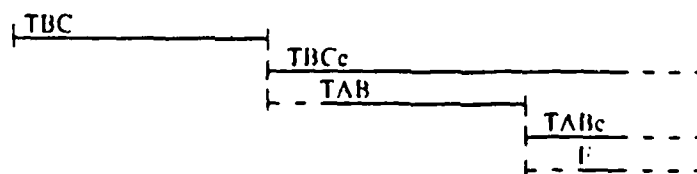


Figure 6. The Final Configuration, with D19

For example, in the problem-solving domain we can allow actions to occur simultaneously, or we can constrain actions of a certain type to occur sequentially. We cannot, however, constrain the domain so that two actions of a certain type may be simultaneous but more than two are not allowed.

For example, consider the state of holding a block. A two-armed robot would be able to hold two objects at one time. In other words, if T1, T2, and T3 are the times of three distinct holding states, then the three can never simultaneously overlap. They may pairwise overlap, however. Such a constraint cannot be expressed in the current system. Generalizing the technique to allow such constraints may be computationally prohibitive. If so, we shall have to introduce special-purpose mechanisms external to the temporal reasoning system to handle such cases.

The other major area of investigation concerns the organization and use of reference intervals in problem solving. Reference intervals are the mechanism for controlling the computation and must be used extensively in an efficient planner. Currently, our reference hierarchy mirrors the action hierarchy: each action has a reference interval which clusters together that action's preconditions, effects, and subactions. Problems arise from interactions between actions. We are considering various ways to adjust the reference hierarchy when such interactions occur.

### *Acknowledgments*

This work was supported in part by the National Science Foundation under Grant IST-8012418, the Defense Advanced Research Projects Agency under Grant N00014-82-K-0193, and the Office of Naval Research under Grant N00014-80-C-0197. We would like to thank Marc Vilain and Hans Koomen for many fruitful discussions concerning this work.

### *References*

- Allen, J. F. (1983a). *Maintaining knowledge about temporal intervals* (TR 86). Rochester, NY: U. Rochester, Computer Science Dept. *Communications of the ACM* 26, 11, pp 832-843.
- Allen, J. F. (1983b). *A general model of action and time* (TR 97). Rochester, NY: U. Rochester, Computer Science Dept.; to appear, *Artificial Intelligence*.
- Allen, J. F. & Koomen, J. A. (1983). Planning using a temporal world model. *Proc., 8th Int'l Joint Conference on Artificial Intelligence*, Karlsruhe, Germany, pp 741-747.
- Bruce, B. C. (1972). A model for temporal references and its application in a question answering program. *Artificial Intelligence*, 3, 1-25.
- Davis, E. (1981). *Organizing spatial knowledge* (TR 193). New Haven, CT: Yale University, Computer Science Dept.
- Ernst, G. W. & Newell, A. (1969). *GPS: A case study in generality and problem solving*. New York: Academic Press.
- Fikes, R. E. & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, 189-205.
- Freuder, E. C. (1978). Synthesizing constraint expressions. *CACM*, 21(11), 958-965.

- McDermott, D. (1978). Planning and acting. *Cognitive Science*, 2 (2), 71-109.
- McDermott, D. (1982). A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6 (2), 101-155.
- Sacerdoti, F. D. (1977). *A Structure for plans and behavior*. New York: Elsevier North-Holland.
- Vere, S. (1981). *Planning in time: Windows and durations for activities and goals*. Pasadena, CA: California Institute of Technology, NASA Jet Propulsion Laboratory.
- Villain, M. (1982). A system for reasoning about time. *Proc., AAAI*, Pittsburgh, PA, pp. 197-201.

## Toward a Theory of Plan Recognition

Henry A. Kautz  
Computer Science Department  
The University of Rochester  
Rochester, NY 14627

TR162  
July, 1985

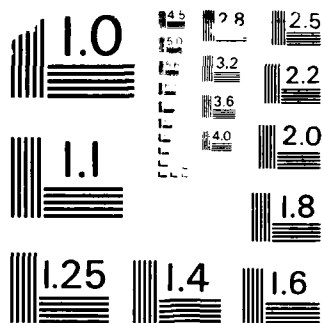
### Abstract

The problem of recognizing an agent's plans arises in many contexts in work in artificial intelligence. The plan recognition techniques suggested in the literature are rarely formally justified. We view plan recognition as a special kind of non-monotonic reasoning, and demonstrate how formal techniques developed for such reasoning -- namely, circumscription and minimal entailment -- can be used in plan recognition.

The first half of this paper reviews a broad range of work in artificial intelligence and philosophy which relates to plan recognition. A formal treatment of a simple case of plan recognition follows, and the paper concludes with proposals for future extensions of this work.

This work was supported in part by the National Science Foundation under grant number RCD-8450125 and in part by the Office of Naval Research under grant number N00014-82-K-0193.





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A



## TABLE OF CONTENTS

1. Introduction.....	753
1.1 Logical Theories of Plan Recognition.....	753
1.2 Kinds of Explanation.....	754
1.3 The Nature of Plan Recognition.....	754
1.4 Applications of Plan Recognition.....	756
1.5 Outline of Paper.....	756
2. Related Work.....	757
2.1 Plan Recognition and Discourse.....	758
2.1.1 Allen and Perrault.....	758
2.1.2 Litman.....	770
2.1.3 Cohen and Levesque.....	775
2.2 Automated Consultation.....	779
2.3 Expert Systems.....	782
2.4 Psychologically -Based System.....	784
2.5 Story Understanding.....	786
2.6 Statistical Inference.....	788
2.7 Non-Monotonic Inference.....	791
3. An Approach to Plan Recognition.....	794
3.1 The Planning Framework.....	794
3.2 A Propositional Example.....	795
3.3 Defining the Plan Recognition Relationship.....	796
3.3.1 The library is complete.....	796
3.3.2 All actions are purposeful.....	797
3.3.3 Simplicity of intention.....	797
3.4 Proof Theory.....	800
3.5 Plan Recognition Algorithms.....	801
3.6 Parameterized Plans and Actions.....	802
4. Future Directions.....	808
4.1 Example Problem.....	808
4.1.1 Multiple Observations.....	809
4.1.2 Plan Hierarchies.....	809
4.1.3 Tests and Conditionals.....	809
4.1.4 Loops.....	809
4.1.5 Concurrent Plans.....	809
4.1.6 Higher-Order Plans.....	809
4.2 Hierarchical Plans.....	810
4.3 Conclusions.....	811
Bibliography.....	812

LIST OF FIGURES

Figure 1: Relation of states to worlds..... 760  
Figure 2: Plan graph and two possible represented plans..... 761  
Figure 3: Search space generated by Plan Inference and  
Plan Construction rules..... 764  
Figure 4: Nested planning..... 767  
Figure 5: Incremental plan recognition/planning..... 771  
Figure 6: Pople's mechanized abductive logic algorithm..... 783  
Figure 7: Construction of  $\mu$  ..... 799  
Figure 8: Splitting and filtering..... 807

# Chapter 1

## Introduction

### 1.1 Logical Theories of Plan Recognition

Plan recognition is the process by which an observer infers the goals and intentions of an agent. It is a special kind of reasoning from effects to causes: physical effects, such as motions or utterances, are related to psychological causes. The observer can then predict the future actions of the agent, and either help or hinder the agent in the pursuit of the agent's goals. Theories of the mind based on work in artificial intelligence and cognitive psychology stress the importance of plans, structured representations of collections of actions, facts, and goals, in this process. The observer draws upon the information stored in a library of common plans, together with a plan inference algorithm, or set of plan inference rules, to perform the recognition.

A logical theory of plan recognition relates sentences describing observations to sentences describing the resulting mental state of the observer. Yet a logical theory of plan recognition is not a psychological theory, in the sense that specific logical rules correspond to specific psychological mechanisms. Nor must a logical theory predict specific kinds of errors people typically make. Instead, a logical theory describes the ability of a perfectly competent agent. Perfect competence is usually identified, in the philosophical tradition, with perfect rationality. The concept of perfect rationality is hard to pin down, unless one simply defines it to be, for example, maximizing expected utility. It is unclear, for instance, if a perfectly rational agent must be logically omniscient. None the less, some aspects of perfect rationality seem uncontroversial: an agent should not hold contradictory beliefs, or at least should be prepared to revise his beliefs if he discovers a contradiction; an agent acts in order to achieve his goals in an efficient manner; an agent tries to keep his beliefs about the world in accord with the actual state of the world; an agent believes that other agents are rational beings like himself; and so on.

Is a logical theory of plan recognition possible or even desirable? It is possible to argue that plan recognition is properly extra-logical, a way of reversing parts of a theory of action; the process is fundamentally heuristic; looking for a logic different from ordinary logic is misguided. But these objections can be answered. First, plan recognition must be logically formalized in order to develop a complete theory of planned behavior. Not only do agents commonly assume that other agents will anticipate their desires, certain acts -- in particular, speech acts -- are planned in order for another agent to recognize the intentions behind them. Second, although particular plan recognition algorithms may not be provably correct or complete, it is important to have a precise statement of the problem which they are approximately solving. Heuristic algorithms should be viewed as special-purpose deductive rules, "hard-wired" theorems, rather than as simply rules of thumb. Finally, a logical theory of plan recognition does not purport to replace deductive logic, and may even be expressed in traditional first-order logic. Probability theory is an example of a very successful framework for reasoning from effects to causes which, as [Kyberg 74] shows, can be given a precise definition in first-order logic.

Philosophers have identified three main modes of reasoning: deductive, abductive, and inductive. They are illustrated by the following syllogism:

- i. All men are mortal.
- ii. Socrates is a man.
- iii. Socrates is mortal.

Deduction allows one to conclude (iii) from (i) and (ii). Abduction allows one to conclude, in the proper circumstances, (ii) from (i) and (iii). (Those circumstances include not knowing that Socrates is some creature other than a man.) Induction allows one to conclude (i) from many cases like (ii) and (iii). Plan recognition is thus a special case of abduction. Philosophers such as Peirce [Goudge 69] have argued that logical theories of abductive and inductive inference are both possible, and necessary to help describe the greater part of scientific inference.

I propose to begin to develop a logical basis for plan recognition, which will be used to justify and explain particular plan recognition algorithms in terms of conceptually simple model-theoretic operations.

## 1.2 Kinds of Explanation

The driver of a '63 Pontiac sedan turns on his left turn signal at the corner of Oak and Hillcrest. What explains this action? Many explanations can be offered, including: the driver wants to signal a left turn; the driver wants to make a left turn; or the driver wants to buy a set of spark plugs at the auto parts store at the end of Oak Street. These explanations are all compatible, and exhibit increasing degrees of speculation on the part of the observer. They also show different ways that a particular action can be connected with a plan.

The driver signals a left turn by turning on the left turn signal. Using the terminology of [Goldman 70], turning on the signal generates the act of signaling. One act generates another if simply performing the first act in appropriate circumstances counts as performing the second. In this case, the appropriate circumstances include being on a highway, and not, for example, being parked in a garage. The driver signals the left turn as part of the plan for making a legal left turn. Finally, making the left turn, and then driving down the street, places the driver at the auto parts store, and thus enables him to buy the spark plugs. Sometimes it is useful to distinguish these three kinds of explanation, which will be called generational, step, and enabling. Some plan recognition algorithms provide different mechanisms for each kind of inference. The distinctions between the types of explanation are not always clear (nor may they even be important); different formulations of the same problem can often assign basically the same explanation to one category or another.

## 1.3 The Nature of Plan Recognition

Suppose we observe a woman enter a train station, as in [Allen 80]. The known plans involving train stations are those to meet or to board a train, and so we conclude that the woman is in fact performing one of those plans. But then we may discover that she is actually looking for a lost dog. Or suppose we learn that Mr. Smith withdrew a large sum of money from his savings account and immediately went to the race track. The obvious temptation is to combine these pieces of information, and conclude that the dissolute Mr. Smith is going to blow his family's fortune on the horses. But the actual explanation may be much more innocent. Mr. Smith may be at the race track in order to hand out religious tracts, and the money may be for a personal computer he is buying later that afternoon.

Plan recognition is a paradigmatic case of non-monotonic reasoning. On the basis of partial evidence, the observer jumps to conclusions not sanctioned by deductive inference. In the first example above, we assume that the known plans involving the train station are the only plans involving it. In the case of Mr. Smith, we assume that all evidence should be accounted for by a single explanation. Additional information -- such as the fact that Mr. Smith never gambles -- would have led us to draw different conclusions.

These examples illustrate one of the difficulties in formalizing plan recognition in monotonic logic. Let  $L$  be a set of axioms for planning, plan recognition, and general real-world knowledge;  $B$  be a set of observations of the actions performed by an agent; and  $\vdash$  be the first-order provability relation. From  $B$  together with  $L$  we wish to infer an explanation for  $B$ , in terms of the intentions of the agent. That is, we wish to find a statement  $P$  such that

$$B \cup L \vdash P$$

where  $P$  describes the agent's overall intentions. No matter what additional observations or pieces of world knowledge are added,  $P$  still can be derived. In particular,

$$B \cup L \cup \{\neg P\} \vdash P$$

For example, if from observing Mr. Smith going to the racetrack we conclude that Mr. Smith wants to gamble, then from observing Mr. Smith going to the racetrack and not gambling we can also conclude that Mr. Smith wants to gamble. While the basic problem here seems obvious, there are proposals for formalizing plan recognition which have precisely this property.

Therefore  $\vdash$  must be replaced by some non-monotonic derivability relation. Let us call this relation  $(L)\vdash$ , to indicate that it somehow incorporates the information in  $L$ . This relation should allow the following statements to simultaneously hold:

$$\begin{aligned} B (L)\vdash P \\ B \cup \{\neg P\} \neg(L)\vdash P \end{aligned}$$

There are a number of ways to go about defining  $(L)\vdash$ . These include:

1. Probability theory (section 2.5).
2. Default logic (section 2.6).
3. Let  $L$  be a set of axioms for planning. Define a function  $\phi$  which maps  $L$  and  $B$  into a stronger set of formulas such that

$$\phi(L,B) \vdash P \text{ iff } B (L)\vdash P$$

(Chapter 3).

4. Define  $(L)\vdash$  as a predicate in a (monotonic) meta-language, or a language with quotation of formulas, in order to discuss non-monotonic inferences from sets of sentences.

The first two approaches are discussed in the section on related work below. The main piece of new work in this proposal, chapter 3, develops the third approach. There I show how the axioms of a theory of planning can be systematically augmented to yield just the implications desired for performing plan recognition. These manipulations are given a precise (model-theoretic) semantics, and, I argue, can be justified by general principles of rationality. The last alternative above is the most general. In my proposals for future work, I am leaning toward using a meta-linguistic approach, in order to have an explicit recognition relationship available, for use in those cases of "intended recognition" discussed above.

## 1.4 Applications of Plan Recognition

Plan recognition is a central component in many programs of research in artificial intelligence. These include work on story understanding [Bruce 81, Wilensky 83], "helpful" natural language systems [Allen 80, Cohen, Perrault, & Allen 81, Carberry 83, Litman 84], program-writing assistants [Rich 81], high-level computer system interfaces [Huff & Lesser 82], and models of strategy and conflict [Carbonell 79].

The general problems of plan recognition are of secondary interest to these researchers, who are mainly interested in developing sophisticated theories of the various application domains, which include some aspects of plan recognition. My work should prove complementary to theirs: inspired by the issues they have raised, and providing a general foundation for the plan recognition techniques they have developed.

## 1.5 Outline of Paper

Chapter 2 discusses related work on plan recognition, expert systems, and probability theory. Chapter 3 shows how plan recognition can be understood as a process of defining a set of "minimal" models which contain the observations. This process is given a proof-theoretic definition, and employs and extends McCarthy's circumscription operator in a novel way. I also discuss a simple plan recognition algorithm which is, in restricted cases, correct according to the semantic theory. In chapter 4 future directions of research are discussed.

## Chapter 2 Related Work

This work builds on efforts in several different areas of research. In artificial intelligence, plan recognition has received most attention from people interested in plan-based theories of discourse. Section 2.1 reviews the plan-recognition aspects of the work of Allen, Litman, and Cohen and Levesque. Plan recognition is also a major part of the work on automated consultants, discussed in section 2.2. Many of the problems encountered in plan recognition are instances of more general problems of reasoning from effects to causes. Section 2.3 discusses work on medical expert systems which addresses some of these issues. Section 2.4 discusses the early and important psychologically-oriented work on plan recognition by Schmidt, Sridharan, and Goodson. Finally section 2.5 briefly overviews work on story understanding, and particularly that by Wilensky.

Outside of artificial intelligence, probability and decision theory provides the most popular framework for dealing with the kinds of concerns present in plan recognition. In section 2.6 I explain why probability theory alone does not immediately solve all the problems of plan recognition and suggest that my work be viewed as complementary to, rather than in competition with, probabilistic approaches.

One of major ways in which my work differs from previous efforts is my attempt to formally describe the non-monotonic aspects of plan recognition. Section 2.7 describes work by McCarthy and Lipschitz on circumscription. Their model theory for circumscription will be generalized and incorporated in my model theory for plan recognition.

## 2.1 Plan Recognition and Discourse

### 2.1.1 Allen and Perrault

Papers by Phil Cohen and Ray Perrault [Cohen 78] first formalized Austin's and Searle's speech act theory in terms of planning. Utterances were viewed as actions which transformed the beliefs of the speaker and hearer. James Allen [Allen & Perrault 80] extended the analysis to include plan recognition. Plan recognition is necessary to account for the fact that a speaker need not fully and literally execute a speech act in order to achieve its effect. Instead, the speaker need only perform an act which suggests to the hearer that the speaker's overall intention is to achieve the desired effect. Phenomena accounted for by plan recognition include indirect speech acts, understanding of sentence fragments, and certain kinds of context-dependent implicatures.

Allen analyzed plan recognition in terms of a set of plan recognition rules together with a heuristic control strategy. The rules isolate those inferences which are plausible, or are suggested by the recognizer's set of beliefs. The control strategy determines which of these inferences are likely, or should actually be accepted. This very general framework is also applied to planning, where rules are used to suggest possible future actions, and the control strategy determines which actions to ultimately perform. Allen's most influential achievement was in determining a powerful and universal set of plan inference rules. The corresponding control strategy was less well developed, but did grapple with some important and difficult issues.

Underlying Allen's system is a state-space formalization of planning. In order to be precise in our terminology, it is convenient to develop the system in several steps. Imagine that the universe consists of a set of instantaneous possible worlds; a particular sequence of these worlds is distinguished as **actual**. Actions are functions from worlds where their preconditions hold to worlds where their effects hold. A state is a set of propositions which describes a set of possible worlds: namely, those in which all the propositions hold. An action instance links two states if the set of worlds obtained by applying the action to each world described by the first state is included in the set of worlds described by the second state. An action description is a syntactic rule for determining what states can be linked by what action instances. (In much literature on planning it is common -- though a bit misleading -- to blur the distinction between actions and action descriptions.) Figure 1 illustrates this terminology.

An action instance may have a **body**, which is either: a sequence of more primitive action instances which specify how that action is performed; or a proposition, such that any plan which achieves the proposition is also an instance of the action. A **planning problem** is a set of action descriptions, an initial state, and a set of goal states. A **solution plan** is a chain of action instances and intermediate states which link the initial and a goal state. A (partial) **plan** is simply the initial state and a goal state together with some action instances and intermediate states.

Rather than directly synthesizing solutions, however, it is generally more efficient to adopt a notation which allows us to represent partially ordered plans. A plan is represented by a directed graph. Each node is either a proposition or an action instance. Arcs link preconditions to action instances, and action instances to effects. N-ary links may connect action instances to their bodies. Figure 2 illustrates a simple plan graph, and two of the



linear plans which it could represent. Note that the plan graph does not represent an entire state as a node. A partial plan corresponds, in general, to a disconnected plan graph. The part of the graph connected to any goal proposition is called the expectation, and that connected to any initial proposition, the alternative. Let us say that a graph is grounded if it is connected, and all of its leaves are initial state propositions. Any solution plan is represented by a grounded plan graph, but the reverse need not hold. Once a grounded plan graph is found, it is necessary to check that it represents at least one linear plan. A planning problem is represented by a set of plan graphs, where each plan graph contains the same initial state but a different goal state.

So much is fairly standard. As explained in [Nilsson 80], planning can be viewed as a (possibly bidirectional) search through the space of intermediate states, in order to construct a graph connecting the initial and goal states. Viewed at the meta-level, however, planning is a search through the space of partial plans. The entire original planning problem is considered one node (or one node for each possible goal state), and the solution plan another node (ibid). Instead of being linked by actions, nodes at this level are linked by rules which suggest ways to further specify or alter a partial plan. These plan construction rules include the backward-chaining heuristics normally hardwired into planning systems. Allen saw that plan recognition could be formally defined so as to be functionally equivalent to planning at this level.

A plan recognition problem consists of a sequence of observed initial states linked by observed action instances, and a set of expected goal states. A solution is sequence of action instances which complete the chain from the last initial state to any goal state. In terms of plan graphs, either a planning or plan recognition problem is simply a set of partial plan graphs. A solution is a linearizable grounded plan graph, which further specifies one of the problem graphs.

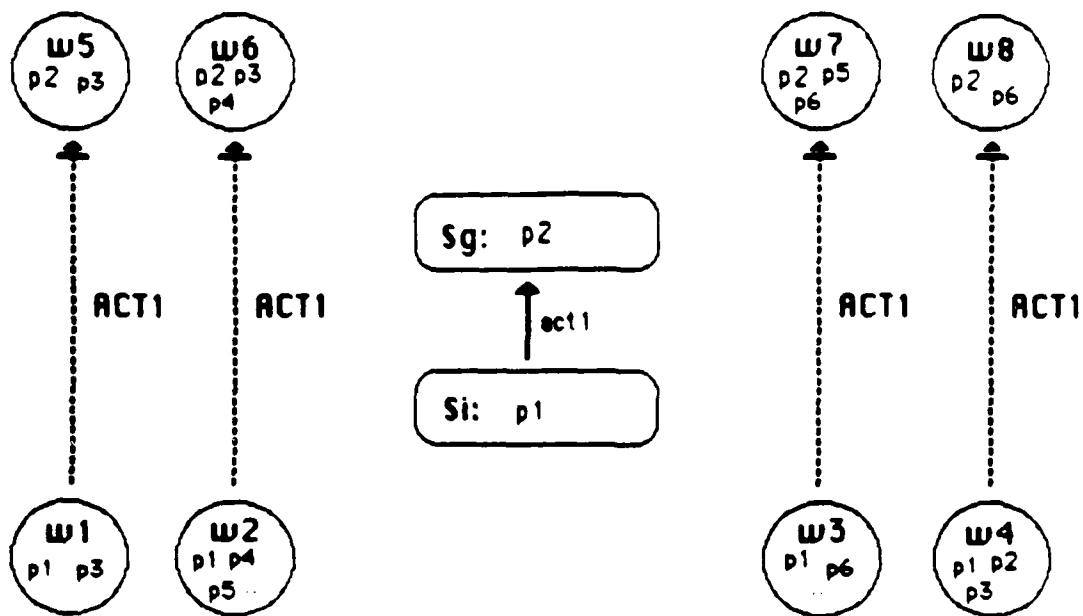
While the rules suggest ways to expand a node in the meta-level search space, a control strategy determines which nodes to expand, and which solutions are good ones. Any planning problem may have an infinite number of solutions, but most should be rejected out of hand, because they contain unnecessary or redundant actions. Likewise, a plan recognition problem may have many implausible solutions. In some cases the system may lack either sufficient information about the problem or computational resources to come up with any solution. In such cases it is incumbent upon the control strategy to choose the best partial graph. Planning and plan recognition may favor different control strategies, as well as different rules. For example, a plan recognition strategy may favor partial solutions which make most use of observed actions, while a planning strategy may favor graphs which ground the greatest percentage of the goal propositions.

Rather than simply reasoning about the "real world", Allen's system actually reasons about an agent's wants and beliefs. KNOWS, BELIEVES, and WANTS are treated as two-place modal operators, each taking an agent as a formula as arguments. Formulas of the form

**X WANT P**  
**X BELIEVE P**

are sometimes abbreviated as **XW P** and **XB P** respectively. When a sentence **P** appears in a formula of the above form, we say that "**P** is embedded by **XW**" or "**P** is embedded by **XB**".

Single propositions, states, or entire plans or plan graphs may fall within the scope of a belief or want operator. To want a proposition is to want to be in a state where that proposition holds; to want a plan is to want one of the sequences of worlds described by that plan to be actual. When an agent **X** plans, each plan graph is actually embedded by



## Relation of states to worlds

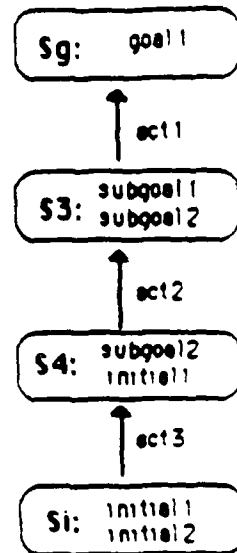
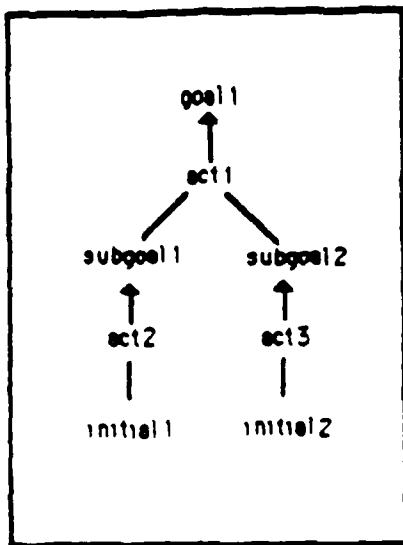
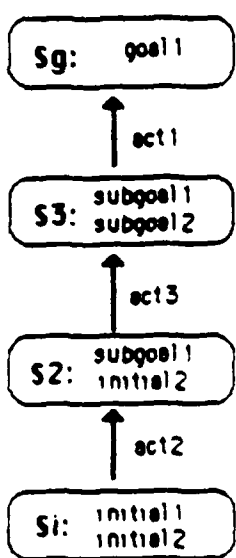
State  $S_i$  consists of proposition  $p_1$ , and represents worlds  $w_1, w_2, w_3, w_4$ , etc

State  $S_g$  consists of proposition  $p_2$ , and represents worlds  $w_5, w_6, w_7, w_8$ , etc.

$act1$  is an instance of action  $ACT1$ , where

$$\begin{aligned} w_5 &= ACT1(w_1) \\ w_6 &= ACT1(w_2) \\ w_7 &= ACT1(w_3) \quad \text{etc} \end{aligned}$$

Figure 1



Plan graph (center) and two possible represented plans

Si: initial state Sg: goal state S2, S3, S4: intermediate states  
goal1, subgoal2, initial2, etc propositions

Figure 2

**XW**. Plan recognition is performed in the context of what the system believes that an agent **A** wants (**SBAW**). The operator **KNOWREF** holds of terms for which the agent knows the referent. The expression **X KNOWIF P** abbreviates  $(X \text{ KNOW } P) \vee (X \text{ KNOW } \neg P)$ .

Table 1, taken from [Allen 83], lists the plan construction and plan recognition rules used by the system. The operators **=>c** and **=>i** are not logical connectives, but indicate "likely" or "possible" inferences. As discussed above, the plan construction rules (those which use the **=>c** operator) refer to propositions that an agent **X** wants, while the plan recognition rules (those which use the **=>i** operator) refer to those that the system believes that an agent **A** wants. But note that both kinds of rules legitimately can apply to either kind of problem. Any proposition embedded by **SBAW** is, of course, also embedded by the inner **AW**. Furthermore, in the epistemic logic used by Allen there is no distinction between what an agent wants and what an agent believes that he wants. Thus, where we set **X=A=S**, any proposition embedded by **XW** is also embedded by **SBAW**. Figure 3 illustrates how one may begin to search the space of plan graphs in several different ways, starting at the graph in the upper left corner. (Each step in the diagram actually corresponds to the application of a pair of rules.)

**Chain backwards:**

action-precondition	$XW \text{ Act} \Rightarrow_c XW P$	P a precondition of Act
action-body	$XW \text{ Act} \Rightarrow_c XW B$	B a part of body of Act
effect-action	$XW E \Rightarrow_c XW \text{ Act}$	E an effect of Act
know-rule	$XW P \Rightarrow_c XW (X \text{ KNOWIF } P)$	
nested planning	$XW (Y \text{ WANT } P) \Rightarrow_c$ $XW (Y \text{ WANT } Q)$	if X believes $Y \text{ WANT } Q \Rightarrow_c$ $Y \text{ WANT } P$

**Plan Construction Rules**

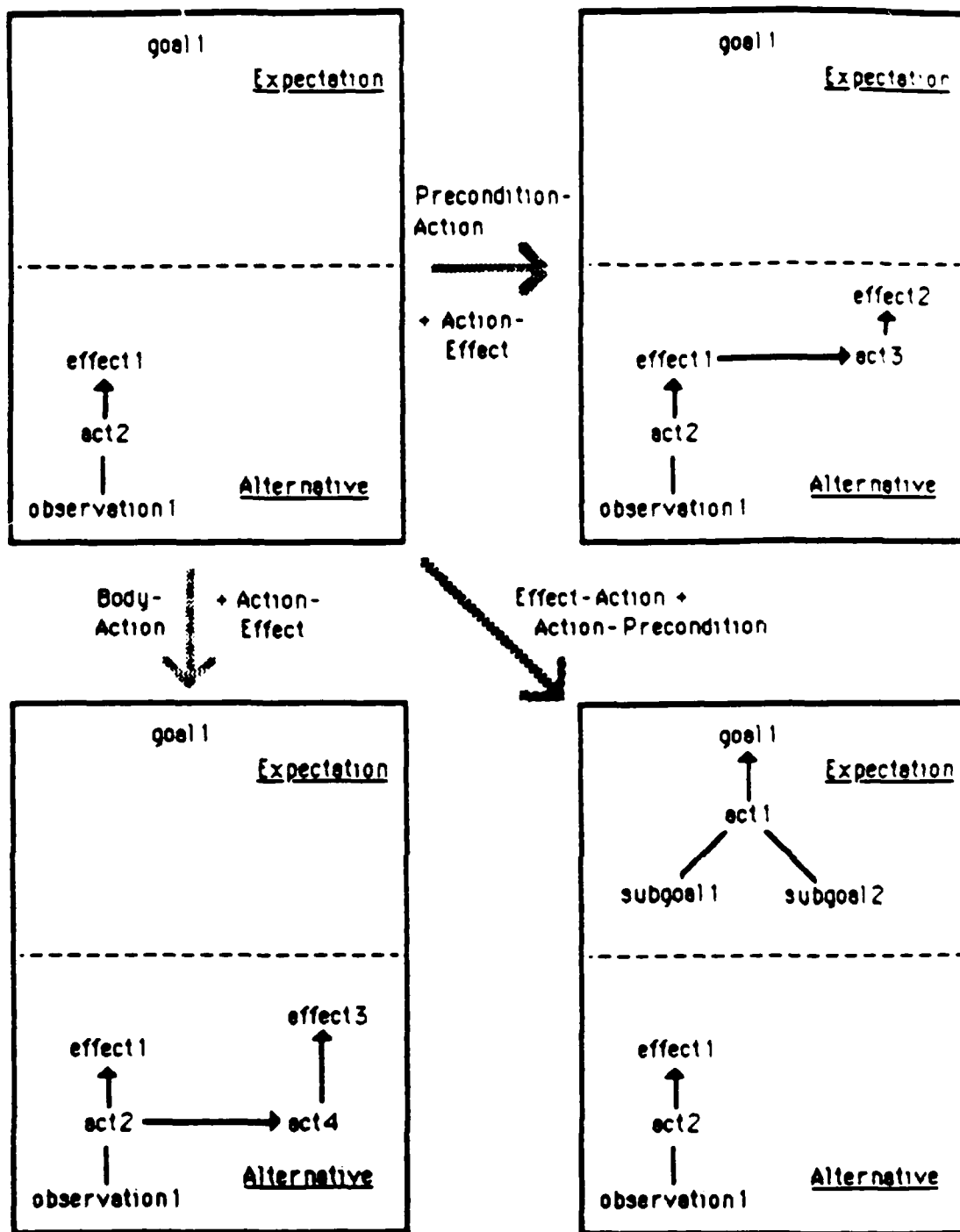
---

**Chain forwards:**

precondition-action	$SBAW P \Rightarrow_i SBAW \text{ Act}$	P a precondition of Act
body-action	$SBAW B \Rightarrow_i SBAW \text{ Act}$	B a part of body of Act
action-effect	$SBAW \text{ Act} \Rightarrow_i SBAW E$	E an effect of Act
want-action	$SBAW (n \text{ WANT } \text{Act}) \Rightarrow_i$ $SBAW \text{ Act}$	n is the agent of Act
know-positive	$SBAW (A \text{ KNOWIF } P) \Rightarrow_i SBAW P$	
know-negative	$SBAW (A \text{ KNOWIF } P) \Rightarrow_i SBAW \neg P$	
know-value	$SBAW (A \text{ KNOWIF } P(a)) \Rightarrow_i$ $SBAW (A \text{ KNOWREF the } x : P(x))$	
know-term	$SBAW (A \text{ KNOWREF } D) \Rightarrow_i$ $SBAW(P(D))$	
recognition of nested planning	$SBAW (S \text{ WANT } P) \Rightarrow_i$ $SBAW (S \text{ WANT } Q)$	if $SBAB (S \text{ WANT } P$ $\Rightarrow_c S \text{ WANT } Q)$
decide inference	$SBAW (SBAW P) \Rightarrow_i$ $SBAW (S \text{ WANT } P)$	

**Plan Inference Rules**

**Table 1: Plan construction and inference rules from [Allen 83]**



Search space generated by Plan Inference and Plan Construction → Rules

Figure 3

The most general and important rules are those labeled **chain backward** and **chain forward**, the former used in construction and the latter in recognition. These implement a GPS-like problem reduction strategy. In fact, the other rules can, for the most part, be reduced to important special cases of these general rules. These are discussed in order, starting with the plan inference rules.

The **want-action** rule states that if A wants some other agent n to want to do some act, then A wants that act to be done. In fact, an agent wanting to do an act is a precondition for all intentional actions. Thus this rule is a special case of **precondition-action**.

**Know-term** is an important rule which states that if you want to know what a term refers to, you may have another goal which takes that term as an argument. This rule arises because in order to actually execute an action, one must generally know the referents of the parameters of the action. Thus there is actually a **KNOWREF** precondition for every term which appears in every action. Again, this is a case of **precondition-action**.

On first examination the rules **know-positive** and **know-negative** do not seem to be cases of this kind of forward chaining. An example of the use of these rules is the situation in which I ask you, "Are the police here?" If I'm a law-abiding citizen, you can infer that my goal is in fact for the police to be here; if I'm a criminal, then probably my goal is for the police not to be here. What then is the link between **KNOWIF P** and either **P** or **¬P**, in the vocabulary of actions and plans? Suppose my goal is **P**. Before I perform any other action to attempt to achieve **P**, it is simply a good idea to check whether **P** already holds. If it does, I do nothing; otherwise, I must find some less effortless plan to achieve **P**. Therefore let us add the following action schema to the library of action descriptions:

**CHECK-POSITIVE**(agent, P)  
precondition: agent **KNOWIF P**  
effect: **P**  
body: if **P** then null; else **achieve(P)**

**CHECK-POSITIVE** thus provides the link between wanting to know whether **P** holds and wanting **P**. The **achieve(P)** clause in the body of the action need not be expanded at the time the agent adopts a plan containing this action. A very cautious agent may, of course, do so, in order to foresee what to do in the worst case. There is, of course, a similar action:

**CHECK-NEGATIVE**(agent, P)  
precondition: agent **KNOWIF P**  
effect: **¬P**  
body: if **P** then **achieve(¬P)**; else null

Given these two actions, the **know-positive** and **know-negative** rules each reduce to an application of the **precondition-action** followed by the **action-effect** rules.

**Know-value** is similar to the previous rules. This rule states that if you want to know if some object "a" has property **P**, then you may want to know the referent of *the* thing with property **P**. The missing action here is something like the following:

CHECK-REFERENT(agent, the x: P(x), a)  
precondition: agent KNOWIF P(a)  
effect: agent KNOWREF the x: P(x)  
body: if P(a) then null; else achieve(agent KNOWREF the x: P(x))

This is a bit over-simplified; there must also be a condition in the precondition or body that the agent believes that a is the only thing (or the only thing in focus) that is a P. Like the two previous actions, this one must be incorporated into plans with care; in particular, if P(a) does not turn out to be true, then the expansion of the body had best not consist of the same CHECK action again.

The decide-inference involves the way in which one agent can get another agent to adopt a goal. If you are helpful and friendly to me, then one way I can get you to adopt the goal P is to simply make you aware that I want P. That is, from the fact that you think I want you to believe that I want P, infer that I want you to have the goal P. One way to make this link explicit is by introducing a "cause to want" action. That is:

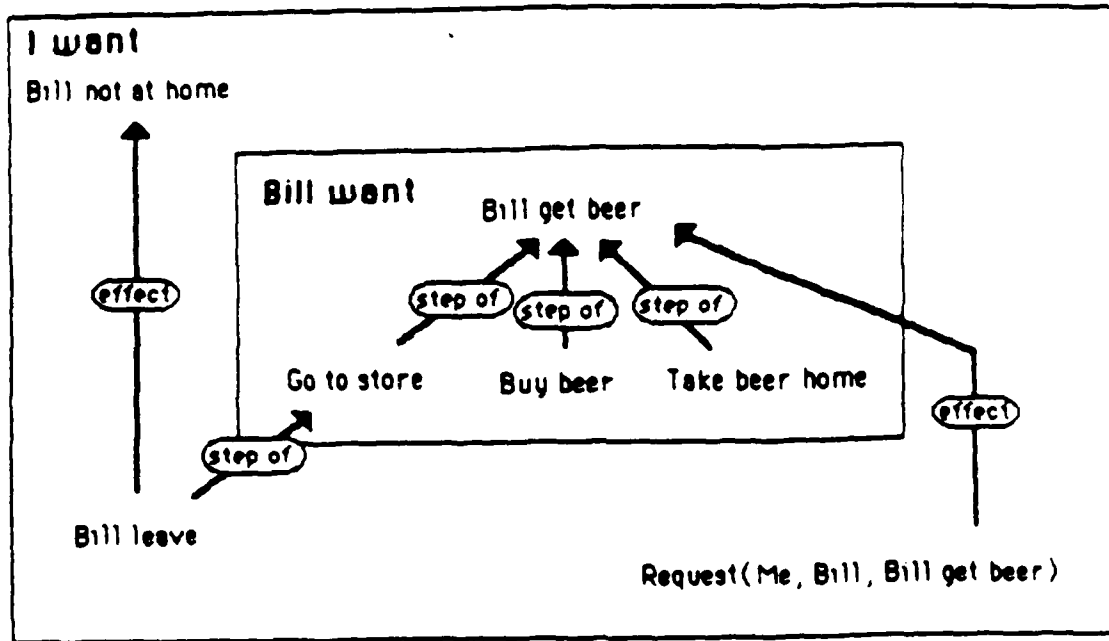
CAUSE-TO-WANT(A, S, P)  
precondition: helpful(S, A)  
effect: S WANT P  
body: achieve(SBAWP)

The discourse planning system by Cohen which provided a basis for Allen's work included just such an operator. Decide-inference is thus just another case of forward chaining.

Under the plan construction rules, the know-rule is a case of the backward chaining rules applied to the CHECK-POSITIVE action above. This just leaves the nested planning construction and recognition rules. These rules are used to reason about planning by others. The nested-planning rule states that in order to get another agent to adopt a goal P, get him to adopt a goal Q, where we can depend on him to accomplish P in trying to achieve Q. This sort of indirect interaction is not actually that unusual. Allen gives an example where I want to get my roommate Bill to leave the house, in order to set up a surprise birthday party. Rather than simply asking Bill to leave, I tell Bill to get some beer. Presumably Bill will then generate a plan to get the beer, which will cause Bill to leave the house. Figure 4 illustrates this situation in terms of the usual effect and precondition links. This does not appear to be the standard form of a plan. The graph "doubles back" at the points where the planning context changes from my wants to Bill's wants.

Allen's system straightens out the graph by simply adding a link of unspecified type from the REQUEST(X, Y, "Get beer") to the Y WANT leave(Y). This rule and the corresponding recognition rule can be motivated, however, by considering general principles of rationality. The planning paradigm relies on the fact that an agent plans and acts *in order to* achieve his goals. But this principle can be expressed more strongly. An agent's adoption of a goal will, given certain considerations, *cause* the agent to intend to execute any plan that achieves that goal. The considerations include the conditions that pursuing that goal does not conflict with some more important goal, and that the intended plan is in some sense the "best" plan available for reaching the goal. Very roughly, this is captured by the following action description:





## Nested Planning

Figure 4

ADOPT-PLAN(Agent, Goal, Plan)  
 effect: Agent WANT Plan  
 body: achieve(Agent WANT Goal) where  
     effect(Plan) implies Goal  
      $\wedge \neg \text{exist } G2 . \text{Agent WANT } G2 \wedge \text{more-important}(G2, \text{Goal}) \wedge$   
          $\text{conflict}(G2, \text{Plan})$   
      $\wedge \text{optimal}(\text{Plan}, \text{Goal})$

One might prefer to formulate the body clause above as a precondition clause instead; that would allow one to consider the time which it takes a plan to be constructed. The predicates "more-important", "conflict", and "optimal" are certainly not easy to define; the nested-planning rule simply ignores those condition. But given even a simplified version of the ADOPT-PLAN action, the nested-planning and recognition of nested-planning rules again reduce to instances of backward and forward chaining. Note that the Q mentioned in the nested-planning rule is any substep of the Plan mentioned in the ADOPT-PLAN action.

While Allen described his plan inference rules as rules of "likely" inference, we have presented them here purely in terms of a logic of action, in order to isolate all notions of likelihood or probability in the control strategy. The control strategy selected and expanded highly-rated nodes in the search space of plan graphs. The implementation details of the strategy are not important, but the following five "scoring" heuristics it employed reveal general principles which will appear throughout later work. Again quoting from [Allen 83], they are:

- (H1) Decrease the rating of a partial plan if it contains an action whose preconditions are false at the time the action starts executing.
- (H2) Decrease the rating of a partial plan if it contains a pending or executing action whose effects are true at the time that the action commences.

These heuristics favor plans which do not contain unnecessary actions. The predicate "optimal" in our ADOPT-PLAN action tries to capture the same basic idea. Later we will relate the notion of a minimal model with that of minimizing the number of unnecessary actions in a plan.

- (H3) Increase the rating of a partial plan if it contains descriptions of objects and relations in its alternative that are unifiable with objects and relations in its expectation.

This heuristic attempts to minimize the number of distinct objects mentioned in a plan, which again is an approximate way to minimize the number of distinct action-instances in the plan. As will be discussed later, an operation more general than standard unification is actually required. The heuristic actually implemented is, roughly: if objects in the alternative and expectation could *consistently* be equal, make them equal, and increase the rating of the plan. This step of "making objects equal" may be better understood as an additional plan inference *rule*, since it transforms one plan graph into another, more specific, graph.

- (H4) Increase the rating of a partial plan if the referent of one of its descriptions is uniquely identified. Decrease the rating if it contains a description that does not appear to have a possible referent.

Plan graphs containing impossible descriptions correspond to no actual plan.

(H5) Increase the rating of a partial plan if an intersection is found between its alternative and expectation, i.e., they contain the same action or goal.

This heuristic favors plan graphs which are mostly grounded. Because plan graphs are constructed incrementally, the effect of this rule is to favor *shorter* plan graphs. Thus, like (H1) and (H2), this heuristic favors plans containing a minimal number of actions.

In summary: the major contribution of this work was to present a set of universal plan recognition rules. Some, such as the **know-positive** rule, appeared rather ad hoc; we have attempted to simplify their presentation. Through the control strategy, the system did try to find the *best* interpretation of the observations, not just *any* interpretation. The system handled all three kinds of "explanation" discussed in the introduction. Generational explanations are found by forward chaining on the **body-action** rule, where the body is a proposition; step explanations, by chaining on **body-action** where the body is a sequence of actions; and enabling explanations, by chaining on the **precondition-action** and **action-effect** rules.

The weakest part of the work involves the control strategy. What exactly are the heuristic rules trying to approximate? (We've provided some possible explanations.) What can actually be recognized from a set of observations can only be determined by *running* a particular implementation of the theory. The plan inference rules by themselves are so powerful that practically any alternative could be eventually linked to any expectation. Important issues of default reasoning -- such as making terms equal if they could consistently be so -- are only casually dealt with. Finally, the system does not attempt to deal with *incremental* plan recognition. The next piece of work addresses many of these problems.

### 2.1.2 Litman

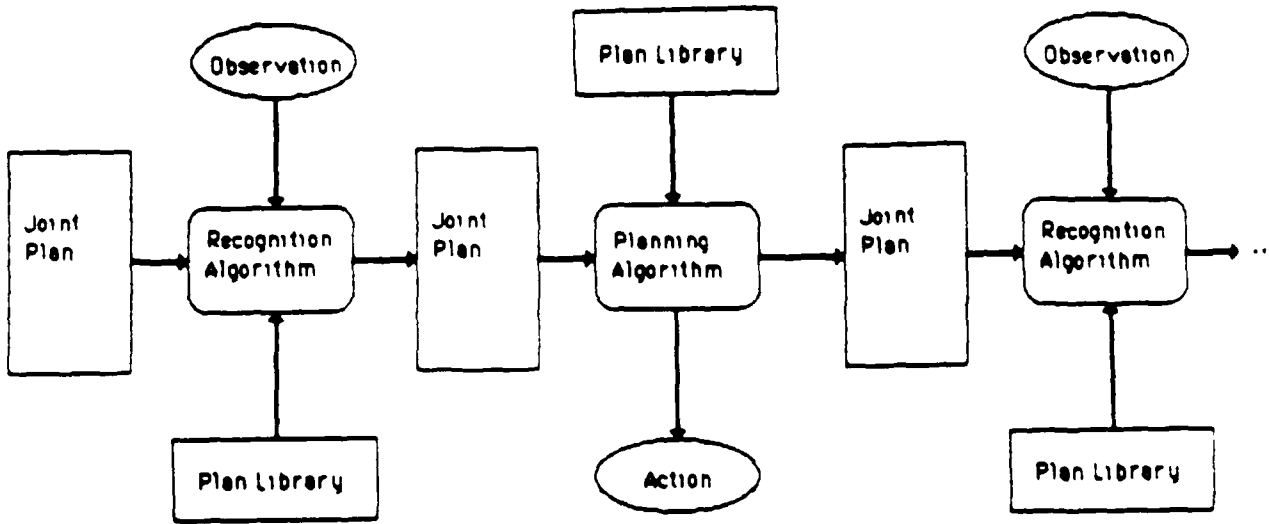
While Allen's work concentrated on how a single utterance could invoke the plan which contained it, work by Barbara Grosz [Grosz 77] tried to connect the structure of a *discourse* to that of an underlying domain plan. Grosz's account was primarily generative: how and in what order can a speaker coherently describe the various steps in a task plan, and how does that plan affect the objects that can be referred to by various kinds of anaphora. While [Sidner & Israel 81] and [Carberry 83] began to extend plan recognition to deal with sequences of utterances, work by Diane Litman [Litman & Allen 84] contains the most detailed and concrete proposals.

Litman's plan recognition system includes a set of domain-specific plan schemas, a set of domain-independent *meta-plan* schemas, and an *incremental recognition* algorithm. The plan and meta-plan schemas are like the action descriptions in the previous section, and are hierarchically organized. Meta-plans are plans which can take other plans as arguments. They include, for example, a plan to help a hearer identify a parameter which appears in another plan (IDENTIFY-PARAMETER), and a plan to insert a repair step into a plan which would otherwise fail (CORRECT-PLAN). The algorithm constructs and updates a data structure which represents the plans being recognized. To be precise, the data structure represents what the observer believes is the state of the *joint intentions* of the actor and observer. The joint intentions of two agents are, roughly, those things which they mutually believe they both want. The observer is furthermore assumed to be totally cooperative, so that whatever it learns of the actor's wants automatically becomes part of its wants as well.

Plan recognition is performed incrementally by applying the recognition algorithm to the current version of the joint plan after each action (utterance) by the actor. The observer may also *plan* and execute a response, and appropriately update the joint plan. Implicit temporal constraints exist between succeeding steps of subplans in the joint plan. At any stage certain actions may be labeled as LAST or NEXT, to indicate the most recent finished action and nearest future action, respectively. Thus the joint plan includes both past and future intentions. Figure 5 illustrates this process.

Litman's system is notable in providing a simple, non-probabilistic, highly constrained recognition algorithm. Many of the complications of the control strategies of earlier systems are eliminated by declaratively encoding knowledge of the planning process in meta-plans, and by employing constraint propagation techniques. For example, cases handled by the rules *know-term* and *know-value* in Allen's system turn out to involve the IDENTIFY-PARAMETER meta-plan. Litman's work is also unique in cleanly accounting for plan suspension and resumption.

We now consider the system in more detail. The joint plan is built as a *stack*. At the bottom of the stack is actor's domain-specific task plan. The actor expands and executes this plan until it must, for some reason, be suspended. For example, the actor may require more information in order to actually execute one of the basic actions in the plan. A meta-plan, such as IDENTIFY-PARAMETER, which takes the lower plan as a parameter, is then pushed on the stack and expanded. In the course of execution, this meta-plan may be suspended as well, and a meta-plan affecting it pushed on top. When an action is actually executed, it must be a step of the plan on the top of the stack. When a plan completes execution, it may be popped from the stack.



**Incremental Plan Recognition/Planning**

**Figure 5**

The plan recognition algorithm attempts to reconstruct as much of the stack as is unambiguously determined by the actions so far observed. When it observes an action, the system attempts to attach somewhere on the stack, according to the following preferences:

1. Attach to the plan on the top of the stack;
2. Attach to a new meta-plan, which refers to a plan somewhere in the stack, and push that meta-plan onto the stack;
3. Attach to a new meta-plan, which refers to some other new plan. If that other plan is also a meta-plan, construct a plan for it to refer to, and so on, until a domain-specific plan is reached. Push everything onto the stack, with the domain-specific plan on the bottom and the original meta-plan on top.

To attach an action to a plan one finds a chain of body (decomposition) links from the observed action to a step of the plan. Since one is merely climbing a tree, this process is fast and always terminates. (This assumes that the agent is actually performing some combination of known plans. Recent work by Martha Pollack investigates cases where the agent is attempting to perform a "buggy" plan, so that the recognizer cannot link the agent's actions to a plan which appears in, or which can be synthesized from plans which appear in, the common plan library.) The other, more computationally explosive types of forward chaining are handled by some of the meta-plans. Cases may arise where there are at the same preference level several mutually incompatible ways to attach an observation to the stack. In such cases the algorithm creates a copy of the stack for each alternative, and *stops* chaining. Later observations should allow the system to determine which alternative is the one intended by the actor. It is important to note that the actor is always executing an action from the plan on the top of his private plan stack. He may, however, have popped the stack or pushed a new meta-plan before acting. Therefore the recognition algorithm may need to manipulate the entire stack before incorporating the action. The ordering of the preference rules indicates that the observer is trying to build a model of the hearer's intentions which introduces as few new plans as possible.

Some of the parameters to a partially recognized plan may be only partly specified. As the plan is built up, however, constraints arise between various parameters. When one plan is attached to another, parameters which may *consistently* be set equal are in fact made equal. This kind of consistency unification, as mentioned in the previous section, serves to minimize the size of the overall plan. A partially specified parameter to a meta-plan may, of course, be a plan. Performing consistency-unification between a plan parameter and a domain plan can have the effect of all of the knowledge precondition, want precondition, and nested planning rules in Allen's system.

Table 2, taken from Litman's paper, illustrates the INTRODUCE-PLAN and IDENTIFY-PARAMETER meta-plan schemas. INTRODUCE-PLAN says that in order for a speaker to get a cooperative hearer to want to perform a plan, the speaker may request that the hearer perform some step of the plan. When the hearer recognizes INTRODUCE-PLAN, the plan recognition algorithm attempts to make the parameterized plan equal to some present or domain-specific plan, using consistency unification. Thus this meta-plan roughly corresponds to the nested-planning rules. The IDENTIFY-PARAMETER meta-plan, as mentioned before, is used to handle information-gathering actions as part of meta-plans concerning domain (or other meta) plans, rather than as direct expansions of the preconditions of domain plans.

---

HEADER: INTRODUCE-PLAN(speaker, hearer, action, plan)  
DECOMPOSITION: REQUEST(speaker, hearer, action)  
EFFECTS: WANT(hearer, plan)  
          NEXT(action, plan)  
CONSTRAINTS: STEP(action, plan)  
              AGENT(action, hearer)

---

HEADER: IDENTIFY-PARAMETER(speaker, hearer, parameter, action, plan)  
DECOMPOSITION: INFORMREF(speaker, hearer, term, proposition)  
EFFECTS: NEXT(action, plan)  
          KNOW-PARAMETER(hearer, parameter, action, plan)  
CONSTRAINTS: PARAMETER(parameter, action)  
              STEP(action, plan)  
              WANT(hearer, plan)  
              PARAMETER(parameter, proposition)  
              PARAMETER(term, proposition)

---

Table 2: Meta-plans from [Litman & Allen 84]

In the examples considered in Litman's paper, the domain plans themselves are so highly structured that there is no need for "pure" precondition-action or action-effect chaining. Whenever two plans are serially linked, they are either substeps of some common plan which already appears in the plan library, or one plan involves knowledge or want preconditions of the other. Our knowledge of discourse may indeed be almost entirely precompiled in this way. The assumption may be less reasonable if one considers plan recognition in general, where the actor is not necessarily overtly trying to make his intentions obvious to the observer. Suppose you observe me buying sugar, butter, eggs, and chocolate chips at the supermarket. Then you would be justified in concluding that I intended to bake some cookies. But while we probably know of a plan such as a SHOP-FOR(item) and COOK(food), do we really want to conclude that we necessarily have a plan such as SHOP-FOR-THEN-COOK(item, food)? It seems more reasonable to assume that some general rule is connecting the two plans. A meta-plan such as the following could be used to express such a rule:

HEADER: ESTABLISH-PRECONDITION(Plan1, Pre, Plan2)  
EFFECT: Pre  
          LAST(Plan1)  
          NEXT(Plan2)  
DECOMPOSITION: Plan1  
CONSTRAINT: effect(Pre, Plan1)  
            precondition(Pre, Plan2)

The problem with such a meta-plan, of course, is that it may be just too general, leading immediately to too many ways to consistently unify its unbound parameters. It may be better to have a number of more specific metaplans: for example, one that establishes "have resource" preconditions, just as IDENTIFY-PARAMETER establishes knowledge preconditions.

In summary, Litman's work demonstrates the power of metaplans, constraint accumulation, and consistency unification for incremental plan recognition. More knowledge of planning and plan recognition is declarative, rather than being hidden in a control strategy. Like Allen's system, Litman's tries to find the *best* explanation for the

observations. The ordered preference rules seem closely related to our notion of minimal model construction. Litman does not provide a rigorous formal treatment of plans or plan recognition. No distinction is made in the work between a particular data structure used to represent a plan and what a plan "really is". While an action or (non-meta) plan may be thought of as a function from state to state, as in the situation calculus, it is not at all clear what a meta-plan *is*. Finally, the precise links between the particular meta-plans she presents (such as INTRODUCE-PLAN) and general principles of rationality and planning are not made (although they no doubt could be). The next piece of work we examine attempts a formal reconstruction of planning and plan recognition from basic principles, but makes some important trade-offs in the process.



### 2.1.3 Cohen and Levesque

Recent work by Phil Cohen and Hector Levesque [Cohen & Levesque 80, Cohen 84] has attempted to formally characterize the space of possible inferences available to an agent engaged in conversation. They have deliberately not dealt with selecting between alternative interpretations of an observation, and have so far been concerned only with single utterances. Their theory is presented as a set of axioms which elaborate many of the notions captured by Allen's plan inference rules. Dynamic logic [Harel 79] combined with a modal epistemic logic provides the framework for the theory. The goal of the work is to provide a semantically clean foundation for plan based theories of discourse, based on general principles of rationality. They claim that philosophical accounts of language which simply postulate "speech acts" are too unprincipled, and that speech acts can be formally derived from these more general principles. We will not attempt to evaluate these linguistic claims, but will examine those axioms that describe general cooperative behavior. These axioms employ a vocabulary for action which is considerably richer than that usually encountered in formalizations of planning. Terms such as "eventually", "causes", and "can" are given definitions in dynamic logic. While Cohen and Levesque have deliberately tried to keep their system as simple as possible, and do not pretend to offer adequate definitions of these very difficult concepts, difficult technical problems do arise. One might also question how intuitive and general are the axioms for plan recognition. My own work has emphasized the non-monotonic aspects of plan recognition ignored by Cohen and Levesque. Toward the end of this section I shall suggest how it might be possible to merge the two approaches.

Dynamic logic is a modal logic for reasoning about action. Actions are semantically interpreted as reachability relations over instantaneous states. Where  $p$  and  $q$  are formulas, the formula:

$$(\text{imply } p (\text{Result agent act } q))$$

means that the result of an agent performing act when  $p$  holds necessarily results in  $q$  holding. Semantically, this means that  $q$  holds in all worlds reachable by the action specified by agent doing act from any world where  $p$  holds. (Note that **Result** is a modal operator.) A solution to a planning problem is a formula of the above form, where  $p$  describes the initial state,  $q$  the goal state, and act is some compound action.

Cohen and Levesque introduce the modal operators **BEL** (believe) and **GOAL**, and define know and mutually-believe in the usual way. In order to reason about planning we need to be able to state such things as: that a state was actually reached by performing a particular action; that some action will *eventually* be done; that in a state, an agent is *able* to successfully perform an action; and that *doing one thing causes* another thing to happen. Once these terms are defined, we can state the plan recognition rules as axioms. For example, the rule that corresponds to forward chaining along precondition-action links can be stated as follows:

Shared-recognition precondition/effect:

```
(imply
  (BMB y x
    (and (or (CAUSE x p (CAN x q))
              (CAUSE x p (CAN y q)))
          (EXPECT y (GOAL x q))
          ¬(GOAL y ¬q)
          (HELPFUL y x)))
    (CAUSE x
      (BMB y x (GOAL x (GOAL y p)))
      (BMB y x (GOAL x (GOAL y (FINTELY-WAIT-FOR x q))))
    )
  )
```

The following example helps illustrate this axiom. Suppose that you (y) and I (x) mutually believe that if the door were opened, you could leave the room. Formally:

```
(or (CAUSE me (open door) (CAN you (you leave)))
    (CAUSE you (open door) (CAN you (you leave))))
```

Furthermore, you expect that eventually I will want you to be gone, and you have nothing against going, and you are helpful to me.

```
(and (EXPECT you (GOAL me (you leave)))
     ¬(GOAL you ¬(you leave))
     (HELPFUL you me))
```

This satisfies the antecedent of the first implication:

```
(BMB you me
  (and (or (CAUSE me (open door) (CAN you (you leave)))
          (CAUSE you (open door) (CAN you (you leave))))
       (EXPECT you (GOAL me (you leave)))
       ¬(GOAL you ¬(you leave))
       (HELPFUL you me)))
```

Therefore, whatever I do to make it mutually believed that I want the door opened (such as saying, "Open the door!"):

```
(BMB you me (GOAL me (GOAL you (open door))))
```

will also make it mutually believed that I do in fact want you to leave. That is, whatever I do will cause you to believe that I want you to have the goal of having me not have to wait forever for you to be gone:

```

(CAUSE me
  (BMB you me (GOAL me (GOAL you (open door))))
  (BMB you me
    (GOAL me (GOAL you
      (FINITELY-WAIT-FOR me (you leave))))))
)

```

Putting this all together, we have an instance of the shared-recognition precondition/effect axiom:

```

(imply
  (BMB you me
    (and (or (CAUSE me (open door) (CAN you (you leave)))
           (CAUSE you (open door) (CAN you (you leave))))
         (EXPECT you (GOAL me (you leave)))
         -(GOAL you -(you leave))
         (HELPFUL you me)))
  (CAUSE me
    (BMB you me (GOAL me (GOAL you (open door))))
    (BMB you me
      (GOAL me (GOAL you
        (FINITELY-WAIT-FOR me (you leave))))))
)
)

```

So far Cohen and Levesque have not tried to introduce body-action type chaining axioms into their system. This is in sharp contrast to Litman's system, which relies on body-action chaining exclusively. It appears that considerable effort is required to even devise a way to state that an action is part of a longer plan (as opposed to *enabling* another plan), since plans are not quite "first-class objects" in the logic; one can only write formulas about the *result* of executing a plan.

While newer work may change the details of these axioms, several points are worth noting. First, all the *possible* interpretations of an action *logically follow* from the action. In the above example, if you also expected that I would eventually have the goal of getting a breath of fresh air, you would have interpreted my request to open the door *both* as a request to leave and a request to air out the room. This may or may not be reasonable. Worse, if I had said, "Close the door!" in the above example, the axiom could have been used to prove that I wanted you to leave, since having the door closed enables you perform the complex action of *opening the door again, and then* leaving the room!

Cohen and Levesque would respond that such interpretations, however bizarre, could be correct in *some* circumstances, and so should follow from the logic. It appears that blatant contradictions (such as concluding that I want both p and not-p) can be avoided by stating all conclusions in terms of what my goals will *eventually* be. (I can consistently want you eventually to be here, and also eventually to be gone.) Furthermore, the addition of more complicated "gating conditions" (the first antecedent above) can further constrain the applicability of the axiom. Indeed, it is already highly constrained by only considering plan recognition where all relevant facts are mutually believed by all parties.

Cohen and Levesque hope to provide a logical foundation for reasoning about interaction and cooperation, but not at this point work towards a practical system. [Blenko 85] discusses some of the problems encountered in working out the semantic details of the various operators (in particular, FINITELY-WAIT-FOR), and notes that a formalism which treats actions as objective *events*, rather than accessibility relations, may simplify matters. (The CAN operator is also troublesome. It appears that (CAN agent goal) is true whenever the agent knows of any plan, of any degree of complexity, that achieves the goal. But then the antecedent of the precondition/effect rule is too weak, since CAN of almost any proposition is always true.) But in addition to these logical details, one must worry about how the various plan recognition axioms should be *justified*. A *foundation* for planning and plan recognition should systematically link rules for planning with rules for plan recognition. Otherwise the axioms will tend to appear as more or less arbitrary heuristics, rather than general principles of rationality.

My own work has concentrated on the problems of multiple interpretations and multiple observations not dealt with by Cohen and Levesque. I believe that some sort of non-monotonicity must be introduced to handle these problems. It is no doubt possible to extend Cohen and Levesque's system along these lines. For example, suppose the antecedent to the precondition/effect axiom above were strengthened to include the condition  $\neg(\text{ABNORMAL } p \text{ } q \text{ } x \text{ } y)$ . Then we add an axiom stating that ABNORMAL holds just in case  $p$  also enables some other goal which is incompatible with  $q$ . By circumscribing ABNORMAL, we can ensure that the precondition/effect rule only applies in unambiguous cases. This is, of course, only a very vague suggestion; still, one would hope that something like this could be done, since Cohen and Levesque's work is one of the most ambitious attempts to date to provide a logic for rational interaction.

## 2.2 Automated Consultation

A natural application area for the discourse systems discussed above is the human/computer interface itself. The Argot project at the University of Rochester, for example, worked largely with dialogues between a computer system user and the (at that time human) operator. A project at BBN studied dialogues between a user and a (simulated) program that edited data structures. It is not surprising, therefore, that plan recognition is a central component of several programs of research aimed at creating automated consultants, systems which would help a person use a particular, complicated program, or perhaps an entire operating system.

One of the earliest automated consultants [Genesereth 79] helped people use MACSYMA, a powerful program for manipulating symbolic equations. Genesereth first created a model, MUSER, of how a user typically breaks down a task when using MACSYMA. This model relates the task, or plan, structure to the structure of the formulas being manipulated. Plans are represented as procedural nets [Sacerdoti 77], together with input/output links between various steps. (The input-output links play both the role of the plan parameters, and the precondition/effect conditions, in the systems described above.) Genesereth suggested that plans could be viewed as a kind of structured dataflow graph. A library contains both common plans, and common mistakes.

When a user had a problem with MACSYMA, he would invoke the advisor, and tell it both his intended goal and what he had actually done -- using Allen's terminology, he would give it the expectation and the alternative. The advisor then attempts to construct a plan graph which connects the two. The plan graph is constructed heuristically, and may contain errors, such as unsatisfied preconditions. The advisor then debugs the plan and tells the user what to do.

The advisor used an ordered set of plan recognition rules, which are very similar to those used by Allen and Litman. The rules apply deterministically: the partial plan is expanded only in unambiguous cases. An "escape hatch" exists in that the advisor can ask the user for clarification if all else fails. The rules, in brief, are:

1. Propagate constraints through the plan, along input/output links.
2. Expand downward (plan-body) as far as can be done deterministically.
3. Expand upward (body-plan) as far as can be done deterministically.
4. Try to identify a node from the expectation with one from the alternative.
5. Add a node which is both a subgoal of the expectation and a supergoal of the alternative.
6. Add any entry in the library which contains the alternative.
7. Assume that inputs which cannot be determined are not required. (This reflects a common source of error in user's plans.)
8. Ask the user for more information about the plan.

Genesereth's work raised many issues and techniques which were developed (or rediscovered) by later researchers. Like Litman's system, the consultant provided a "limited" inference mechanism, and could not just chain off in arbitrary directions. He did not, however, formalize the principles above (except in a particular implementation in LISP), or deal with multiple expectations, belief contexts, or multiple plans.

The Programmer's Apprentice [Rich 81] is a consultant for developing and debugging LISP programs. This ambitious work recognizes and debugs program, rather than plan, fragments. It deals with many difficult notions, such as iteration, assignment to variables, and the distinction between a function and the *implementation* of a function, that are not yet of central concern in our own work.

An operating system consultant under development at the University of Massachusetts [Huff & Lesser 82] is notable for dealing with multiple, concurrent plans, and relating plan recognition to parsing. The system tracks user's actions, and allows users to specify high-level actions which are disambiguated by context. The system has a hierarchical library of operating-system level tasks performed by programmers, such as *compile* or *edit*. Part of the task library is a "task grammar", a grammar which allows symbols to be rewritten as extended regular expressions. For example, the plan to update source code is partly described by the rewrite rule:

```
update_source_unit =>
  ((edit compile check_results)
   (edit compile)
   (compile check_results)
   (compile))-
```

Since a programmer may be working on several different projects during the same session, the notion of a regular grammar is extended to that of a *shuffle grammar*. If *e* and *f* are expressions, their shuffle, written *e\$*f**, is the set of strings constructed by mixing together a string of *e* with a string of *f*. The interleave of an expression *e*, written *e@*, is the expression shuffled with itself, an arbitrary number of times. For example, the fact that several unrelated programs may be worked on simultaneously is represented by the grammar rule

```
programming_work =>
  (do_programming
   do_documentation
   make_errors)@
```

The intelligent interface tries to parse the (partial) input of the user as it is received. It employs heuristics for ordering alternative partial interpretations of an observation when parsing. (Huff and Lesser note that the grammar is, in general, context-sensitive, so that any practical parser must be heuristic-based.) The heuristics try to "minimize" the amount of mixing performed by the shuffle operator, and the number of shuffles invoked by the interleave operator. For example, it should prefer the shuffle *eeefffff* over *eeffefff*, which is preferred over *efefefef*. Likewise for interleave, *s* is preferred over *s\$\$s*, which is preferred over *s\$\$s*. The heuristics include:

1. Prefer (linking an observation to) an existing plan instantiation over creating a new instantiation.
2. Prefer a new related instantiation to a new unrelated one.
3. If alternative interpretations both appear in the same higher-level containing plan, prefer the interpretation which appears first in that higher-level plan.

It is important to note that these heuristics, which are crucial to the success of the system, stand outside the plan grammar. Huff and Lesser justify them simply on the grounds that people don't jump around at random when they work. As with Cohen and Levesque's work, the formal part of the work only outlines the space of all possible, context-independent explanations for an action, instead of the smaller space of reasonable explanations.

## 2.3 Expert Systems

Few connections have been drawn between work on plan recognition and expert systems. Yet an early paper by Harry Pople, who was later to become famous for the INTERNIST expert system, explicitly made the connection between diagnosis, plan recognition, and even perception, analogy, and intuition as examples of abductive reasoning [Pople 73]. Earlier attempts to automatically generate abductive hypotheses (statements which would entail the observations) exhaustively had not been promising. The principle of Ockham's Razor suggests that a good explanatory hypothesis should imply all the data, and yet be concise as possible. Pople described how this could be implemented in a system based on resolution. The method is to attempt to generate a linear resolution proof of the observations from the domain axioms. This proof will not succeed, of course, but certain of the leaves of the partial proof tree will represent possible abductive hypotheses. Ockham's razor is applied by factoring across partial trees -- that is, by combining lower leaves of the tree. This process is illustrated in figure 6. The result of such factoring, which Pople called *synthesis*, should be a hypothesis (leaf) which explains (entails) several pieces of the data. Synthesis corresponds to the unification heuristic in Allen's plan recognition system. As I have argued, synthesis is thus a special case of the more powerful consistency unification operator which is actually required in order to implement Ockham's razor. No doubt drawing on Pople's language, Eugene Charniak has used the term *abductive unification* for what we call consistency unification.

More recently Reggia, Nau, and Wang [Reggia et al 83] have proposed a formal model of expert systems based entirely on Ockham's razor. They associate a set of possible manifestations with each disease. Given a set of symptoms (observed manifestations)  $M+$ , the problem is to determine a set of diseases which *covers*  $M+$  and is *parsimonious*. In simple cases my formulation of the plan recognition problem also reduces to this kind of minimal cover problem.

Perhaps the most interesting work on diagnostic systems involves building "deep" models of the causal processes that create the symptoms [Kunz 83]. It may be possible to relate the construction of models of an agent's intentions in a plan recognition system to these kinds of causal models.

The literature on expert systems is growing explosively, but I will not attempt to make further connections with it. Most of the work emphasizes matters of efficiently handling large numbers of facts, and of course ignores all issues of beliefs, goals, and rationality. In some ways, too, the plan recognition problem is *easier* than most of those faced by expert systems, since we can assume that we are dealing with cooperative agents and not obstreperous diseases. Yet there are tantalizing parallels in the two areas; for example, the problem of control *focus* in either a plan recognition or expert system.

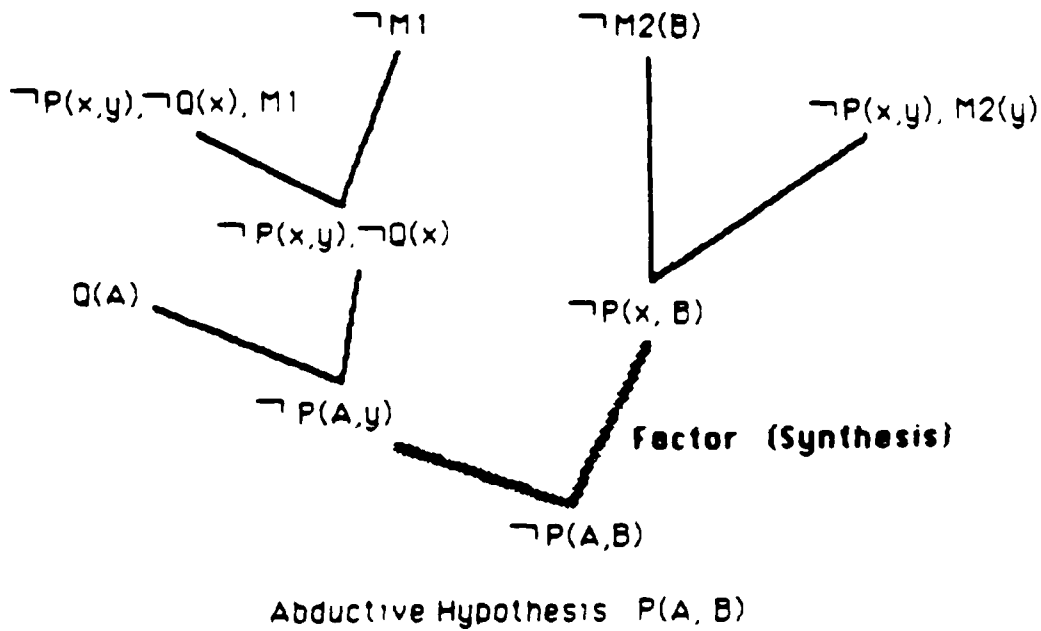


Domain axioms

$P(x,y) \wedge Q(x) \supset M1$   
 $P(x,y) \supset M2(y)$   
 $Q(A)$

Symptoms  $M1, M2(B)$

Linear resolution tree



**Pople's mechanized abductive logic algorithm**

**Figure 6**

## 2.4 A Psychologically-Based System

One of the first papers to explicitly invoke the phrase "plan recognition" was the report by Schmidt, Sridharan, and Goodson on the BELIEVER system. BELIEVER was designed to illustrate and test a psychological theory of "how descriptions of observed actions are utilized to attribute intentions, beliefs, and goals to the actor." [Schmidt 78] Schmidt and his colleagues conducted experiments in which human subjects were presented simple linguistic descriptions of sequences of actions by a single agent. The descriptions were deliberately made stylistically "flat", in order to avoid issues of narration and rhetoric. The sequences were interrupted at various points, and the subjects were asked to summarize events so far, describe what the agent was trying to do, or predict what the agent would do next. The researchers observed that:

1. Summaries often included non-described, but expected, actions. The temporal order of non-causally related events was poorly remembered.
2. Subjects did not provide summaries which referred to a disjunctive set of plans, but they did provide "sketchy" summaries.
3. Subjects often provided summaries of the form: "The agent was *trying to do* (some act), but failed *because* (some reason)."

From these and similar observations, Schmidt concluded that people understood and remembered event sequences by recovering the implicit structure of causal relations between the events. This suggests the psychological reality of relationships such as *enables* or *in order to* between actions. (Note that such relations are not made *explicit* in many formalizations of planning.) The data also suggests that plan recognition is a single-minded, hypothesis-driven process. Based on the initial observations (descriptions), the subjects seemed to devise a single hypothesized plan (for the actor). This hypothetical plan would be incrementally revised and made more detailed as further observations were made.

BELIEVER implemented this strategy of plan recognition. On the basis of a model of the "world" of an agent, and a series of statements about actions by the agent, the system constructed and maintained a data structure called an *expectation structure*, which recorded the system's hypothesis about the agent's intentions. The expectation structure contained a single plan graph, similar to those described in the section above on James Allen's work. This plan graph consisted of a set of actions and propositions, some of which corresponded to observations, and others, to domain-specific goals, partially ordered by *in order to* and *enables* links. Unlike Allen's later approach, the expectation structure was assumed to be connected.

The initial expectation structure could be invoked in several ways. If BELIEVER was simply told the actor's overall goal, it would construct a plan by backward chaining. If BELIEVER was told the setting, it would retrieve a "canned" parameterized plan from memory. Finally, BELIEVER could try to infer candidate goals from the observed actions. While Allen concentrated on this final type of inference, it was only treated in passing in the work on BELIEVER.

The actions in an expected plan typically had many unbound parameters. As each observation was input, BELIEVER tried to match the description against an *expected action* (that is, an action in the expected plan whose preconditions were true). A match could bind parameters in the action; these bindings would be propagated through the plan, possibly triggering other bindings. Failure of the observation to match an expected action would trigger *hypothesis revision critics*. These revision critics were also invoked if the system noticed that some event had occurred which would mean that the expected plan would not work as it stood.

The plan revision critics could change the expected plan in many different ways, by patching in pieces to take care of "accidents", adding subplans to "undo" side-effects of plans, changing a goal to a closely related goal (e.g., change the goal *eat* to *drink*), and so on. The aim was to be able to recognize plans which contain errors and mishaps. For example, given the sequence:

John went to the cabinet.  
John took out a record.  
John went to the hi-fi.  
John dropped the record on the floor.

we would want the system to recognize that John has the plan of playing the record, and that John had an accident, and not simply give up since there is no reasonable domain plan for throwing records on the floor. Given this input, a revision critic in BELIEVER would probably add a patch to the expected plan to re-achieve the state *holds(John, record)*, which would enable the expected action *put-onto(John, record, hi-fi)*. Once this patch was made, BELIEVER would then expect the action *pick-up(John, record)*.

The work on BELIEVER is important for stressing the importance of recognizing failed and erroneous plans, and the techniques of incremental binding of plan parameters. The latter technique was stressed in later work on incremental plan recognition, such as Litman's work discussed above. Recent work by Martha Pollack [Pollack 84] investigates problems in recognizing erroneous plans. Schmidt was unable to provide, however, any interesting global strategies for controlling the all-powerful plan revision critics. No distinction was made between plan revisions due to problems in the described world, and those due to non-monotonic inferences made by the observer itself. Aside from the kind of plan abstraction which arises from using plans with unbound parameters, no use of hierarchical planning was used. The issues of bottom-up goal inference were, as noted above, largely ignored. Schmidt did note that the use of plan hierarchies could lead to much more powerful bottom-up plan recognition algorithms.

## 2.5 Story Understanding

Plan recognition forms the basis for much work in *story understanding*. As with the BELIEVER system, the goal of that work is to build a model of the situation described by a piece of text, and in particular, to model the "beliefs" of the characters described by the text. (Belief is in quotations because fictional characters naturally cannot have real beliefs.) Adequacy of the model is typically tested by using it to answer questions about the original story. Story understanding requires knowledge about rhetorical techniques (e.g., Lehnert's "plot unit" theory [Lehnert 80]), knowledge of the physical world described by the story, and most relevant to our work, knowledge of human interaction, as simulated by the characters in the story. The work on plan-based theories of discourse was partly inspired by work on plan-based theories of character interaction in Bruce's study of children's stories [Bruce 81]. Shank's theory of *scripts* was designed to account for all kinds of regularities in the world, both physical and social [Schank 75]. For example, the restaurant script tells us that restaurants typically have tables and chairs, and that a person in a restaurant typically has the goal of eating.

While scripts alone only provide a limited version of plan recognition, Wilensky [Wilensky 82] has extended Schank's conceptual dependency theory to handle sophisticated theories of planning and plan recognition. Wilensky is interested in planning problems which involve little or no search ("canned" plans are known for all possible goals) but do involve complicated interactions between goals. Wilensky argues that most everyday planning problems are of this sort. In Wilensky's theory, production rules, called *themes*, relate situations to the goals induced by the situations. For example, a theme such as *Maintain-Bodily-Comfort* might invoke the goal *stay-dry* when the planning agent is in a situation (or is thinking about a situation) in which it is raining. An invoked goal is expanded by its standard plan. Certain *meta-themes* look for interesting interactions between active plans and goals. For example, the agent may have a goal to go outside and get the newspaper, as well as the goal not to get wet, which was invoked because it is raining. The *Achieve as Many Goals as Possible* meta-theme fires in this situation, and adds a goal to *resolve conflict*, where this meta-goal includes pointers to the conflicting plans. The planner then finds a meta-plan, such as *Replan* or *Abandon Goal*, in order to achieve this meta-goal.

The data structures used to represent themes, plans, and meta-plans can be used interchangeably for planning or plan recognition (which Wilensky calls *understanding*). For example, given the following text:

John wanted the newspaper.  
It was raining outside, so John called for his dog Spot.

a system built on this theory could conclude that John wants Spot to fetch the newspaper by reasoning as follows: Having the newspaper is one of John's goals. The standard plan for this is to go outside and get the newspaper, and so this standard plan is (tentatively) included in John's wants. The fact that it is raining would invoke the *stay-dry* goal, so this is added to John's wants. The system then simulates John's planning: the goals conflict, so a *resolve conflict* goal is also created. There are many different ways that *resolve conflict* can be achieved, so the recognizer stops planning. Now the system learns that John called for Spot. The effect of this is determined to be that Spot is with John. The system tries to *connect* this new piece of input, as tightly as possible, with the current plan. It notices that one expansion of the *replan* meta-plan for the *resolve conflict* meta-

goal is to alter the *get paper* plan so that some other agent goes outside and gets the paper. Making Spot the other agent connects this plan to the input.

Wilensky's meta-plans are roughly comparable with the plan construction and plan recognition rules in Allen's system, but have the advantage of being a single set of rules. Meta-plans to *Avoid Wasting Resources* and *Maximize Value of Goals Achieved* encode knowledge about what is a *reasonable* plan, instead of hiding all such knowledge in a control procedure, or in rating rules. Wilensky's notion of a meta-plan is similar, but not identical, to Litman's. Litman's meta-plans, we recall, are not invoked to achieve explicit meta-goals, but instead are recognized when their bodies occur, or planned for when their (non-meta-level) effects are desired. But whenever a meta-goal appears, it is *immediately* attached to meta-plan, so the difference does not appear crucial.

Several important aspects of plan recognition are not formally encoded as meta-plans, but are simply stated as "text comprehension principles", presumably hardwired into the system. The most important of these are: coherence, least commitment, and parsimony.

**Coherence** apparently means that the recognized plan is consistent. This principle may be related to the "consistency" part of the consistency unification algorithm discussed earlier.

**Least commitment** means that a recognizer shouldn't assume that *any* particular explanation found is *the* explanation, and then have to undo it. This principle is also reflected in Litman's aim of designing a largely *deterministic* plan recognition algorithm.

**Parsimony** means that the explanation should "maximize the connections between the inputs". Parsimony may be related to my notion of a minimal model of an agent's plans, as discussed in Chapter 4. Parsimony is an old idea in A.I. circles; but no one really knows how to state it both precisely *and* sensibly! (One crucial problem involves inference chains: if an arbitrary number of inferences are allowed, then usually an arbitrary *number* of connections can be drawn between inputs.)

While Wilensky has implemented parts of this theory in many computer programs, much of it (especially the principles above) remains vague. Wilensky has not dealt with the problems of encoding temporal relations (except for those induced by causality), and the problems of belief contexts (such as "quantifying in" or nested and mutual belief).

## 2.6 Statistical Inference

Probability theory provides the basis for reasoning from observations to supposed causes in practically all the sciences and humanities, with the exception of most areas of artificial intelligence. The overwhelming success of statistical inference in all these diverse fields for the last few hundred years should at least cause the computer scientist to hesitate a bit before devising a new sort of theory. Indeed, after largely rejecting probability theory a decade ago, a growing number of researchers are building new systems (or re-describing old systems) for such tasks as vision or diagnosis based on classical or more recent theories of probability. In this section I will describe (part of) plan recognition in terms of very elementary statistics. My own work in plan recognition, and much of that cited above, can be viewed a method of performing certain steps in the probabilistic inference. A logical theory of plan recognition, towards which I am working, also addresses issues about which classical probability theory has little to say: most importantly, the problem of when a likely statement should be *accepted* as fact. The literature on statistical inference is, of course, colossal, and I have neither the space nor competence to review it here. I merely hope to suggest that my work is complementary to statistics, and is not a case of reinventing the wheel. The discussion which follows is largely based on comments made in a talk by Eugene Charniak.

Let  $H_1, H_2, \dots$  be various hypotheses about an actor's intentions, and  $A_1, A_2, \dots$  be various actions. We identify the set of  $H_i$  with the set of (known) plans. Part of the plan recognition problem is to determine which plans are likely on the basis of observed actions. That is, where  $P$  is the probability function, and  $A = \{A_1, A_2, \dots, A_m\}$  represents the observations, find a set of hypotheses  $H = \{H_1, H_2, \dots, H_n\}$  such that  $P(H|A)$  is large.

Bayes formula tells us that the following holds:

$$P(H | A) = \frac{P(A | H) P(H)}{P(A)}$$

Given a fixed set of observations  $A$  and certain simplifying assumptions, it is possible to determine the *relative* probabilities of various candidate hypothesis sets  $H$ . Assume that every plan (every  $H_i$ ) has about the same prior probability  $k$ , and exactly one possible expansion (body). In cases where a plan should have multiple expansions, introduce an  $H_i$  for each expansion, and then define the original plan as the disjunction of the expansions (e.g.,  $P = H_1 \vee H_2 \vee \dots$ ).

Let us begin by considering the case where the actor only performs exactly one plan - that is, the  $H_i$  are disjoint. To calculate relative probabilities, the denominator  $P(A)$  can be ignored, since it is the same for all candidate  $H$ 's.  $P(H) = k$  (since  $H$  must be unary) and

$$P(A | H) = \begin{cases} 1 & \text{if } (\forall A_i \in A) (\exists H_j \in H). \text{stepof}(A_i, H_j) \\ 0 & \text{otherwise} \end{cases}$$

So the most likely plan is simply the one which incorporates all actions. But since plans are parameterized, there are an infinite number of candidate  $H$ 's to consider. We need some way of searching this huge space. Classical statistics seems to have little help to

offer; statistical theory has concentrated on the special case where the various  $H_i$  are of the *same* known form (e.g. the percentage of red balls in the urn) parameterized only by some numerical coefficients. But here we must deal with a large number of fundamentally different plans which take different discrete parameters. The minimal model construction I propose in this paper corresponds to this search. Plan recognition algorithms try to heuristically solve this problem.

Now relax the assumption that the  $H_i$  are disjoint. Now it is harder to compute  $P(A | H)$ :

$$\begin{aligned}
 P(A | H) &= 1 \text{ if } (\forall A_i \in A) (\exists H_j \in H) . \text{stepof}(A_i, H_j) \\
 &= P(A' | H) \text{ where } (A_i \in A') \\
 &\quad \text{iff } (A_i \in A) \wedge \neg(\exists H_j \in H) . \text{stepof}(A_i, H_j) \\
 &\quad \text{otherwise.}
 \end{aligned}$$

All we can tell is that if  $H$  doesn't account for all the observations, then  $P(A | H) < 1$ . If we are comfortable assuming that the  $H_i$  are *independent*, so that

$$P(H_1 \wedge H_2) = P(H_1) P(H_2)$$

then we can note the following: if hypothesis sets  $H$  and  $H'$  are of the same size, and if  $H$  accounts for all of  $A$  but  $H'$  does not, then  $P(H | A) > P(H' | A)$ . Furthermore, if  $H$  and  $H'$  both account for  $A$ , the smaller set has the higher probability. The plan recognition construction I will describe will have similar characteristics.

I have argued that in performing plan recognition (as opposed to more general cause-effect reasoning) it is particularly justifiable to strongly invoke Ockham's razor: to prefer explanations which only require the actor to have as few unrelated intentions at a time as possible. This condition does not fall out of the probabilistic framework so far: it may give a high probability to an  $H$  which does not quite account for all of  $A$ , but would have to be greatly expanded to account for the remainder. This can be remedied by adding additional constraints on our probability function. For example, one such constraint could be:

$$\begin{aligned}
 (\forall H_i) P(H_i) &> P(G) \\
 \text{where } G &= \text{disjunction of } \{ H_i \wedge H_j \mid i \neq j \}
 \end{aligned}$$

In the end, then, probability can give us a way to compare the relative merits of alternative hypotheses about an actors intentions, but does not automatically give us a way of selecting hypotheses to compare. We don't seem to need any of the high-powered mathematical techniques statisticians have developed. And still the problem of deciding what likely statements to accept as true remains.

In my model-theoretic treatment of plan recognition, the statements that the observer should accept are simply those which are valid in the minimal model construction. Using probabilities, we need some sort of rule of acceptance, or at the least be prepared to deal with all the complexities of allowing an agent to hold *degrees of belief*. [Kyberg 74] deals in detail with the complexities of both these problems. The present situation appears particularly difficult, because we have not come up with absolute probabilities, but only a method for ordering the relative probabilities of certain statements. Thus no simple rule -- such as "accept  $H$  if  $P(H) > .95$ " will do.

I have no doubt that a purely probabilistic treatment of plan recognition is possible, and perhaps even desirable. But adopting a probabilistic framework would lead far afield into many difficult areas, and not immediately clarify our understanding of the particular problems at hand.



## 2.7 Non-Monotonic Inference

A system of inference is a method for drawing conclusions from a body of facts. Non-monotonic inference systems have the property that a conclusion may be overturned if the body of facts is enlarged. Systems of non-monotonic inference include probabilistic inference, logics with default rules [Reiter 80], and logics with a circumscription operator [McCarthy 80].

The relation of plan recognition to probability is discussed in the previous section. How useful are default rules for plan recognition? It is straightforward to cast, for example, Allen's plan recognition rules as default inference rules. Using Reiter's logic, we could write, for example, the action-body chaining rule as:

$$\frac{\text{Want}(\text{agt}, \text{act1}) \wedge \text{stepof}(\text{act1}, \text{act2}) : M \text{ Want}(\text{agt}, \text{act2})}{\text{Want}(\text{agt}, \text{act2})}$$

This says that if an agent wants an act that is a step of a higher level act, and it is consistent that (M) he wants the higher level act, then conclude that he wants the higher level act. But little has been gained from the use of default logic. All known systems of default logic share the property that given a set of facts and set of rules, one may reach different and perhaps mutually contradictory sets of conclusions, depending on the order in which one applies the default rules. The logic insures that each set of conclusions, or *extension*, is internally consistent, but gives no way choosing between them. As a basis for plan recognition, then, default logic suffers the same criticisms as the dynamic logic framework of Levesque and Cohen: too much of the problem remains hidden in the strategy which orders the application of various rules of inference. While in Cohen and Levesque's system all possible interpretations of an observation were logically entailed, using a default logic, some more or less random possible interpretation would be logically entailed.

Circumscription is a specialized form of non-monotonic inference developed by McCarthy to handle the "qualification problem" in planning. McCarthy wanted to be able to formally state that "the objects that can be shown to have a certain property P by reasoning from certain facts A are all the objects that satisfy P." [McCarthy 80] For example, we might have a description of a bunch of blocks on a table, and want to make a plan to build a certain kind of tower. It might be necessary to pick up a block B, which can only be performed if B is clear. If we cannot *prove* that there is anything on top of B, then, in general, we want to be able to conclude, by circumscribing the predicate on, that *nothing* is on top of B.

The circumscription of a predicate is a formula of second-order logic which involves the entire collection of facts at hand. McCarthy showed that the circumscription of a predicate is true in all *minimal models* of the original collection of facts, where a minimal model is defined to be one in which there are no unnecessarily true instances of the predicate. While [Davis 80] pointed out that circumscription cannot always capture the notion of a minimal model, [Perlis 85] showed that it can for any predicate with a finite extension.

In the next section of this proposal I will invoke minimal models to describe the inferences performed in plan recognition. For example, from the fact that an action occurs,

one wants to be able to conclude the disjunction of all plans containing that action. That disjunction holds in all models of the plan axioms which are minimal in the predicate used to state that an action occurred. Circumscription serves for this inference. We actually want to infer something much stronger: the disjunction of only the *most succinct* plans which incorporate the observations. Later I show how the notion of a minimal model and the circumscription formula can be strengthened to account for these inferences.

McCarthy's definition of predicate circumscription is as follows. Let  $\psi(p,R)$  be a formula of first-order logic in which the  $n$ -ary predicate  $p$ , and any predicates in the set of predicates  $R$ , appear. The circumscription of  $p$  relative to  $\psi(p,R)$ , where the predicates in  $R$  vary, written  $\text{circ}(\psi(p,R),p;R)$ , is the following formula:

$$\psi(p) \wedge \forall p',R'. [ \psi(p',R') \wedge \forall x . p'(x) \supset p(x) ] \supset [ \forall x . p(x) \supset p'(x) ]$$

The symbol  $x$  stands for a list of  $n$  variables. The lists of varying predicates,  $R$  and  $R'$ , can be omitted if empty. This formula can be understood as asserting that any predicate  $p'$  which satisfies all the conditions imposed by  $\psi$  on  $p$ , and whose extension is contained in that of  $p$ , is exactly the same as  $p$ . That is, there is no *proper subset* of the extension of  $p$  which satisfies all the conditions imposed on  $p$ .

For example, the circumscription of  $p$  relative to  $p(A)$  is:

$$p(A) \wedge \forall x . p(x) \supset x=A$$

That is, that  $A$  is the only  $p$ -thing. The circumscription of  $p$  relative to  $p(A) \vee p(B)$  is:

$$[A=B \wedge p(A) \wedge p(B)] \oplus [p(A) \wedge \neg p(B)] \oplus [p(B) \wedge \neg p(A)]$$

where the symbol  $\oplus$  is exclusive or. It is important to note that circumscription makes no assertions about the predicates in  $\psi$  other than  $p$ , unless the circumscription formula explicitly marks them as *variable*, by including them in  $R$ . These other predicates are called *parameters*, because we can imagine setting them to arbitrary extensions which are consistent with  $\psi$ . One should also note that circumscription does *not* guarantee that the *size* of the extension of  $p$  is as small as is consistent with  $\psi$ . For example, the circumscription of  $p$  relative to

$$A \neq B \wedge A \neq C \wedge B \neq C \wedge [p(A) \vee (p(B) \wedge p(C))]$$

does not entail  $p(A)$ , but only

$$[p(A) \oplus (p(B) \wedge p(C))]$$

The model theory of circumscription is spelled out in precise detail in the next section. For now it suffices to say that a model of a formula is *minimal* in  $p$  if there is no other model of that formula which is identical, except that  $p$  holds of some things in the first model but not in the second. As mentioned above, it is easy to show that the circumscription of a predicate relative to a formula is valid in all minimal models of the formula. The notion of a minimal model is powerful enough to capture notions such as transitive closure or decidability of arithmetic, which cannot be axiomatized. Thus the

second-order circumscription formula does not, in all cases, allow one to formally deduce all statements valid in the minimal models. In many useful cases, however, circumscription is complete for these semantics, and the circumscription formula is in fact equivalent to some first-order formula. [Minker & Perlis 85] show that circumscription is complete if we assume an axiom to the effect that the circumscribed formula has a finite (although possibly arbitrarily large) extension. In the examples in this paper the circumscribed predicates turn out to have only a few true instances, and so the finiteness limitation is no problem.

## Chapter 3

# An Approach to Plan Recognition

In this section the semantic, or model theoretic, aspects of plan recognition are discussed. At least in the simple cases examined, plan recognition corresponds to a sequence of simple model-theoretic operations. This semantic framework can then be used to motivate and justify various plan recognition algorithms.

It is useful to draw a few distinctions which limit the scope of the present proposal. First, we limit ourselves to recognizing instances of plans schematized in a plan library, rather than dealing with the full, arbitrarily complex set of plans that an agent could synthesize in order to meet some set of domain specific goals. Second, we formalize only a part of that knowledge about plans and actions which is common to the observer and agent, and not explicitly represent the separate beliefs, desires, and intentions of the two. Finally, we adopt an event-based framework for planning [Allen & Koomen 83, McDermott 81], rather than the more traditional use of the situation calculus.

A small set of general assumptions underlie the non-sound inferences performed in plan recognition. First, the plan library is taken to be complete: all the particular ways of performing a high-level plan are specified in the library. Second, we assume that all observed actions are purposeful, and can be explained by their appearance in a plan. Third, we invoke a principle of simplicity of intention. An agent is likely to be performing only one or a small number of plans at any time. There may be other general assumptions, but these suffice for the examples in this section. The assumptions appear to be justifiable by appeal to general principles of rationality, as well as the contingencies of plan recognition. These assumptions can be semantically characterized in terms of the minimization of certain predicates.

### 3.1 The Planning Framework

First-order logic provides the framework for planning, where plans, actions, and times, as well as agents and physical objects, are all types of individuals. Instances of plans or actions are designated by constants or constructed by combining the appropriate functions. For example,

pickup(George, Block6, T2)

could be a term representing the action instance of George picking up a certain block during the time interval designated by T2. Functions which designate plans have the same form as action functions. The predicates *do* and *perform* hold of those action instances and plan instances respectively which actually occur. The distinction we are drawing between actions and plans is rather arbitrary, and should eventually be abolished; for now, think of actions as irreducible physical acts, and plans as all more complex acts. Plans and actions are described by a collection of sentences called the *plan library*. Those sentences relating actions and plans are of the following form:

$$\text{do}(a_1) \wedge \text{do}(a_2) \wedge \dots \wedge \text{do}(a_n) \wedge \xi \supset \text{perform}(p_j)$$

Each  $a_i$  is a term designating an action, and  $p_j$  designates a plan.  $\xi$  stands for a subformula which further *constrains* the applicability of this axiom; in particular,  $\xi$  may assert that certain relationships must hold between the arguments (if any) taken by the various action functions. For the time being, we will only consider constant action terms, and make no use of  $\xi$ . The axiom states that if actions 1 through  $n$  occur, then plan  $j$  also occurs. Each sequence of actions to the left of an implication sign is a particular expansion, or body, of the plan to the right. Note that each sequence of actions is a *sufficient*, but not *necessary* condition for performing the plan: there may be multiple possible expansions for the same plan. An agent's *knowledge base* also contains other sorts of axioms, such as those relating a plan to its effects, an action to its preconditions, frame conditions for actions, as well as general domain-specific information, but they will not affect what follows.

A planning system could use a library of this form to determine possible ways of performing a plan. For example, suppose we tell such a system to perform some plan  $P$ . The system must find some expansion for  $P$  which is correct for the current state of affairs. That is, the system should prove a theorem of the form:

$$\omega \wedge \text{do}(a_1) \wedge \text{do}(a_2) \wedge \dots \wedge \text{do}(a_n) \wedge \tau \supset \text{perform}(P)$$

where  $\omega$  is a formula representing what is known about the world,  $\tau$  is an expression constraining the various temporal indexes which appear in the  $a_i$ , and the antecedent of the implication together with the plan library is *consistent*. There are, of course, many ways that this framework for planning can be elaborated.

### 3.2 A Propositional Example

In the following example, action instances are represented by constants, and no use is made of constraint expressions. Let  $L$  be the plan library:

$$\begin{aligned} \text{do}(a_1) \wedge \text{do}(a_2) &\supset \text{perform}(p_1) \\ \text{do}(a_3) \wedge \text{do}(a_4) &\supset \text{perform}(p_1) \\ \text{do}(a_1) \wedge \text{do}(a_3) \wedge \text{do}(a_5) &\supset \text{perform}(p_2) \\ \text{do}(a_2) \wedge \text{do}(a_5) \wedge \text{do}(a_6) &\supset \text{perform}(p_3) \end{aligned}$$

We wish to define a semantic relationship which captures the intuitions discussed above about what plans should be recognized from a set of observations. We write  $B(L) \models F$  to mean that on the basis of plan library  $L$ , having obtained the observations in  $B$  (and *only* the observations in  $B$ ), one may conclude the formula  $F$ . Examples of formulas which stand in this relationship are:

$$\begin{aligned} \text{do}(a_6) (L) &\models \text{perform}(p_3) \\ \text{do}(a_1) (L) &\models \text{perform}(p_1) \vee \text{perform}(p_2) \\ \text{do}(a_3) (L) &\models \text{perform}(p_1) \vee \text{perform}(p_2) \\ \text{do}(a_1) \wedge \text{do}(a_3) (L) &\models \text{perform}(p_2) \end{aligned}$$

Thus, if  $a_6$  occurs, then  $p_3$  must occur, since that plan provides the only explanation for that action. From  $a_1$  (or  $a_3$ ) individually we can only conclude that either  $p_1$  or  $p_2$  is the intended plan, since  $a_1$  (or  $a_3$ ) appears in an expansion of both plans. If *both*  $a_1$  and  $a_3$  occur, then  $p_2$  must be intended, since the actions belong to different expansions of  $p_1$ .

### 3.3 Defining the Plan Recognition Relationship

The relationship will be defined in several steps, corresponding to the three different assumptions discussed above.

#### 3.3.1 Step 1 : The library is complete

The assumption that all ways of expanding a plan appear in the library corresponds to minimizing the set of plans which occur, while the set of actions which occur acts as a parameter. That is, we minimize the predicate `perform`, with `do` as a parameter. This means that whenever a plan occurs, it is entailed that some body of that plan -- some set of actions-- also occurs. We employ the following definitions.

$W(F)$  is the set of (Herbrand) models of the set of formulas  $F$ .

$m_1 \leq[p;R] m_2$  holds just in case  $m_1$  and  $m_2$  are models which agree on all predicates other than  $p$  or those in the set of predicates  $R$ , and the extension of  $q$  in  $m_1$  is a subset of the extension of  $q$  in  $m_2$ .

$\min(M, p;R)$  is the subset of  $M$  which is minimal in  $p$ , where the predicates in  $R$  vary; that is,

$$\{m \mid m \in M \wedge \neg \exists m'. m \in M \wedge m' \leq[q;R] m \}.$$

(As before, the list of varying predicates  $R$  is omitted if empty.) Now consider the formulas which are valid in  $\min(W(L), \text{perform})$ . These include the "completions" of the formulas in the plan library. For the example problem, these are:

$$\begin{aligned} (\text{do}(a_1) \wedge \text{do}(a_2)) \vee (\text{do}(a_3) \wedge \text{do}(a_4)) &\equiv \text{perform}(p_1) \\ \text{do}(a_1) \wedge \text{do}(a_3) \wedge \text{do}(a_5) &\equiv \text{perform}(p_2) \\ \text{do}(a_2) \wedge \text{do}(a_5) \wedge \text{do}(a_6) &\equiv \text{perform}(p_3) \end{aligned}$$

A little calculation reveals that the original library  $L$  has 360 different Herbrand models, but only 64 of these survive this first minimization. Each step in the construction of  $(L) \models$  can be thought of as filtering out some models which represent "non-optimal" interpretations of possible observations.

### 3.3.2 Step 2 : All actions are purposeful

Next, we add the assumption that actions only occur when they appear in some plan. This corresponds to minimizing the predicate `do`. Note that `perform` is a parameter at this step. If we tried to minimize `do` and `perform` simultaneously, the resulting models would contain no true instance of any action or plan.

The current set of models is  $\min(\min(W(L), \text{perform}), \text{do})$ . Many of the implications needed to perform plan recognition are valid in this set of models. In particular, the following statements from our example problem hold:

$$\begin{aligned} \text{do}(a_6) &\supset \text{perform}(p_3) \\ \text{do}(a_1) &\supset \text{perform}(p_1) \vee \text{perform}(p_2) \\ \text{do}(a_3) &\supset \text{perform}(p_1) \vee \text{perform}(p_2) \\ \text{do}(a_1) \wedge \text{do}(a_3) &\supset \text{perform}(p_2) \end{aligned}$$

There are only 10 distinct Herbrand models in  $\min(\min(W(L), \text{perform}), \text{do})$ . These are all illustrated in table 3 below, and the reader can verify that the formulas above hold in all of them. But this set of models must be further constrained.

<u>model</u>		<u>do holds of</u>				<u>perform holds of</u>				
m1	<->									
m2	<->	a1	a2					p1		
m3	<->			a3	a4			p1		
m4	<->	a1		a3		a5			p2	
m5	<->		a2			a5	a6			p3
m6	<->	a1	a2	a3		a5		p1	p2	
m7	<->	a1		a3	a4	a5		p1	p2	
m8	<->	a1	a2			a5	a6	p1		p3
m9	<->		a2	a3	a4	a5	a6	p1		p3
m10	<->	a1	a2	a3		a5	a6	p1	p2	p3

Table 3:  
Models in  $\min(\min(W(L), \text{perform}), \text{do})$

### 3.3.3 : Simplicity of intention

So far, the knowledge base has been manipulated before any observations were made. The assumption that the agent is performing only a few different plans is implemented by forcing the extension of `perform` to be as small as possible, *after* the observations are added. That we must wait is obvious from that fact that before any observations are made, it is *consistent* that the agent is not performing *any* plan.

Suppose we observe `a2` and `a5`. Adding this information to our knowledge base corresponds to selecting those models from figure 1 in which  $\text{do}(a_2) \wedge \text{do}(a_5)$  holds. This set of five models is pictured in table 4.

<u>model</u>		<u>do holds of</u>				<u>perform holds of</u>			
m5	<->		a2			a5	a6		p3
m6	<->	a1	a2	a3		a5		p1	p2
m8	<->	a1	a2			a5	a6	p1	p3
m9	<->		a2	a3	a4	a5	a6	p1	p3
m10	<->	a1	a2	a3		a5	a6	p1	p2

**Table 4:**  
Models in

$$W(\text{do}(a2) \wedge \text{do}(a5)) \cap \min(\min(W(L), \text{perform}), \text{do})$$

The most *concise* explanation for the occurrence of this pair of actions is  $p_3$ . However,  $p_3$  is not valid in the set of models above; in particular, it does not hold in  $m_6$ . Instead, only this weaker formula holds:

$$(\text{do}(p1) \wedge \text{do}(p2)) \vee \text{do}(p3)$$

The kind of minimization used before (which, it will turn out, is simply circumscription), can't be used to obtain the desired result. Therefore we define the following, much stronger, form of minimization.

$m_1 \leq_{[p]} m_2$  holds just in case the size of the extension of  $p$  in  $m_1$  is no greater than it is in  $m_2$ .

$\text{mincard}(M, p)$  is the subset of  $M$  whose members are minimal in  $[p]$ ; that is,

$$\{m \mid m \in M \wedge \neg \exists m' . m' \in M \wedge m' \leq_{[p]} m\}$$

Note that in the ordering relation above, the models  $m_1$  and  $m_2$  do not need to agree on all predicates other than  $p$ , as was the case before. That is to say that all other predicates are allowed to vary, instead of being parameters.

Now we can put the pieces together. First, minimize *perform* and then *do* in the plan library. Next, add the observations. Finally, select those models which minimize the number of distinct plans. Call this final set of models  $\mu$ . Formally:

$$\mu = \text{mincard}(W(B) \cap \min(\min(W(L), \text{perform}), \text{do}), \text{perform})$$

where  $B = \text{observations } \text{do}(a_j), \dots, \text{do}(a_k)$

See figure 7. In the example problem  $\mu$  contains only the single model  $m_5$ . Thus we obtain the desired result that

$$\mu \models \text{do}(p3)$$

The plan recognizer may, of course, be mistaken. The two actions  $a_2$  and  $a_5$  may actually belong to the unrelated plans  $p_1$  and  $p_2$ . In that case one must expand  $B$  with



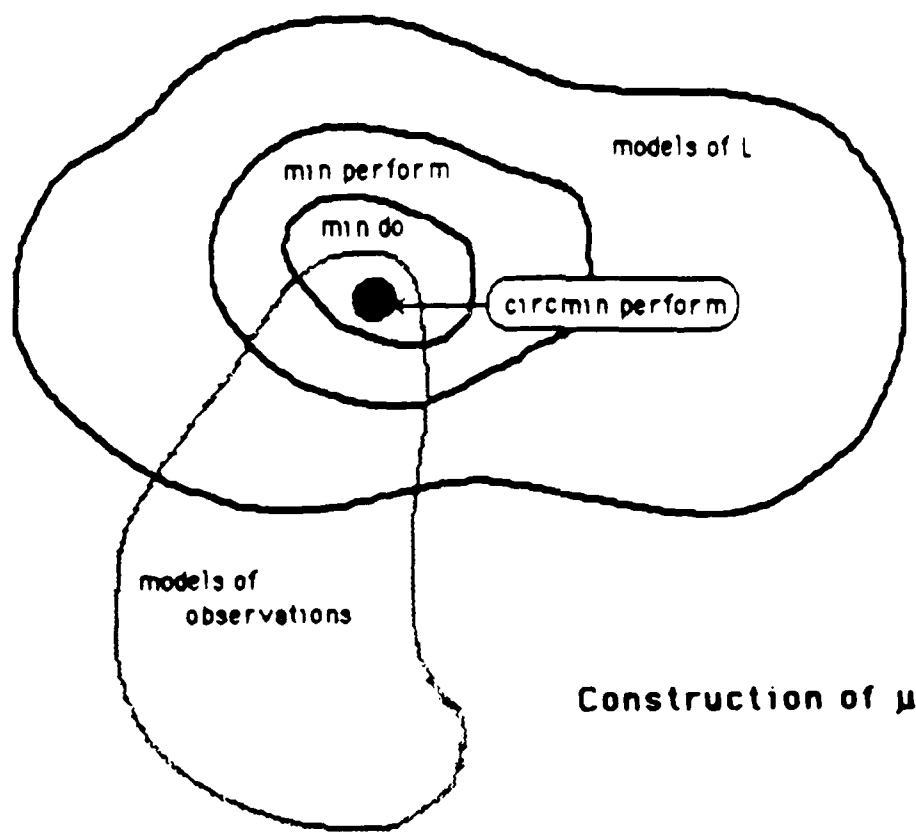


Figure 7

this additional information and recompute  $\mu$ . Such are the dangers of non-monotonic inference. The semantic relation between observations and recognized plans is simply

$$B(L) \models F \text{ if and only if } \mu \models F \\ \text{where } \mu \text{ is defined as above}$$

### 3.4 Proof Theory

Circumscription is the proof-theoretic counterpart of the min operator [Davis 80]. Minimizing the *cardinality* of the extension of a predicate, as does the mincard operator, is not the same. One can define, however, a formula of second-order logic which precisely captures the latter sort of minimization. At least in the simple examples examined so far, only a small number of first-order instantiations of the second-order formulas involved are actually required to obtain the desired results. The plan recognition process can thus be given a proof-theoretic, as well as semantic, definition.

Let  $\psi(p,R)$  be a formula of first-order logic in which the unary predicate  $p$ , and the set of predicates  $R$ , appear. Define:

$$\text{circcard}(\psi(p,R), p;R) = \text{df} \\ \psi(p, R) \wedge \forall p', R', f. [ \psi(p', R') \wedge \forall x. p'(x) \supset \exists y. x=f(y) \wedge p(y) ] \supset \\ [ \forall x,y. p(x) \wedge p(y) \wedge x \neq y \supset f(x) \neq f(y) ]$$

The variable  $p'$  ranges over predicates, and  $R$  is a list of variables ranging over predicates. The variable  $f$  ranges over unary functions. This formula can be understood as asserting that any predicate  $p'$  which satisfies all the conditions imposed by  $\psi$  on  $p$  has an extension which is no smaller than that of  $p$ . That is, any function  $f$  from the extension of  $p$  which is *onto* the extension of  $p'$  is also 1 to 1. Quantifying over the predicates in  $R$  allows them to be set to the values which let the extension of  $p$  be as small as possible.

We would like *circcard* to correspond to the *circmin* operator. It appears to do so, in those cases where the predicate  $p$  has a finite extension. I will not give a formal proof of this correspondence in this paper. As is the case with circumscription, it appears that it is easy to show that *circcard* is correct with respect to the *circmin* semantics, but difficult to show that it is complete.

The following example illustrates the *circcard* formula. Let

$$\psi(p) = A \neq B \wedge [ (p(A) \wedge p(B)) \vee p(C) ]$$

Minimizing the cardinality of  $p$  should entail that  $p(C)$ , but circumscribing  $p$  only strengthens the disjunction to exclusive or. In proof theoretic terms,

$$\text{circcard}(\psi(p), p) \vdash p(C) \\ \text{circ}(\psi(p), p) \vdash \neg p(C) \\ \text{circ}(\psi(p), p) \vdash (p(A) \wedge p(B)) \oplus p(C)$$

To see this, substitute  $x=C$  for the predicate variable  $p$ , and the constant function  $C$  for the function variable  $f$  in the circard formula. This yields:

$$A \neq B \wedge [ (p(A) \wedge p(B)) \vee p(C) ] \wedge \\ \{ [ A \neq B \wedge [ (A=C \wedge B=C) \vee C=C ] \wedge \\ \forall x . x=C \supset \exists y . x=C \wedge p(y) ] \supset \\ [ \forall x,y . p(x) \wedge p(y) \wedge x \neq y \supset C \neq C ] \}$$

This formula simplifies to:

$$A \neq B \wedge [ (p(A) \wedge p(B)) \vee p(C) ] \wedge \\ [ \forall x,y . p(x) \wedge p(y) \wedge x \neq y \supset C \neq C ]$$

or equivalently:

$$A \neq B \wedge [ (p(A) \wedge p(B)) \vee p(C) ] \wedge \\ [ \forall x,y . p(x) \wedge p(y) \supset x=y ]$$

The second line above says that there is only one  $p$  thing. Therefore, since  $A \neq B$ , the formula implies that  $p(A)$ .

The proof-theoretic plan recognition operator, which corresponds to the  $(L) \models$  construction above, is defined by simply combining the appropriate circumscription and circard operators:

$$\text{circard}( B \wedge \text{circ}( \text{circ}(L, \text{perform}), \text{do}), \text{perform}, R )$$

where  $B$  is a formula representing the observations, and  $R = \{ \text{do}, \dots \}$ , the set of all predicates other than  $\text{perform}$  or equality in  $L$ .

Making useful inferences from such a complex formula is extremely difficult: practical systems would probably have to rely on the less formal recognition algorithms sketched below.

### 3.5 Plan Recognition Algorithms

How can a plan recognition algorithm which respects these semantics be implemented? In the simplest case, where there are no constraint expressions, and action and plans are represented by constants, plan recognition is equivalent to a version of the minimal cover problem. Consider each plan body to be a set of actions. An *acceptable cover* of a set of observations is a set of plan bodies which include all the observations, and includes no two bodies of a single plan. The observer should conclude the disjunction of all acceptable minimal covers of the observations. The medical diagnosis system of [Reggia 83] performs precisely this calculation. An unpublished paper by Reggia presents, in great detail, algorithms for solving such minimal cover problems. No universally efficient algorithm is likely to be found, since the minimal cover problem is NP-complete.

In practice, however, one may use algorithms which perform efficiently when the size of the minimal cover sets is small.

To help lay the groundwork for the discussion of plan recognition in more complicated cases, consider the following very simple "filtering" algorithm. This algorithm works properly when each plan has a single expansion, and all observations can be explained by a single plan, and returns *FAIL* when they cannot. (The first limitation is easy to lift, but it is considerably more complicated to handle cases where more than one plan must be postulated.)

#### Propositional Plan Recognition Algorithm

1. Initialize  $P$  to the set of known plans,  $\{p_1, \dots, p_n\}$ .
2. If  $P = \{p_j\}$ , then conclude  $\text{perform}(p_j)$  and halt.
3. If there are no more observations, conclude the disjunction of the elements of  $P$ ,  $\text{perform}(p_1) \vee \dots \vee \text{perform}(p_j)$ .
4. Obtain an observation  $\text{do}(a_{\text{obs}})$ .
5. Remove from  $P$  any plan whose body does not contain  $\text{do}(a_{\text{obs}})$ .
6. Go to step 2.

The key step is #5, where each observation is integrated into the partially recognized plan, and so maintains the minimality of  $\text{perform}$ . In the more general case, this step involves performing consistency unification between the observation and each candidate plan.

### 3.6 Parameterized plans and actions

As discussed above, plans and actions are usually represented by functions, whose arguments include the agent performing the act, the specific objects acted upon, the time at which the act occurred, and so forth. (It is also possible to relate these arguments to an action instance by the use of a role predicate, as in [Allen & Frisch 82], but this does not affect the present discussion.) Time intervals can be related by predicates such as *meets*, *if one immediately follows the other*, or *before*, or *during* (with the obvious interpretations), etc. See [Allen 84] for details. For example, an axiom for the plan stack could be:

$$\begin{aligned}
& \forall x y t_1 t_2 t_3 t_4 t_5 . \\
& \quad \text{do}(\text{goto}(x, t_1)) \wedge \\
& \quad \text{do}(\text{grasp}(t_2)) \wedge \\
& \quad \text{do}(\text{goto}(y, t_3)) \wedge \\
& \quad \text{do}(\text{release}(t_4)) \wedge \\
& \quad \{ \text{box}(x) \wedge \\
& \quad \quad \text{clear}(x, t_2) \wedge \\
& \quad \quad \text{clear}(y, t_4) \wedge \\
& \quad \quad \text{meets}(t_1, t_2) \wedge \text{meets}(t_2, t_3) \wedge \text{meets}(t_3, t_4) \wedge \\
& \quad \quad t_5 = \text{join}(t_1, t_2, t_3, t_4) \} \\
& \quad \supset \text{perform}(\text{stack}(x, y, t_5))
\end{aligned}$$

The formula asserts that one may stack  $x$  on  $y$  by going to  $x$ , grabbing it, going to  $y$ , and then dropping  $x$ . Box  $x$  must be clear in order to be grasped, and  $y$  must be clear in order to have something put on it. The entire plan is performed over the concatenation (the join) of the time of each action. The conjunction of *do*-predications is the body of the plan. The expression in curly braces constrains the plan.

How should the simple plan recognition algorithm be extended? Instead of simply locating observations within candidate plans, the algorithm must merge observations with these plans. Before this step is performed, all the parameters in the candidate plans have been bound to various skolem functions, or to terms which appeared in previous observations. Thus the merging process may require that some terms which appear in the observation statements must be set equal to terms which appear in the candidate plans. Standard unification involves merging two expressions by finding particular instantiations for their variables. In the final step of the semantic construction for plan recognition, all the functions and constants are allowed to vary. Each acts like an existentially-bound variable, which must take a value such that a model for the observations and library exists which is minimal in the number of performed plans. Because terms are allowed to vary, but there must exist at least one consistent set of binding for all terms, we call the merging process consistency unification.

Below is the revised plan recognition algorithm. Note that there may be several distinct ways to merge an action with a candidate plan.

### First-Order Plan Recognition Algorithm

1. Let  $\kappa$  be the knowledge base. Let  $\beta$  be a copy of the plan-body axioms of the plan library  $L$ , but with each universally quantified variable in  $L$  replaced by a new skolem function (see comments below). Initialize  $P$  to the set of triples,

$$\{ \langle p_j, \{a_1, \dots, a_n\}, x_j \rangle \mid \text{do}(a_1) \wedge \dots \wedge \text{do}(a_n) \wedge x_j \supset \text{perform}(p_j) \in \beta \}$$

2. If  $P = \{ \langle p, \{a_1, \dots, a_n\}, x \rangle \}$ , then conclude  $x \wedge \text{perform}(p)$  and halt.

3. If there are no more observations, conclude the disjunction of the elements of  $P$ ,  
 $[x_i \wedge \text{perform}(p_i)] \vee \dots \vee [x_j \wedge \text{perform}(p_j)]$ .

4. Obtain an observation  $\text{do}(a_{\text{obs}})$ , where any existentially-quantified variables in  $a_{\text{obs}}$  are replaced by skolem constants.

5. For each  $\langle p, \{a_1, \dots, a_n\}, x \rangle \in P$  do:

Remove  $\langle p, \{a_1, \dots, a_n\}, x \rangle$  from  $P$ ;

For each  $a_i \in \{a_1, \dots, a_n\}$  do:

Let  $\varphi$  be the weakest set of equality assertions such

that  $\xi \wedge \varphi \vdash a_i = a_{\text{obs}}$

If  $\kappa \cup \{ \text{do}(a_1) \wedge \dots \wedge \text{do}(a_n) \} \cup \{ \xi \wedge \varphi \}$  is consistent, then

add  $\langle p, \{a_1, \dots, a_n\}, \xi \wedge \varphi \rangle$  to  $P$

6. Go to step 2.

The reader may be wondering why *universally* quantified variables are skolemized in step 1. The algorithm attempts to establish that there exists *some* instance of some known plan. While the plan-body axiom is of the form

$$\forall x . [ \text{do}(a_1(x)) \wedge \dots \wedge \text{do}(a_n(x)) \wedge \xi \supset \text{perform}(p(x)) ]$$

the formula to be established is of the form:

$$\exists x . [ \text{do}(a_1(x)) \wedge \dots \wedge \text{do}(a_n(x)) \wedge \xi \wedge \text{perform}(p(x)) ]$$

Universally-quantified variables in the axiom thus correspond to existentially quantified variables in the formula to be established. Therefore we are in fact skolemizing existentially-quantified variables, as is usually the case.

The algorithm may be performed incrementally, where the system is allowed to perform other actions between steps 3 and 4; alternatively, all the observations may be gathered beforehand. There is no implicit temporal relation between succeeding observations; all temporal relations are explicitly encoded in time indexes.

The following example illustrates the algorithm. Let the plan library contain the stack plan above, and some others not containing the goto action: say,

$$\forall t_1 t_2 t_3 .$$

$$\begin{aligned} & \text{do}(\text{grasp}(t_1)) \wedge \\ & \text{do}(\text{release}(t_2)) \wedge \\ & \{ \text{meets}(t_1, t_2) \wedge \\ & \quad t_3 = \text{join}(t_1, t_2) \} \\ & \supset \text{perform}(\text{clap}(t_3)) \end{aligned}$$

(The robot arm on display at the Toronto Science Center appears to perform exactly these two plans, over and over.) The system observes a goto motion followed by grasp. A (hand-run) trace of the algorithm follows:

(step 1) Set  $P = \{$   
 $\langle \text{stack}(C,D,S), \{ \text{goto}(C,S1), \text{grasp}(S2), \text{goto}(D,S3), \text{release}(S4) \},$   
 $\text{box}(C) \wedge \text{clear}(C, S2) \wedge \text{clear}(y, S4) \wedge \text{meets}(S1, S2) \wedge$   
 $\text{meets}(S2, S3) \wedge \text{meets}(S3, S4) \wedge S = \text{join}(S1, S2, S3, S4) \rangle,$   
 $\langle \text{clap}(R), \{ \text{grasp}(R1), \text{release}(R2) \},$   
 $\text{meets}(R1, R2) \wedge R = \text{join}(R1, R2) \rangle \}$

(step 2)  $P$  is not unary, so continue.

(step 3) There are more observations, so continue.

(step 4) Obtain the next observation,  $\text{do}(\text{goto}(\text{OBJ99}, T34))$ . Note that  $T34$  is a constant created to "timestamp" the observation.

(step 5) Update  $P$ . The  $\text{clap}$  plan is disqualified, and there are two consistent ways to merge  $\text{do}(\text{goto}(\text{OBJ99}))$  with  $\text{stack}$ .  $\bar{P}$  is now (with changes in italics)

$$\{ \langle \text{stack}(C,D,S), \{ \text{goto}(C,S1), \text{grasp}(S2), \text{goto}(D,S3), \text{release}(S4) \},$$
 $\text{box}(C) \wedge \text{clear}(C, S2) \wedge \text{clear}(y, S4) \wedge \text{meets}(S1, S2) \wedge$ 
 $\text{meets}(S2, S3) \wedge \text{meets}(S3, S4) \wedge S = \text{join}(S1, S2, S3, S4)$ 
 $\wedge \text{OBJ99} = C \wedge S1 = T34 \rangle,$ 
 $\langle \text{stack}(C,D,S), \{ \text{goto}(C,S1), \text{grasp}(S2), \text{goto}(D,S3), \text{release}(S4) \},$ 
 $\text{box}(C) \wedge \text{clear}(C, S2) \wedge \text{clear}(y, S4) \wedge \text{meets}(S1, S2) \wedge$ 
 $\text{meets}(S2, S3) \wedge \text{meets}(S3, S4) \wedge S = \text{join}(S1, S2, S3, S4)$ 
 $\wedge \text{OBJ99} = D \wedge S3 = T34 \rangle \}$ 

(step 2) Continue as before.

(step 3) At this point, the system can conclude the disjunction of two plans: either the agent is going to stack OBJ99 on something, or stack something on OBJ99.

(step 4) Obtain the next observation,  $\text{do}(\text{grasp}(T40))$ , where the system knows that  $\text{before}(T34, T40)$ .

(step 5) The action  $\text{do}(\text{grasp}(T40))$  can merge with the first candidate in P. The second candidate is eliminated, because

$$\text{meets}(S2, S3) \wedge S3 = T34 \wedge S2 = T40$$

from the constraint expression, together with  $\text{before}(T34, T40)$

from the knowledge base, is inconsistent. This is because together they would imply

$$\text{meets}(S2, S3) \wedge \text{before}(S3, S2)$$

(step 2) The system concludes

$$\text{perform}(\text{stack}(C, D, S)) \wedge$$

$$\text{do}(\text{goto}(C, S1)) \wedge \text{do}(\text{grasp}(S2)) \wedge$$

$$\text{do}(\text{goto}(D, S3)) \wedge \text{do}(\text{release}(S4)) \wedge$$

$$\text{box}(C) \wedge \text{clear}(C, S2) \wedge \text{clear}(y, S4) \wedge \text{meets}(S1, S2) \wedge$$

$$\text{meets}(S2, S3) \wedge \text{meets}(S3, S4) \wedge S = \text{join}(S1, S2, S3, S4)$$

$$\wedge \text{OBJ99} = C \wedge S1 = T34 \wedge S2 = T40$$

or, more simply, that

$$\text{perform}(\text{stack}(\text{OBJ99}, D, S))$$

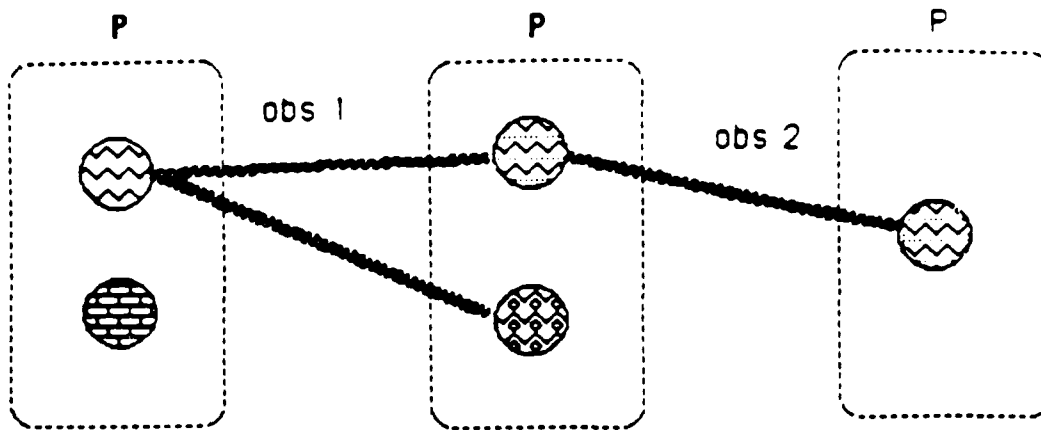
where the full extent of the time interval S, and the destination object D, are not yet determined.

Figure 8 illustrates the splitting and filtering of the candidate set P that the algorithm performs. Note that the system should reach this conclusion even if the single-plan limitation of the algorithm were lifted, since it is the most concise account of the observations.

For the limited case described, where the actor cannot be performing more than one plan, the algorithm appears to be correct, according to the semantic treatment of plan recognition discussed earlier. Because of the initial minimizations of do and perform, each observed action entails the existence of a corresponding containing plan. Maintaining the minimality of the extension of perform requires that each succeeding observed act be equal to some step in a previously hypothesized plan.

It is not possible to implement a plan recognition algorithm which is complete in the first-order case. For example, if arbitrary constraint expressions are allowed, then testing the consistency of the potential plan could be undecidable. In practice, however, most cases of inconsistent merges can be blocked. Many can be detected by using a system of types; see, for example, [Allen & Frisch 82]. Constraint propagation systems, such as RUP [McAllester 80], can also be used to approximately solve the problem of inconsistency detection. Those remaining cases where a plan recognition system may not realize that a potential interpretation is inconsistent may prove difficult and confusing for humans as well.





### Splitting and Filtering

The set of candidate plans,  $P$ , initially has 2 members STACK and CLAP. After the first observation, CLAP is filtered out, and STACK is split into two candidates. After the second observation, only one version of STACK remains.

Figure 8

## Chapter 4 Future Directions

### 4.1 Example Problems

The theory presented in the last chapter only handles the simplest cases of plan recognition. There are two more less independent ways the theory can be developed further.

One is to integrate plan recognition with multi-agent planning. For example, one would want to represent situations in which an observer must plan to do something in order to determine another agent's plan, or in which an agent plans to do something in order that his plan be recognized. Discourse analysis, as discussed in chapter 2, includes many problems of this sort.

A second way to develop the theory is to apply it to cases where the plans to be recognized have greater internal structure. We need to consider hierarchical plans, where a step of one plan may itself be a plan, rather than a primitive action; plans with loops and tests; and possibly plans which modify other plans, such as Litman's or Wilensky's meta-plans.

For the immediate future we intend to concentrate on the second set of problems. We will not immediately try to base our work on real data (as did [Schmidt 78]). Instead, the following series of constructed examples should illustrate the intended scope of the work. If their subject matter seems contrived, the reader should feel free to restate them using a more realistic domain, such as the Unix (tm) operating system, or strategic warfare. Please note our logic is aimed at simulating the human's reasoning in these examples, not the robot's!

#### The Scenario

The postman has just delivered a box containing Robbie IX (partly assembled), the latest product from Heuristic Automaton Limited. You snap Robbie together and press the ON button. Robbie begins moving about the house, and quickly discovers a set of LEGO (tm) blocks, together with a booklet entitled "LEGO Fun for Everyone!" Robbie scans the booklet, then begins playing with the blocks. You want to ask Robbie what he's building, but unfortunately, you neglected to order Robbie's optional Natural Language Module. However, by comparing Robbie's actions with the step-by-step plans illustrated in the booklet, you are able to make a good guess about what he's up to. . . .

#### 4.1.1 Example 1: Multiple Observations

Robbie takes a set of wheels. He's building some kind of vehicle, you muther. Let's see, there are plans here for a racing car, a firetruck, and a lunar explorer. All of those require *two* sets of wheels, so he's going to have to find another set. He's got them, and now he's getting some red blocks. He must be building a fire truck.

#### 4.1.2 Example 2: Plan Hierarchies

The booklet has two larger pictures of the firetruck: one inside a fire house, and another, on a city street. Robbie must be building one of these "city" models, rather than the "lunar landscape" or "racetrack" scenes.

#### 4.1.3 Example 3: Tests and Conditional Actions

Robbie tries to spin one of the wheels. It seems stuck; Robbie pulls it off. The instructions say that "wheels should turn freely on their axle; lubricate if necessary." Robbie must have been checking that. He's wandered off now; probably searching for an oil can, you think.

#### 4.1.4 Example 4: Loops

Robbie returns with the 3-In-One, oils the wheels, and finishes the truck. Now he'll begin something new, you think. Robbie picks up a large flat base, to which a number of of blocks are attached. He pulls off one block, drops it, pulls off another, drops it ... Robbie must be *clearing off the base, by removing all pieces* attached to it.

#### 4.1.5 Example 5: Concurrent Plans

Robbie picks up a rod-shaped piece. It must be the firehouse: that base is the floor, and the rod is part of the fire pole. Ooops -- he's sticking a green ball onto the rod, making a tree. It's the street scene after all, and that base must be for something else -- a building, maybe, or a billboard.

#### 4.1.6 Example 6: Higher-Order Plans

By now the LEGO village occupies most of the house. Only the court house and the jail remain to be built. But both, according to the manual, are built from *red* blocks, and there aren't nearly enough pieces of that color left. Robbie hesitates. For a moment you fear that he's about to disassemble the firetruck to get the pieces for the buildings, then disassemble the buildings to rebuild the firetruck, forever and anon. Instead, Robbie begins building something that looks like a court house out of *white* bricks. Very good, you conclude, Robbie has *modified* the court house plan by *substituting white for red*.

## 4.2 Hierarchical Plans

While the minimal model construction may not suffice to handle all the examples above, it can be easily extended to deal with a *static* plan hierarchy.

Let the predicate **do** be applicable to plans or actions which occur. A **basic** plan is one which cannot be decomposed. An **ultimate** plan is a highest-level, domain-specific plan, which need not be explained by its appearance in a more complex plan. Define:

$$\text{dobasic}(x) \equiv [ \text{do}(x) \wedge \text{basic}(x) ]$$

$$\text{doulimate}(x) \equiv [ \text{do}(x) \wedge \text{ultimate}(x) ]$$

The plan library is as before, but with **do** used in place of **perform**, and some plans marked as **basic** or **ultimate**. The plan recognition construction is as follows:

1. Minimize (circumscribe) **basic** and **ultimate**. This ensures that plans not explicitly marked with either predicate are known to be "intermediate level" plans.
2. Minimize **do**, where the predicate **doulimate** is allowed to vary. The predicate **dobasic** acts as a parameter. This means that intermediate and ultimate-level plans *only* are minimized.
3. Minimize **do**, where the predicate **dobasic** is allowed to vary. The predicate **doulimate** acts as a parameter. This means that basic and intermediate-level plans *only* are minimized.
4. Add the observations.
5. Now a choice arises. Minimizing the cardinality (circcard) of **doulimate** corresponds, as before, to making the assumption that as few different highest-level plans as possible are being performed. Minimizing the cardinality of **dobasic** corresponds to the assumption that the agent will perform as few basic actions as possible; this invokes a principle of "least effort" by the agent.

The following example illustrates the new aspects of this construction. Let the plan library be.

$$\text{do}(a) \wedge \text{do}(b) \supset \text{do}(d)$$

$$\text{do}(d) \wedge \text{do}(e) \supset \text{do}(f)$$

$$\text{do}(c) \supset \text{do}(e)$$

$$\text{basic}(a), \text{basic}(b), \text{basic}(c)$$

$$\text{ultimate}(f)$$

Plans **d** and **e** are intermediate-level. Step 1 adds the constraint that:

$$\text{basic}(x) \supset [ x=a \vee x=b \vee x=c ]$$

$$\text{ultimate}(x) \supset x=f$$

Step 2 strengthens the implications in the plan library to equivalences. The process works for any number of intermediate levels in the library. In intuitive terms, a plan at level *n* in

the hierarchy can only hold if it must because some set of basic plans hold; but the basic plans force the level  $n$  plan to hold by forcing its level  $n-1$  substeps to hold. The result in the example is:

$$\begin{aligned} \text{do}(a) \wedge \text{do}(b) &\equiv \text{do}(d) & \text{do}(d) \wedge \text{do}(e) &\equiv \text{do}(f) \\ \text{do}(c) &\equiv \text{do}(e) \end{aligned}$$

Step 3 means that if any plan occurs, then it must be because some ultimate-level plan which contains it occurs. Since the example here has only a single ultimate-level plan, every plan is constrained to entail it:

$$\begin{aligned} \text{do}(a) &\supset \text{do}(d) & \text{do}(d) &\supset \text{do}(f) \\ \text{do}(b) &\supset \text{do}(d) & \text{do}(e) &\supset \text{do}(f) \\ \text{do}(c) &\supset \text{do}(e) \end{aligned}$$

The final two steps are, of course, trivial in this example (everything is evidence for  $f$ ). Appropriate semantic and proof-theoretic descriptions of the entire construction, can, as in the previous chapter, be pieced together. The model set  $\mu$  becomes:

$$\begin{aligned} \text{mincard}(W(\mathbf{B})) \cap & \\ \text{min}( & \\ \text{min}( & \\ \text{min}(\text{min}(W(\mathbf{L}), \text{basic}), \text{ultimate}), & \\ \text{do}\{\text{doulultimate}\}), & \\ \text{do}\{\text{dobasic}\}), & \\ \text{doulultimate}) & \end{aligned}$$

where  $\mathbf{B}$  = observations

As discussed above, the final **doulultimate** could be replaced by **dobasic**, in order to make different assumptions for the final minimization. The proof-theoretic operator is defined analogously.

## 4.5 Conclusions

This paper has discussed the importance of plan recognition for work in artificial intelligence, and has begun to sketch a formal basis for non-quantitative methods of recognition. We have shown that plan recognition is an important form of non-monotonic reasoning, and can be used to test various methods of non-monotonic logic, such as circumscription or default logic. Future work will extend both the model theory and corresponding algorithms to handle more complicated plans.

## References

- Allen, James (1983) Recognizing intentions from natural language utterances. *Computational Models of Discourse*, Michael Brady & Robert Berwick eds., The MIT Press, Cambridge.
- Allen, J.F. & C.R. Perrault (1980) Analyzing intention in dialogues, *Artificial Intelligence* 15, no. 3, pp. 143-178.
- Allen, James & Frisch, Alan (1982) What's in a semantic network?, *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, Toronto.
- Allen, James & Koomen, Johannes (1983) Planning Using a Temporal World Model. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany.
- Blenko, Tom (1985) Report on Commonsense Summer: Suggestions and Offers. To be published.
- Bruce, B.C. (1981) Plans and social action, *Theoretical Issues in Reading Comprehension*, R. Spiro, B. Bruce, & W. Brewer, eds., Lawrence Erlbaum, Hillsdale, New Jersey.
- Carberry, Sandra (1983) Tracking Goals in an Information Seeking Environment. *Proceedings of the National Conference on Artificial Intelligence. AAAI-83*, Washington, D.C.
- Carbonell, Jaime (1979) The Counterplanning Process: A Model of Decision-Making in Adverse Situations, technical report (unnumbered), Computer Science Department, Carnegie-Mellon University, Feb.
- Cohen, P.R. (1978) On knowing what to say: planning speech acts, TR 118, Department of Computer Science, University of Toronto.
- Cohen, Phillip (1984) Referring as Requesting, *Proceedings of COLING 84*, Stanford University, California, pg 207.
- Cohen, P. & Levesque, H. (1980) Speech Acts and the Recognition of Shared Plans, *Proceedings of the Third Biennial Conference, CSCSI*, Victoria, B.C., pg 263-271.
- Cohen, P., C. Perrault, & J. Allen (1981) Beyond Question-Answering, Report No. 4644, BBN Inc., Cambridge, MA.
- Davis, Martin (1980) The Mathematics of Non-Monotonic Reasoning, *Artificial Intelligence* 13, pp. 73-80.
- Dwyer, M.G. (1982) In depth understanding: A computer model of integrated processing for narrative comprehension, TR 219, Department of Computer Science, Yale University.

- Genesereth, Michael (1979) The Role of Plans in Automated Consulting. *Proceedings of IJCAI-79*.
- Goldman, Alvin (1970) *A Theory of Human Action*. Princeton University Press, Princeton.
- Goudge, T.A. (1969) *The Thought of C.S. Peirce*, Dover.
- Green, C. (1969) Application of theorem proving to problem solving, *International Joint Conference on Artificial Intelligence*, pp. 219-239, Washington, D.C.
- Grosz, B.J. (1977) The Representation and Use of Focus in Dialogue Understanding, Technical Note 151, SRI International, Palo Alto.
- Harel, David (1979) *First-Order Dynamic Logic*, Springer-Verlag.
- Huff, Karen & Victor Lesser (1982) KNOWLEDGE-BASED COMMAND UNDERSTANDING: An Example for the Software Development Environment, Technical Report 82-6, Computer and Information Sciences, University of Massachusetts at Amherst, MA.
- Kunz, John C. (83) Analysis of Physiological Behavior Using a Causal Model Based on First Principles, in *Proceedings of AAAI-83*, Washington, D.C., pp. 225-228.
- Kyberg, Henry (1974) *The Logical Foundations of Statistical Inference*. D. Reidel, Dordrecht-Holand.
- Lehnert, Wendy (1980) Affect Units and Narrative Summarization, TR 179, Department of Computer Science, Yale University.
- Lewis, David (1969) *Convention*, Harvard University Press, Cambridge.
- Lifschitz, Vladimir (1984) Some Results on Circumscription, *Proceedings of the AAAI Workshop on Non-Monotonic Reasoning*.
- Litman, Diane & James Allen (1984) A Plan Recognition Model for Clarification Subdialogues, TR 141, Department of Computer Science, University of Rochester, NY.
- McAllester, David (1980) An Outlook on Truth Maintenance, AI Memo 551, Massachusetts Institute of Technology.
- McCarthy, J. & P. Hayes (1969) Some philosophical questions from the standpoint of artificial intelligence, *Machine Intelligence 4*, Edinburg University Press, Edinburg, UK.
- McCarthy, John (1980) Circumscription -- A Form of Non-Monotonic Reasoning, *Artificial Intelligence 13*, pp. 27-39.
- McCarthy, John (1984) Applications of Circumscription to Formalizing Common Sense Knowledge, *Proceedings of the AAAI Workshop on Non-Monotonic Reasoning*.
- McDermott, Drew (1981) A Temporal Logic for Reasoning About Processes and Plans, Research Report #196, Department of Computer Science, Yale University, March.

- Minker, J. & Perlis, D. (1985) *Circumscription: Finitary Completeness Results*. Computer Science Department, University of Maryland.
- Nilsson, N.J. (1980) *Principles of Artificial Intelligence*. Tioga Press, Palo Alto.
- Perlis, D. (1980) *Truth, Syntax, and Reason*. PhD Thesis, Computer Science Department, University of Rochester.
- Pople, Harry (1973) *On the Mechanization of Abductive Logic*, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, CA.
- Pople, Harry (1977) The formation of compound hypotheses in diagnostic problem solving: an exercise in synthetic reasoning, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, M.I.T., Cambridge, MA.
- Reggia, J., D.S. Nau, & P.Y. Wang (1983) Diagnostic expert systems based on a set covering model, *Int. J. Man-Machine Studies* 19, pp. 437-460.
- Reichman, R. (1981) *Plan Speaking: A Theory and Grammar of Spontaneous Discourse*. Report 4681, BBN Incorporated, Cambridge.
- Reiter, R. (1980) A Logic for Default Reasoning, *Artificial Intelligence*, vol. 13, no. 2, April.
- Reiter, Ray (1982) Circumscription Implies Predicate Completion (Sometimes), *Proceedings of the National Conference on Artificial Intelligence, AAAI-82*, William Kaufman, Inc.
- Rich, Charles (1981) *Inspection Methods in Programming*, AI-TR-604, Laboratory for Artificial Intelligence, M.I.T., June.
- Robinson, G. & L. Wos (1969) Paramodulation and Theorem-Proving in First-Order Theories with Equality, *Machine Intelligence* 4, pp. 135-150, Edinburg University Press, Edinburg, UK.
- Sacerdoti, Earl (1977) *A Structure for Plans and Behavior*, Elsevier, New York.
- Schank, R. (1975) *Conceptual Information Processing*, American Elsevier, New York.
- Schmidt, C.F., N.S. Sridharan, & J.L. Goodson (1978) The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence, *Artificial Intelligence* 11, pp. 45-83.
- Sidner, Candace & David Israel (1981) Recognizing Intended Meaning and Speakers' Plans, *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 203-298, University of British Columbia, Vancouver, BC.
- Sidner, Candace (1983) Focusing in the Comprehension of Definite Anaphora, in *Computational Models of Discourse*. M. Brady and R. Berwick, editors. M.I.T. Press, Cambridge



Smith, Reid G. & Randall Davis (1981) A Framework for Cooperation in Distributed Problem Solving, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11, no. 1, Jan.

Stefik, Mark (1980) *Planning with Constraints*. Report STAN-CS-80-784, Department of Computer Science, Stanford University.

Wilensky, Robert (1982) Talking to UNIX in English: An Overview of U.C., *Proceedings of AAAI-82*, Carnegie-Mellon University, Pittsburgh.

Wilensky, Robert (1983) *Planning and Understanding*, Addison-Wesley, Reading, MA.

# A FORMAL LOGIC THAT SUPPORTS PLANNING WITH A PARTIAL DESCRIPTION OF THE FUTURE

Richard N. Pelavin  
Department of Computer Science  
University of Rochester  
Rochester, N.Y. 14627

## Abstract

This paper presents a formal logic that can be used to reason about planning to achieve one's goals. A robust temporal logic (Allen's interval logic) will be extended with a counterfactual-like modality, called IFTRIED, that can be used to describe what can and cannot be done by the planning agent, the robot. This allows us to represent planning problems where the robot has a partial description of the past, present, and expected future, and its goal is to bring about a set of desired future conditions. We will show how the IFTRIED modality can be used to represent knowledge about what can and cannot be done. We will also briefly discuss how the logic can be used to reason about the interaction of two concurrent actions and will specify some conditions under which two actions can be executed together.

## Introduction

This paper presents a formal logic that can be used to reason about planning to achieve one's goals. As an example, a typical goal might be to get to the corner store without getting wet. One must be able to reason that this could be achieved by walking to the store and taking an umbrella if it is going to rain. In general, we are interested in robot planning problems where the robot has a partial description of the past, present, and expected future, and its goal is to bring about a set of desired future conditions. The robot must construct a plan, which consists of actions that it can cause, that it believes can be executed and if executed would make the goal hold. Plans with concurrent actions will be considered.

Given a logical statement representing the robot's current goal (i.e. desired future conditions), planning to achieve this goal can be thought of as finding a constructive proof of the following form:

Given the sentences representing the robot's current beliefs, prove that there exists a plan  $P$  such that  $P$  can be executed and if  $P$  is executed, the goal conditions would hold

One of the most successful approaches to representing events and their effects in A.I. has been situation calculus. In this logic an event is represented by a function that takes a situation, which is an instantaneous snapshot of the world, as an argument and returns the situation that results from applying the event to its argument. In effect, the world is viewed as a sequence of situations linked by events.

Typically, planning problems in situation calculus had the following form. Given a partial description of the current situation and a goal, which is a partial description of a desired situation, a sequence of actions must be found that when applied to the current situation yields a situation where the goal is true. This type of planning problem is very limited. Plans are treated as sequences of actions and therefore cannot contain concurrent actions. The planner's description of the world is limited to statements about the current situation, assertions cannot be made about the past or future. For example, situation calculus cannot be used to represent that it will rain in 5 minutes. The types of goals that can be handled are also limited. Only goal conditions that hold at the completion of plan execution are treated; goals that hold during execution, such as "do not get wet while going to the store", are not handled.

There have been a number of planning systems that have handled a larger class of planning problems than situation calculus. Wilkin's SIPE [6] and Vere's DEVISER [5] are two examples. SIPE can handle plans with concurrent actions. Wilkins introduced the notion of *resources* to reason about the interaction of parallel actions. A resource is defined as an object that an action uses during its execution. If two actions share the same resource, they cannot be executed in parallel.

Vere's DEVISER explicitly represents time thereby allowing the user to represent facts about future conditions along with facts about the present state. The system can represent that an event not under the control of the planning agent will occur at some specified time. One can specify that some goal condition must hold between two time points. A goal can also be the conjunction of two sub-goals that hold at two different times. Vere's system cannot, however, represent relative temporal information. For example, he did not allow goals of the form "achieve goal A sometime between 2:00 and 3:00 and achieve goal B after goal A". He also did not allow disjunctive type knowledge such as: either event A will occur starting at 5:00 or event B will occur starting at 5:00.

The planning systems just mentioned have no formal basis. They can be viewed as ad hoc extensions to the simple state-space planning paradigm that can be represented by situation calculus. An alternate approach, which will be taken here, is to develop a formal logic that adequately represents the planner's view of the world. A planning mechanism then can be constructed that is based on this logic.

Recently, Allen [1] and McDermott [3] have presented logics that explicitly specify the temporal intervals over which events occur and properties hold. Both absolute and relative temporal information can be represented. Overlapping events can be represented by asserting that their associated intervals overlap in time. A robot planner can use either of these logics to represent its knowledge about the past, present, and future. By taking a goal to be any temporal statement, one can specify conditions that must hold during any time in the future. Therefore, goal conditions that hold during plan execution can be handled

Lacking from both McDermott's and Allen's logics is a construct that can be used to represent how the robot can affect the world. Without extending either logic it is impossible to state "if condition C holds then action ACT can be executed". It is also impossible to state that a condition, such as whether or not it is raining, cannot be affected by the robot's actions. It is essential that a logic that supports planning be able to represent this type of knowledge. Consider the following example. Suppose that the robot knows that the only way it can get wet is by being outside without an umbrella while it is raining. If the robot's goal is to go to the store without getting wet, it must make sure that these three conditions are not simultaneously true. The robot must be able to deduce that it has no control over

whether or not it is raining. It cannot plan to stay dry by executing actions that prevent a raining event. A plan must be constructed that is contingent on whether or not it is raining. If it is raining during plan execution, an umbrella must be taken on the walk to the store. If it is not going to rain, it is not necessary to take an umbrella on the walk.

The logic being presented extends Allen's temporal logic with a modal operator that represents sentences describing how the robot can affect the world. Allen's logic will be briefly presented in the next section. We will then introduce a new class of objects that refer to plans at particular execution times. Following this, we will discuss what we mean by "can be executed". The modal operator expressing what the robot can and cannot do will then be presented. This modality, called IFTRIED, will express counterfactual statements having the form "if the robot tries to execute plan P during interval I, statement Q would be true". We will then give a brief overview of how one evaluates counterfactuals and briefly introduce Stalnaker's semantic theory of counterfactuals. Finally, it will be shown how IFTRIED can be used to represent sentences describing what the robot can and cannot affect.

The notational conventions used throughout this paper are as follows. LISP type notation will be used to represent logical sentences. For example, the sentence (P a) refers to the unary predicate P with argument a. The logical connectives will be represented by: *AND* for conjunction, *OR* for conjunction, *IF* for material implication, *IFF* for equivalence, *FORALL* for the universal quantifier, and *EXISTS* for the existential quantifier. All variable terms will be prefixed with a "?".

### Allen's Temporal Logic

Allen formulated interval logic in terms of a sorted first order logic. The syntax is divided into terms that denote events, time intervals, objects in the domain, and properties which are propositions that can hold or not hold over particular (time) intervals. He introduced a set of binary predicates that specify the temporal relation between intervals such as *IN*, *STARTS*, *MEETS*. For example, (MEETS i1 i2) means that the interval denoted by i1 immediately precedes the interval denoted by i2, that is, i1 is before i2 and there is no interval between i1 and i2. The *HOLDS*

predicate was introduced to specify the intervals over which a property holds. The statement (HOLDS  $p$   $i$ ) means that the property denoted by  $p$  holds over the interval denoted by  $i$ . An axiom was asserted capturing the fact that if a property holds over an interval, then that property holds over any interval contained in this interval. The OCCURS predicate was introduced to specify the intervals over which an event occurs. The statement (OCCURS  $ev$   $i$ ) means that the event denoted by  $ev$  occurs over the interval denoted by  $i$ . Allen distinguished events from processes. An event refers to some activity that results in some accomplishment, such as walking to the store, while a process is an activity not involving a culmination, such as "I am walking". The events that can be caused by an agent are called actions. For simplicity, we have assumed that the robot is the only agent and therefore all actions will refer to events that the robot can cause. The robot can affect the world by executing some set of its actions in some specified order. This collection of temporally ordered actions is called a plan. Objects that refer to plans at particular execution times will be added to the ontology.

#### Plan instances

In situation calculus, a plan is taken to be a sequence of actions to be executed in the current situation. This is clearly inadequate for our purposes since we want to allow plans with concurrent actions. Secondly, we want to consider plans with execution times starting any time in the future, not just the current situation. Thus, instead of talking about plans we will talk about *plan instances* which refer to plans at particular execution times. Similarly, an *action instance* is defined as an event that the robot can cause at a particular execution time.

Plan instances can be classified into three different categories. The first type is isomorphic to the set of action instances. Saying that one of these plan instances occurs is equivalent to saying that its associated action instance occurs. The second type of plan instance refers to inactive processes occurring over some time period, such as "staying at the same location between 3:00 and 3:10". Each plan instance of this type can be viewed as the non-occurrence of some set of action instances. For example, if the robot can only change locations by executing a "goto" action,

staying in the same location between 3:00 and 3:10 can be viewed as the non-occurrence of any goto action instance that starts between 3:00 and 3:10.

The third type of plan instance is formed by composing two plan instances together. By definition, a composite plan instance occurs iff both of its components occur. A plan instance with concurrent actions can be represented by composing two plan instances with overlapping execution times.

It is important to note that plan instances are partial descriptions of the robot's actions over some time period. An alternate approach would be to treat a plan instance as a complete specification of all the actions executed and not executed over some time period. If this approach was adopted, a plan instance could be defined as an ordered pair consisting of an interval denoting the time of execution along with a set specifying *all* the action instances to be executed during this interval. We have not adopted this approach here because of the following. First of all, this definition of plan instances is subsumed by our definition. Secondly, if a plan instance is a complete specification over some execution interval, two plan instances with overlapping execution times cannot both occur. It is useful to allow for plans instances that overlap in time. For example, two plans instances might be separately constructed to achieve two different goals. These plan instances might overlap in time. In order to achieve both goals, one must be able to deduce whether or not these two plan instances can both occur when taken together.

### Successful Executions and Execution Attempts

Reasoning about how the robot can affect the world involves finding plan instances that can be executed and determining what the world would look like if the plan instance is executed. First, we must discuss what it means to say that a plan instance can be executed. We develop this notion by making a distinction between successful executions and execution attempts. A similar distinction was made by Haas [2]. It will be assumed that the robot can attempt to execute any plan instance, but it might be the case that the plan instance does not successfully complete. We will say that a plan instance *can be executed* iff it successfully completes when attempted. Saying that a plan instance occurs is taken to mean that it successfully completes. Consider the following example. If a robot is more

than 5 feet from a wall at 3:00, it can execute the plan instance "move forward 5 feet starting at 3:00". If the robot is 3 feet from the wall at 3:00 and attempts this plan instance, it will move forward three feet and then aimlessly grind its gears for a short period. In this situation, the robot cannot execute "move forward 5 feet starting at 3:00".

For traditional reasons, the conditions under which an execution attempt leads to a successful execution will be called *preconditions*. In state-based systems, preconditions were used to specify whether or not an action can be executed in the current state. These were defined as formulas such that if they hold in a state then the action can be successfully applied yielding a new state. Our new notion of preconditions is somewhat of a misnomer because they might specify conditions that must hold while the plan instance is in progress. For example, one of the preconditions for walking over the Elmwood avenue bridge between 2:00 and 2:10 is that the bridge is open between 2:00 and 2:10.

#### Reasoning About the Effects of Plan Instances

The robot must be able to determine the effects of its actions. This involves reasoning about what statements would be true if a particular plan instance is attempted. Given a logical statement describing the goal  $G$ , the robot must find a plan instance that can be executed and if executed would make  $G$  true. Since by definition if a plan instance occurs then it must have been attempted, the following expresses that plan instance  $pi$  can be executed and if executed  $G$  would be true

S1) If the robot attempts  $pi$ , then  $pi$  occurs and  $G$  holds

In general, we are interested in reasoning about sentences having the form.

S2) If the robot tries to execute  $pi$ , then  $P$  holds  
where  $P$  is any logical statement

If S2 is treated as a material implication, it would vacuously hold if the antecedent, "the robot tries to execute  $pi$ ", is false. Thus, we shall interpret S2 as a counterfactual statement. Allen's logic will be extended with a counterfactual



modality, called IFTRIED, that represents sentences having S2's form. IFTRIED takes two arguments, a term denoting a plan instance and a logical statement. The statement (IFTRIED  $pi$  P) expresses the counterfactual "if  $pi$  were attempted then P would be true". We will use the phrase *pi's preconditions hold* to mean that  $pi$  would occur if attempted. This can be expressed in the logic as (IFTRIED  $pi$  (PLAN-OCC  $pi$ )) where (PLAN-OCC  $pi$ ) means that plan instance  $pi$  occurs.

It must be noted that IFTRIED can be used to describe what the robot could have done along with what it can do. We are assuming that our logic can represent the robot's view of the world at any particular time. Suppose that we are examining the robot's beliefs at some particular time which we will call the current time. If  $pi$  is a plan instance whose execution time starts before the current time, the statement (IFTRIED  $pi$  P) means that if the robot *had* attempted  $pi$ , P would be true. (IFTRIED  $pi$  (PLAN-OCC  $pi$ )) means that the robot *could have* executed  $pi$ . If  $pi$  has an execution time that starts after the current time, (IFTRIED  $pi$  P) means that if  $pi$  is attempted, P would be true. In the remainder of this paper, we will not explicitly identify the current time and therefore ignore tense distinctions. For example, we will use "can" to mean "can or could have".

### Evaluating Counterfactuals

Before exploring IFTRIED in more detail, it will be helpful to discuss how one evaluates a counterfactual. This is best captured by the following test proposed by Frank Ramsey.

Suppose that you want to evaluate the counterfactual, "if A then C". First you hypothetically add the antecedent A to your stock of beliefs and make the minimal revision required to make A consistent. You then consider the counterfactual to be true iff the consequent C follows from this revised stock of beliefs.

Now, we are interested in how the robot evaluates (IFTRIED  $pi$  C) at some particular time. (IFTRIED  $pi$  C) is considered true iff proposition C is amongst the belief set which is computed by adding " $pi$  is attempted" to the the set of current beliefs about the actual world and making minimal revision for consistency. If the robot

currently believes that  $p_i$  is attempted, no revision is necessary and  $(\text{IFTRIED } p_i \text{ } C)$  is believed iff  $C$  is believed. This is also the case if the robot currently believes that  $p_i$  occurs since necessarily if a plan instance occurs then it is attempted.

Our formal analysis of IFTRIED is based on Stalnaker's [4] treatment of counterfactuals. The intuitions behind his theory agrees with the Ramsy test. His model theory is based on possible world semantics. A function is introduced in the model that takes a possible world  $w_0$  and proposition  $P$  as arguments and returns the "closest" world to  $w_0$  where  $P$  is true. The counterfactual "If  $A$  then  $C$ " is evaluated as true at possible world  $w_0$  iff  $C$  is true in the closest world to  $w_0$  where  $A$  is true.

Stalnaker emphasized that most matters of "closeness" are decided by pragmatics, not semantics. He did, however, specify a minimal set of requirements that a closeness function must meet. These properties lead to a small number of valid axioms. In our model theory, we have introduced an accessibility relation, called CL, that closely parallels Stalnaker's closeness function. His closeness properties have been translated into properties that CL must meet. We also have imposed some additional properties on CL that arise from the specific nature of counterfactuals having the form "If plan instance  $p_i$  is attempted then  $P$  would be true".

The details of the semantic theory, the set of valid axioms, and inference rules will not be presented here. A later paper will cover these issues. The remainder of this paper will provide an intuitive feel for the meaning of IFTRIED by illustrating how it is used.

### Using IFTRIED

We will now show how IFTRIED can be used to encode statements describing what the robot can and cannot affect. Using this knowledge along with beliefs about the actual world (which are encoded by statements not containing the IFTRIED modality), the robot tries to deduce that there exists a plan instance that can be executed and if executed would make the current goal true. As previously mentioned, stating that a plan instance can be executed is equivalent to stating that it would occur if attempted. Therefore, given a logical statement  $G$  representing

the current goal, the robot looks for a plan instance  $p_i$  such that  $(\text{IFTRIED } p_i (\text{PLAN-OCC } p_i))$  and  $(\text{IFTRIED } p_i G)$  are true. (note: typically, we are looking for a plan instance with a future execution time and therefore must add the additional restriction that  $p_i$  must have an execution time starting after the current time). For convenience, we will make the following definition:

$$(\text{CAN-OCCUR } p_i) =_{\text{def}} (\text{IFTRIED } p_i (\text{PLAN-OCC } p_i))$$

Now, it might be the case that  $(\text{CAN-OCCUR } p_i)$  is false, but if another plan instance would be executed then  $(\text{CAN-OCCUR } p_i)$  would be true, that is, there exists some  $p_{i2}$  such that  $(\text{IFTRIED } p_{i2} (\text{CAN-OCCUR } p_i))$  holds. This seems to be closer to the english usage of the word "can". One might say that some action can be performed when it is known that this action can only be performed in conjunction with some other action that brings about its preconditions.

#### Conditions that the Robot Cannot Affect

A proposition will be called *inevitable* iff the proposition is true and would remain true no matter what plan instance the robot attempts. Examples of inevitable propositions are statements about causal laws and statements that relate a plan instance to its preconditions and effects.  $(\text{INEV } P)$  will be used to represent that  $P$  is inevitable.  $\text{INEV}$  is defined in terms of  $\text{IFTRIED}$  as follows

$$(\text{INEV } P) =_{\text{def}} (\text{AND } P (\text{FORALL } ?p_i (\text{IFTRIED } ?p_i P)))$$

If some proposition is inevitable, then the proposition is inevitable no matter what plan instance the robot attempts. This can be represented by the following statement which is a valid axiom schema in our system:

$$(\text{IF } (\text{INEV } P) (\text{FORALL } ?p_i (\text{IFTRIED } ?p_i (\text{INEV } P))))$$

where  $P$  is any logical statement

This schema is used when we are reasoning about nested  $\text{IFTRIED}$  statements. From the assertion that some causal law  $P$  is inevitable, we can deduce that  $(\text{IFTRIED } p_{i1} (\text{IFTRIED } p_{i2} P))$  is true for any plan instance terms,  $p_{i1}$  and  $p_{i2}$ .



If a plan instance occurs then each of its effects holds, no matter what the robot does or could have done. The effects of (walk home store)@i3:10-3:30 are that the robot is outside during execution and is at the store immediately after execution. Therefore, it is inevitable that if (walk home store)@i3:10-3:30 occurred, the robot would be outside during interval i3:10-3:30 and at the store immediately after 3:30 (i.e. during some interval that is met by i3:10-3:30). This can be represented by the following:

```
(INEV (IF (PLAN-OCC (walk home store)@i3:10-3:30)
  (AND (HOLDS (at outside) i3:10-3:30)
    (EXISTS ?i-as
      (AND (HOLDS (at store) ?i-as)
        (MEETS i3:10-3:30 ?i-as)))))))
```

From the assumption (INEV (IF (PLAN-OCC pi) EFF)) which relates pi to one of its effects EFF, the following can be derived:

```
(IF (CAN-OCCUR pi)
  (IFTRIED pi EFF))
```

This statement says that if pi can be executed then EFF would be true if pi is attempted. The reason why this is deducible from (INEV (IF (PLAN-OCC pi) EFF)) is as follows:

- 1) (IFTRIED pi (IF (PLAN-OCC pi) EFF)) is entailed by our assumption (INEV (IF (PLAN-OCC pi) EFF))
- 2) By definition, (CAN-OCCUR) is equivalent to (IFTRIED pi (PLAN-OCC pi))
- 3) Modus ponens distributes out of the IFTRIED modality, that is, the following is a valid axiom schema:

```
(IF (IFTRIED pi (IF P Q))
  (IF (IFTRIED pi P) (IFTRIED pi Q)))
where P and Q are any logical sentences
```

4) From 1-3, it is easily seen that  $(\text{IF } (\text{CAN-OCCUR } p_i) (\text{IFTRIED } p_i \text{ EFF}))$  logically follows from our assumption

### A Simple Example

Suppose that the robot's goal is to get to the store just after 3:30. We will let  $G$  represent this goal statement. If the robot currently believes that it will be at home just prior to 3:10, it will be able to deduce that it can execute  $(\text{walk home store})@i3:10-3:30$  and thereby achieve its goal. On the other hand, suppose that the robot believes that it will be at home from 2:30 to 3:00 but has no beliefs about where it will be from 3:00 to 3:30. In this situation, the robot cannot deduce that it can execute  $(\text{walk home store})@i3:10-3:30$ . If, however, we could introduce another plan instance that guarantees that the robot will be at home just prior to 3:10,  $(\text{walk home store})@i3:10-3:30$  can be executed and the goal achieved.

Let us now consider the plan instance where the robot stays at home between 3:00 and 3:10. We will let the function term  $(\text{stay-at home})@i3:00-3:10$  denote this plan instance. Its preconditions are that the robot is at home just prior to 3:00 and its effects are that the robot will be at home between 3:00 and 3:10. From the belief that the robot is at home between 2:30 to 3:00, it can be derived that  $(\text{stay-at home})@i3:00-3:10$  can be executed and if executed the robot can execute  $(\text{walk home store})@i3:10-3:30$ . This can be represented as follows

```
(AND (CAN-OCCUR (stay-at home)@i3:00-3:10)
      (IFTRIED (stay-at home)@i3:00-3:10
               (CAN-OCCUR (walk home store)@i3:10-3:30)))
```

Since successfully executing  $(\text{walk home store})@i3:10-3:30$  brings about the goal  $G$  (i.e. the robot is at the store just after 3:30), we have the following:

```
(IFTRIED (stay-at home)@i3:00-3:10
         (IFTRIED (walk home store)@i3:10-3:30 G))
```

The above example suggests that goal  $G$  can be obtained by executing a plan instance composed of the plan instances  $(\text{stay-at home})@i3:00-3:10$  and  $(\text{walk$

home store)@3:10-3:30. In this example, this happens to be the case. We cannot, however, always deduce that a plan instance composed of two plan instances, pi1 and pi2, can be executed together when the following is true

```
(AND (CAN-OCCUR pi1)
      (IFTRIED pi1 (CAN-OCCUR pi2)))
```

This issue will be addressed in the next section.

### Plan Instance Interaction

As just illustrated, statements with nested IFTRIED modalities can be used to reason about the interaction of two plan instances (remember that CAN-OCCUR is defined in terms of IFTRIED). It is important to note that the plan instance arguments in a nested statement can have any temporal ordering. In other words, the statement (IFTRIED pi1 (IFTRIED pi2 P)) can be formed regardless of the temporal relation between pi1 and pi2. This is useful when we are reasoning about the combined effects of two plan instances with overlapping execution times.

Consider the function term (stay-at home)@3:00-3:30 which denotes the plan instance where the robot stays at home between 3:00 and 3:30. Assuming that the robot currently believes that it will be at home just prior to 3:00, it can deduce the following.

```
(AND (CAN-OCCUR (stay-at home)@3:00-3:30)
      (IFTRIED (stay-at home)@3:00-3:30
                (CAN-OCCUR (walk home store)@3:10-3:30)))
```

This is because execution of (stay-at home)@3:00-3:30 makes it true that the robot will be at home just prior to 3:10. Unlike the example in the previous section, it is not the case that we can combine (stay-at home)@3:00-3:30 and (walk home store)@3:10-3:30 to form a composite plan instance. This is under the assumption that the robot knows it cannot be at two places at once. If (walk home store)@3:10-3:30 occurs then the robot will be outside between 3:10 and 3:30 and if (stay-at

home)@3 00-3 30 occurs then the robot will be at home between 3 00 and 3 30. Therefore, the two plan instance cannot occur together.

One might ask under what conditions can we conclude that plan instances  $pi_1$  and  $pi_2$  can be executed together when it is known that  $pi_1$  can be executed and if attempted,  $pi_2$  can be executed. That is, the following is known

C1) (AND (CAN-OCCUR  $pi_1$ )  
(IFTRIED  $pi_1$  (CAN-OCCUR  $pi_2$ )))

This happens to be the case if it is inevitable that attempting  $pi_2$  does not preclude the occurrence of  $pi_1$ . By this we mean the following is true:

C2) (INEV (IF (PLAN-OCC  $pi_1$ ) (IFTRIED  $pi_2$  (PLAN-OCC  $pi_1$ ))))

C2 can be read as: it is inevitable that if  $pi_1$  occurs then  $pi_1$  would still occur even if  $pi_2$  were to be attempted. We must note that C2 is a sufficient but not necessary condition that guarantees that  $pi_1$  and  $pi_2$  can both be executed assuming C1 is true.

If it happens to be the case that  $pi_1$ 's execution time is before  $pi_2$ 's execution time, condition C2 will necessarily hold. This is because we are assuming that it is inevitable that attempting a plan instance does not affect any conditions prior to the plan instance's execution time.

### Conclusion

The logic presented in this paper extended a robust temporal logic (i.e. Allen's temporal logic) with a counterfactual-like modality that can be used to represent sentences describing how the robot can affect the world. This allowed us to represent planning problems having the following form:

Given a partial description of the past, present, and future, and a set of goal conditions, find a plan instance (which can be composed of concurrent action



instances) that can be executed and if execute would make the goal conditions hold

This extends the class of planning problems typically handled by situation calculus in the following ways. One can represent knowledge about the past and future along with knowledge about the present state. Plan instances can contain concurrent actions and have any execution time; they are not restricted to be sequences of actions to be executed in the current situation. Finally, any temporal statement can be a goal statement; we are not restricted to goals that describe conditions that must hold just after plan execution.

The IFTRIED modality can be used to specify what propositions can and cannot be affected by the robot's actions. Without making extensions, neither McDermott's or Allen's logic could encode this type of knowledge. We have shown how to represent that some proposition is true no matter what the robot does. We have also shown how to represent that some condition can not be affected by the robot's actions, such as whether or not it is raining.

By nesting the IFTRIED operator, we can represent how plan instances interact with each other. We presented sufficient conditions that guarantee that two plan instances can be executed together. Other forms of interaction can also be represented. For example, we can represent that two plan instances can be separately executed, but cannot be executed together. This might be the case if the two plan instances had concurrent execution times and shared the same resource.

#### Acknowledgements

The research described in this paper was supported in part by RADC under Grant SU 353-9023-6. Thanks to James Allen for comments on the initial drafts.

### References

- [1] J. A. Allen, "Towards a General Theory of Action and Time", Artificial Intelligence, vol. 23 no. 2, pp. 123-154, 1984
- [2] A. Haas, "Planning in a Changing World", BBN Laboratories, Cambridge Mass., 1984
- [3] D. McDermott, "A Temporal Logic for Reasoning About Processes and Plans", Research Report 196, Dept. of Computer Science, Yale University, 1981
- [4] R. Stalnaker, "A Theory of Conditionals" in W L Harper, R. Stalnaker, and G. Pearce (eds.) IFS, Dordrecht, Holland: Reidel Publishing Company, 1981, pp. 41-55
- [5] S. A. Vere, "Planning in Time: Windows and Durations For Activities and Goals", Research Report Jet Propulsion Lab., Nov 1981
- [6] D. Wilkins, "Domain Independent Planning: Representation and Plan Generation", Technical Note 266r, SRI International, May 1983

# COMPUTER ARCHITECTURES FOR VERY LARGE KNOWLEDGE BASES

P. Bruce Berra  
Electrical and Computer Engineering  
Syracuse University  
Syracuse, New York 13244-1240

## Introduction

The current state of the art in knowledge based expert systems is such that the intensional database (IDB) of rules and the extensional database of facts (EDB) are small and main memory resident. However, there is a current need for expert systems with large and very large knowledge bases. With these systems comes the problem of the efficient management of the knowledge base. Data base management system technology can help with smaller knowledge bases but when real time requirements and a very large knowledge base are involved one must consider innovative hardware approaches.

Thus, the long term goal of this research project is to develop innovative computer architectures that efficiently manage very large knowledge bases in a real time environment.

There are many ways to represent the knowledge in an expert system. We have chosen a logic programming framework because of its strong mathematical foundation, its commonality with relational data base management, prior and current Prolog and MetaProlog work at Syracuse University and the potential for making significant improvements in the performance of logic programs through the exploitation of search parallelism.

In this report we begin with a description of the partial match retrieval problem that results when one seeks to retrieve

facts from the EDB. We then discuss our approach to the improvement of the performance of the partial match retrieval problem and present an initial computer architecture. Finally, we discuss accomplishments to date and future plans.

### Partial Match Retrieval

In our initial investigations we have taken the approach that the intensional data base (IDB) of rules is separate from the extensional data base (EDB) of facts. We assume that the inferencing engine goes about the process of solving logic programming problems through processing of the rules and making calls to the EDB to obtain facts that are needed in the process. As previously mentioned we are concerned with a very large knowledge base of facts and high access time requirements.

In order to illustrate that the problem at this level becomes one of partial match retrieval consider the following fact type.

HAPPENING (Person, Event, Data of Event)

One instance of this fact type can be the following:

HAPPENING (J. Jones, Graduated, June 1982)

This is interpreted as the fact the J. Jones graduated from something in June 1982. In the process of solving a logic program the query to the EDB could translate into any of the following queries:

What happened to J. Jones?

Who graduated?

What events took place in June 1982?

When did J. Jones graduate?

What happened to J. Jones in June 1982?

Who graduated in June 1982?

Did J. Jones graduate in June 1982?

and many others.

The point here is that access may be required on any subset of the fact type arguments. The queries above represent access on one, two and all argument positions. The problem becomes a special case of the partial match retrieval problem. It is a special case because we, in general, will specify in what argument position we expect to find the value we are seeking.

The partial match retrieval problem can be solved by first indexing on fact type and then creating an index on each argument position within a fact type. While this approach effectively solves the problem for small EDB's it is ineffective for a VLKB. Since we are indexing on each argument position the index data can be as large as or larger than the facts themselves. In a small EDB with just a few megabytes of fact data doubling the size of the data base is not a severe price to pay for retrieval performance. However, if one has several gigabytes of fact data, doubling the total amount of data by creating indexes on each argument position does not represent a viable solution to the problem. This is true not only from the storage point of view also from the accessing point of view since one must manage much more data. Thus, other methods must be found, when dealing with gigabytes of data that give improved performance.

### Surrogate Files

Over the past year we have made detailed comparative evaluations of techniques that reduce the size of the index data and offer improved retrieval performance. The principal techniques that we have evaluated include superimposed code words (SCW), concatenated code words (CCW), combinations of SCW and CCW and transformed inverted lists (TIL). We will use superimposed code words to illustrate the ideas.

In our previous example suppose we use a hashing function to transform each of the arguments to a fixed binary representation (say 32 bits). That is,

H(J. Jones)	100...1
H(Graduated)	110...0
H(June, 1982)	<u>110...1</u>
Logical OR result	110...1

We can logically OR the individual binary code words to form a superimposed code word. We would then attach a unique identifier to the code word. This unique identifier would be the same one that is used to identify the fact in its complete form. Thus, there would be a surrogate file of SCW's with one entry per fact and this file would be used to improve retrieval performance.

As illustrated in Figure 1 if one wanted to retrieve a fact based upon one or more arguments, one would pass each of the arguments through the same hashing function, and OR their fixed length binary representation together in order to generate the query code word

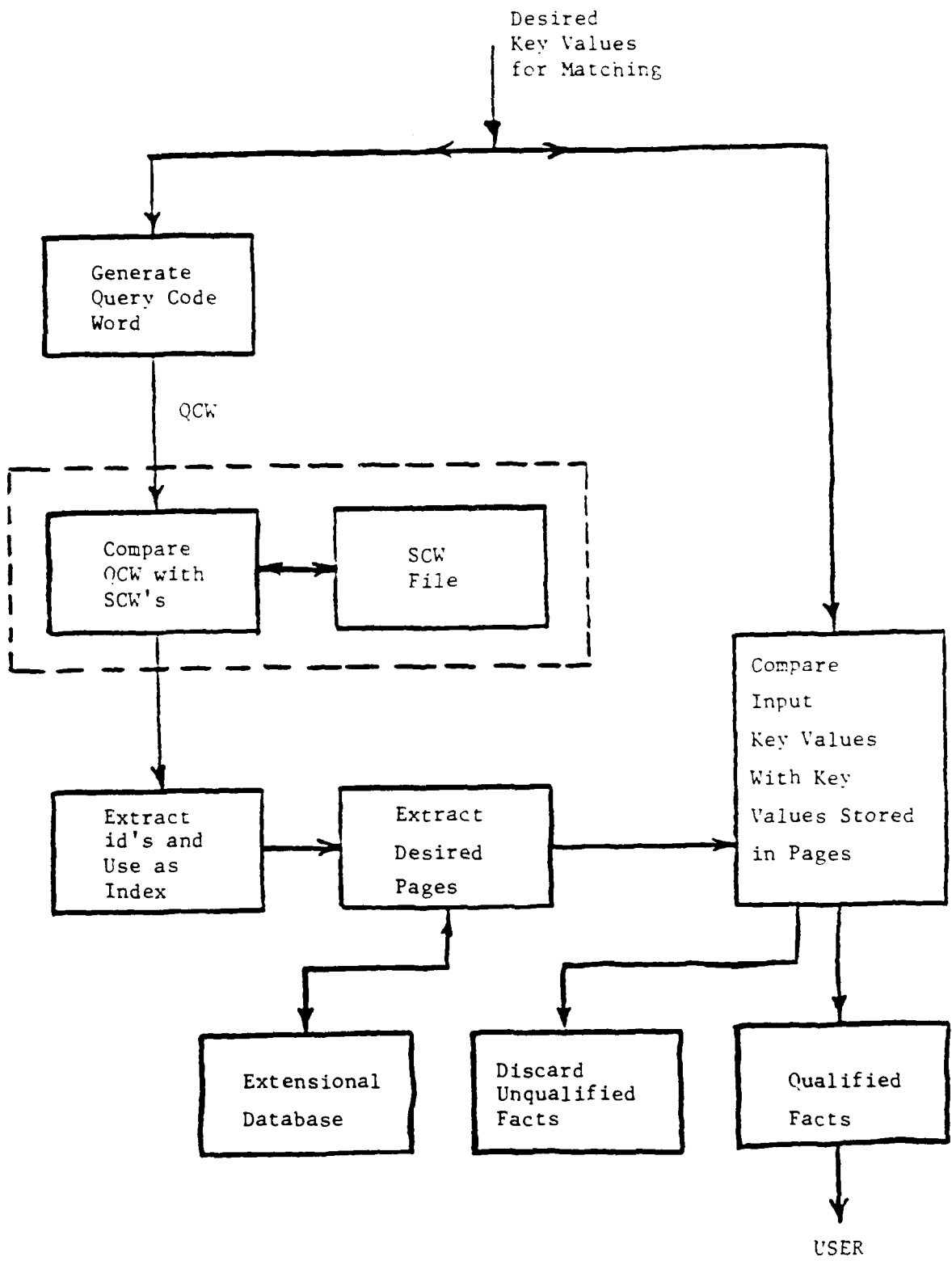


Figure 1 Partial Match Retrieval Using Superimposed Code Words (SCW).



(QCW). One would then compare the QCW with all of the SCW's for that fact type seeking matches on the ones of the QCW. This process is well suited for as associative memory and special hardware has been prototyped by Ahuja and Roberts (1980). All of the SCW's that had ones in the same positions as the QCW would be possible qualifiers. The id's of the possible qualifiers would then be used to retrieve the pages in question. The input key values would then be compared with the possible matches. If the desired fact(s) exists within the data base it will be found. However, a number of pages may be retrieved that do not contain desired facts, called false drops. There is an inverse relationship between the number of bits used in the SCW and the number of false drops. The use of SCW's in the context of logic programming has also been discussed by Wise and Powers (1984).

In the use of concatenated code words we would concatenate the binary representation along with the unique identifier to form the CCW. The use of CCW's is one extreme while the method of SCW's is the other. The concatenation method almost insures (subject to the quality of the hashing function in avoiding collisions) that the unique identifier(s) that is obtained is indeed the one(s) that is desired. However, it does so at the expense of a longer word length. The method of SCW's reduces this long word length but does so at the expense of increasing the number of pages retrieved that do not contain desired facts.

In order to fix the ideas consider the example three argument fact type. When we attach the unique identifier it becomes a relation in the relational data base management context and each of

the facts can be referred to as a tuple. We will assume that the relation comes from a much larger EDB, the tuples consist of a MByte of data and there are approximately 15,500 facts involved. To develop a concatenated code word with unique identifier for each fact would require about 20 bits for each argument position and 14 bits for the unique identifier. Thus, the total number of bytes of CCW would be 144 KB or 14.4 percent of the total. With SCW comparable values are about 80 KB or 8 percent of the total. This number is obtained by allowing 14 bits for the unique identifier and 28 bits for the SCW. Roberts (1979) suggests that the length of the SCW should be such that the number of ones and zeros occur with equal probability in order to minimize false drops. The value of 28 bits for the SCW may vary by a few bits with more detailed analysis but the basic point is that there is not a lot of difference between a SCW method and the concatenation method for small numbers of argument positions in fact types. The difference occurs when there are many arguments per fact. In this case the method of SCW may be of considerable value.

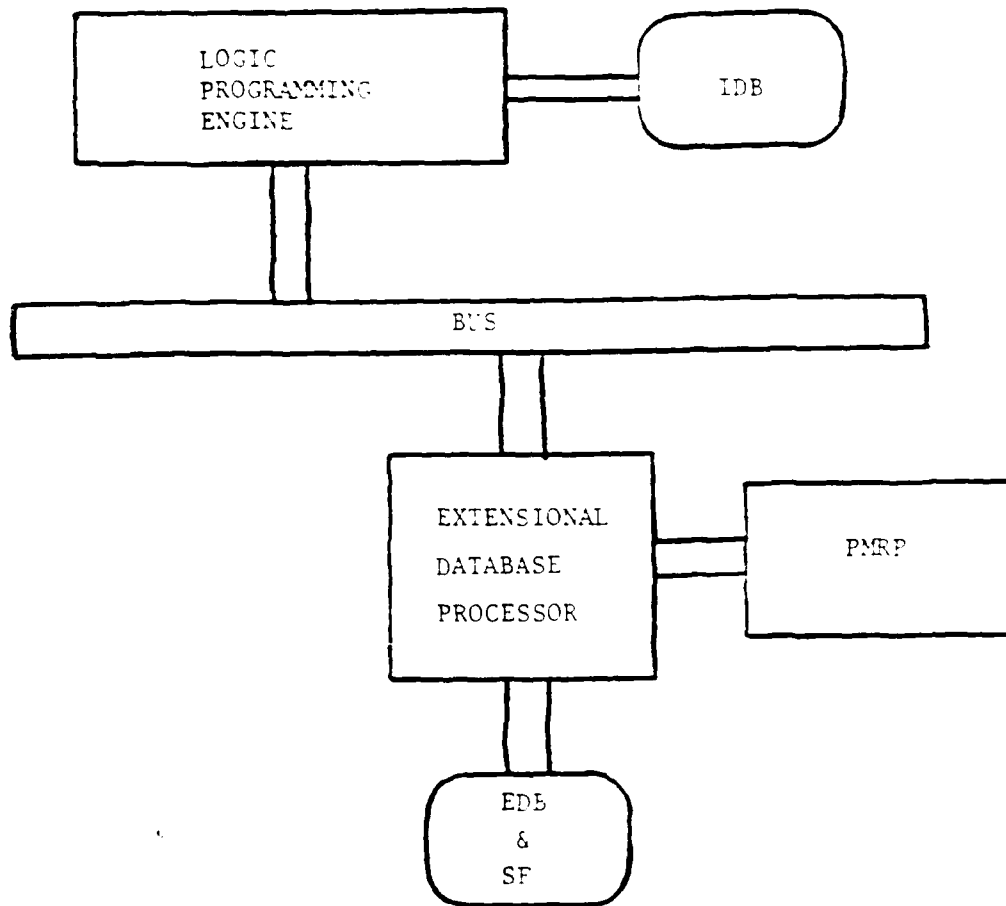
It may be that there is a point in between these two methods that is optimal. For instance, Lloyd et al (1980, 1982) has taken an interesting approach in which he selects bit values from various positions of the binary representation of the argument values in the facts and concatenates them to form a code word. Also, we have explored the development of a method that is a hybrid of the SCW method and the concatenation method. The resulting code word has unique portions for all of the arguments and non-unique portions that represent the ORing of two or more arguments. While this method

While this method reduces the length of the code word it does however, increase the probability of false drops from that of CCW.

Another approach that we are investigating is the use of transformed inverted lists. In this method we hash the original argument values for each position and keep a list with pointers to the original facts. If one is searching for a fact based on a single argument this method has definite advantages since less index data needs to be processed. However, when a number of arguments are involved additional processing will be required since lists will have to be intersected. Also, updating the list of facts will be more difficult than with SCW and CCW. We are in the process of preparing a report in which we compare these various techniques.

### Computer Architecture for Partial Match Retrieval

In addition to improving the performance of partial match retrieval through the use of surrogate files one can gain an additional measure of performance through the use of special hardware. Shown in Figure 2 is our current architectural configuration. We assume that the intensional data base (IDB) is managed by the inferencing engine (although it need not be) and that the EDB is managed by the back end system. The EDB processor is envisioned to be a sequential computer with a data base management system running on it that will manage the facts. It may also be involved in the management of the surrogate file but that has not been determined as yet. The partial match retrieval processor is a special piece of hardware that will be designed especially for the processing of the surrogate file depending upon what indexing method we choose.



- PMRP - PARTIAL MATCH RETRIEVAL PROCESSOR
- IDB - INTENSIONAL DATABASE
- EDB - EXTENSIONAL DATABASE
- SF - SURROGATE FILE

Figure 2 Architecture for Partial Match Retrieval.

### Summary of FY 85 Research

During this reporting period most of our current investigations have focused on two related areas; 1) the development of techniques for assessing the extensional database (EDB) of facts in minimum time and 2) the development of parallel computer architectures that can further speed up EDB processing. When one needs to satisfy subgoals in a query, access to the EDB can take place on any subset of arguments that exist in the clause type under consideration. The problem then becomes one of partial match retrieval with some form of indexing over all argument positions. In order to reduce the amount of index data we are investigating the use of surrogate files that are composed of superimposed code, concatenated code words, transformed inverted lists or some combination of the three. These are transformations of EDB key values into binary representations. While a code word exists for every fact in the EDB the total code word file is on the order of 20% of the size of the EDB of facts. This is compared with sizes of upwards of 100% for other methods of indexing. As pointed out previously we are in the process of preparing a report on the results of our analyses.

With regard to the development of parallel computer architectures we have postulated a back end system that consists of a control processor, a special purpose partial match retrieval processor and disks for the storage of the surrogate file and the EDB. We are currently investigating several approaches to the partial match retrieval processor from associative processors to streaming processors.

In addition to those tasks discussed above we are investigating the interface between Prolog/Metaprolog and a relational database management system (INGRES). We are also engaged in a review of the state of the art in optical storage devices and optical processors with a view toward the use of such devices in our future research. Finally we have begun a study of various expert system builders.

## Research Plans for FY 86 and 87

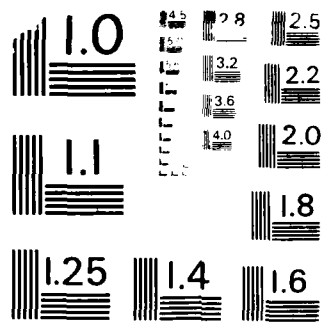
During the next year (1986) we will continue our evaluation of partial match retrieval methods and special computer architectures for their efficient implementation all in the context of logic programming. We expect to begin the design of the special purpose processor about midway through the year. We will begin to assemble the necessary hardware, software and knowledge for the knowledge base back end system as shown in the initial architecture. We will have to pay special attention to those functions that are required for a complete system. These functions include creating the EDB, back up, update, integrity, security and others. In addition we will need to identify an expert system application area in order to ground our research.

We will also continue to investigate the use of optical storage devices with their potentially high bandwidths (200-400 MBytes/Sec.) in order to increase the rate at which the EDB can be processed by the back end system. If resources permit we will also investigate processing of the data in optical form.

There is considerable interest in architectures that increase the rate at which logical inferences can be performed. As these processors develop, increased emphasis will need to be placed on the management of the knowledge base. There are many open questions regarding how the inference processor, the knowledge base processor and the other important parts of the system can be integrated. Thus, our research will concentrate on the development of the processor and the knowledge base a prototype Prolog/Metaprolog system. This research is being carried out with other research being carried out at the same time.







MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS - 1963

### Research Plans for FY 88 and 89

If we are to effectively handle VLKB's we must deal with the storage of the knowledge in disk form. Since the bandwidth (3-5 M Bytes/Sec.) of magnetic disks is not expected to increase much during the time period of this research we must consider the distribution of the EDB over many disks. This will allow for simultaneous parallel read out from the disks, thus effectively increasing the bandwidth. Another way to increase the bandwidth of the disks is to compact/decompact the data being sent to and from the disks. Estimates are that compaction can effectively increase the bandwidth by a factor of two. Thus, with the minimization of the amount of extra data through the use of surrogate files and the methods described above we hope to be able to deal effectively with the magnetic disk bandwidth problem. Finally, we expect a positive impact based upon our optical disk research.

In the time period 1988 through 1989 we plan to construct an advanced architecture as illustrated in Figure 3. We will use the techniques described above for aiding rapid EDB retrieval but many other problems will arise regarding distributing the EDB, coordination and communication among the processors, the numbers of processors of various types needed for effective operation of the system, the establishment of a VLKB and the interfacing to the inferencing mechanism. Our long term goal then is to be able to demonstrate such a system by the end of the project.

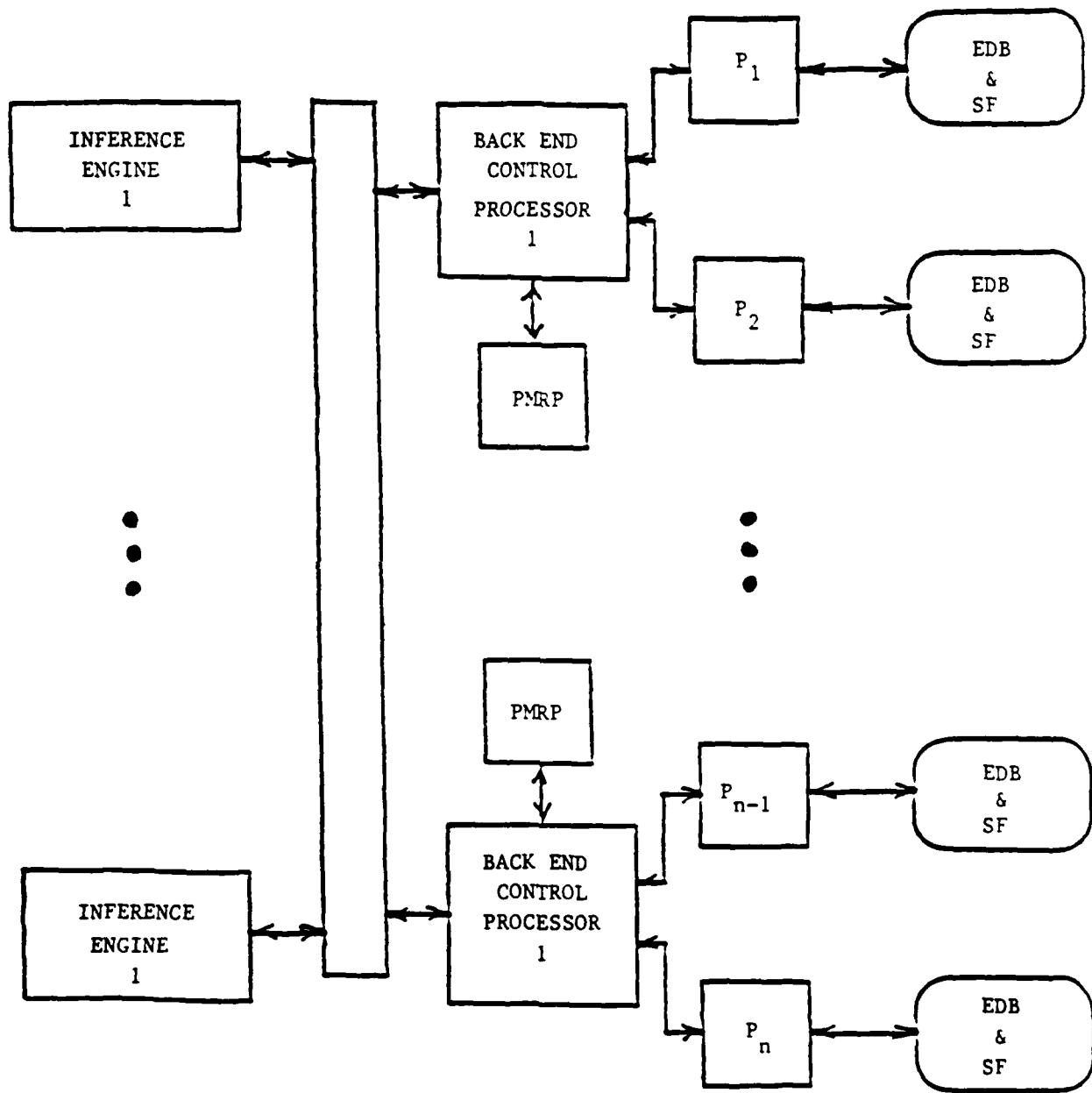


Figure 3 Advanced Architecture

### References

Ahuja, S.R., C.S. Roberts, "An Associative/Parallel Processor for Partial Match Retrieval Using Superimposed Codes," International Conference on Parallel Processing, pp. 211-227, August 1980.

Lloyd, J.W., "Optimal Partial-match Retrieval for Dynamic Files," BIT 20, pp. 406-413, 1980.

Lloyd, J.W. and K. Ramamohanarao, "Partial-Match Retrieval for Dynamic Files," BIT 22, pp. 150-168, 1982.

Wise, M.J. and D. Powers, "Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words," 1984 International Symposium on Logic Programming, Atlantic City, NJ, February 1984.

# Knowledge Base Maintenance Using Logic Programming Methodologies

Annual Report: 1984 - 1985  
Artificial Intelligence Consortium

Kenneth A. Bowen  
Logic Programming Research Group  
School of Computer and Information Science  
313 Link Hall  
Syracuse University  
Syracuse, NY 13210

## Summary

The on-going work of this project is focused on the development of extensions of the logic programming language Prolog which are suitable for application to the problem of maintaining consistency and logical structure for large dynamic knowledge bases. The logical representation of the assertions of a knowledge base is to identify them with the facts and rules of the logic programming language. Such a collection of facts and rules is called a theory. To adequately model the dynamic character of the knowledge base, one must be able to efficiently manipulate alternative theories, a (virtual) sequence of such alternative theories providing a representation of the changing knowledge base. Moreover, it is desirable that the process of manipulating alternative theories be logically well-grounded. Present-day Prolog, while it has highly efficient implementations, does not meet these requirements. The project is developing so-called metalevel extensions of Prolog meeting the following requirements:

- A) The extension allows one to express alternative and changing KBs;
- B) The extension has a logical basis;
- C) The implementation methodology for Prolog can be expanded to efficiently implement the metalevel extensions.

The metalevel extensions have the character that logical concepts which are implicit in Prolog systems are made explicit in the extension. In particular, theories, which are only implicit in Prolog, become explicit first-class objects capable of being the values of variables and of being dynamically constructed and modified in a logical manner. While the immediate motivation for the reification of theories was the representation of change in knowledge bases, a pleasant side-effect is the ability to cleanly implement a number of classical artificial intelligence knowledge representation schemes such as frames and semantic nets. A similar approach is being taken to the problem of control, though it is not yet as well developed.

## Background and Goals

This research is directed at the development of logic-based methods for manipulating and maintaining large complex knowledge bases. The focus of concern is with the problem of reason or truth maintenance. Assertions may be recorded in the knowledge base which may later need to be withdrawn, either because the assertion was incorrect, though this was unknown, or the assertion may have been a hypothesis tentatively added to examine its consequences in the presence of the rest of the knowledge base assertions. The difficulty arises from the possibility that other assertions may be deduced from the problematical assertion and recorded in the KB. When we later go to withdraw the problematical assertion, we must locate all other assertions in the KB which apparently depend on it, determine whether or not they possess support other than the problematical assertion, and if they do not, withdraw them also, recursively iterating this process to their dependents, etc. The approach taken by this project is from the point of view of logic programming, since the central concern of reason maintenance is deductive consequence and consistency.

The most successful exemplar of the logic programming paradigm to date is Prolog. Unfortunately, it is inadequate to the task of large scale knowledge base maintenance. The most desirable way to represent a knowledge base in Prolog is to identify the assertions of the KB with Prolog facts and rules. However, because pure Prolog only provides for one monolithic program database, one cannot directly represent a changing KB. Practical Prolog adds the assert/retract facility to accomplish this. However, this has negative aspects:

- 1) *the logical basis of the system is destroyed;*
- 2) *since the KB is intertwined with the management code, this is very poor programming practice for large systems;*
- 3) *it does not allow one to express and maintain alternative KBs or contexts.*

However, the implementation techniques of Warren [1983] for pure Prolog are extremely powerful, efficient and flexible. Consequently, the approach taken by this project has been to seek an extension of Prolog with the following properties:

- A) *the extension allows one to express alternative and changing KBs;*
- B) *the extension has a logical basis;*
- C) *one can extend Warren's implementation techniques to provide efficient implementations of the extension.*

## Principal Efforts

The first year of the project has been devoted to this task. We have explored a class of extensions of Prolog which can be described as meta-level extensions. Our activities have been divided among the following interrelated subtasks:

- i) Elaborating various versions of the metalevel constructs and examining their applicability to the problem of KB maintenance and also to the implementation of expert systems, since the latter is often intertwined with the use and maintenance of KBs.
- ii) Elucidating the logical status of the metalevel constructs;
- iii) Examining methods of extending Warren's methods to accommodate the constructs.

As a basic part of this effort, we devoted considerable effort during the year to the construction of a high-quality Prolog implementation based on Warren's techniques. We explored a number of approaches to this, and eventually constructed a Warren abstract machine (WAM) which executes abstract instructions emitted by two distinct compilers. The WAM is a byte-code interpreter implemented in C. As such it is quite portable (to byte-addressable machines which includes the DEC VAX architecture, the Motorola 68000-series architecture, the National Semiconductor 32000-series architecture and numerous others), yet is very efficient. It executes the standard naive reverse benchmark at 12K LIPS on the VAX 780. (While this is really a poor benchmark of Prolog performance, it provides a rough yardstick by which to compare systems.) It is by far the fastest of the portable systems to date. It executes at slightly better than one-half the speed of Warren's Quintus Prolog product.

One of the two compilers which emits code for this WAM is coded in Prolog, the other in C. The latter is integrated with the WAM, incrementally compiling and loading the code presented to it. The other is coded in Prolog. It can be compiled by the C version and run in the WAM environment, loading its code for execution by the WAM. The process of constructing these two compilers taught us much about the subtleties of Prolog compilation. Details of these systems and some of the compilation techniques we have evolved are presented in the papers attached as appendices to this report.

We devoted several months to exploring the problem of creating interpreters for the metalevel extensions by writing them in Prolog and executing them over our WAM (extended by a few simple low-level constructs to provide for simulation of the metalevel notions.) While we had some success at this, overall we found too many difficulties in the approach. Simply put, the few low-level extensions of the WAM that we had introduced were inadequate to the task. Using them to simulate the metalevel constructs produced code which was simply too complex, and failed to capture all of the intended functionality of the constructs.

Thus, towards the end of the year, we turned our full attention to the question of truly extending the WAM to provide suitable facilities for the correct implementation, via a



compiler, of the metalevel extensions. Fundamentally, the metalevel extensions are governed by the position that important logical concepts which are implicit in Prolog must be made explicit in the metalevel extensions. Theories -- that is, collections of formulas constituting a program database -- have occupied the greatest part of our attention. To make them explicit, they must be capable of being the values of variables and they must be dynamically creatable and modifiable. To achieve this, the most important underlying task is a modification of the treatment of compiled code and the space it occupies. Roughly, we must be able to treat code sequences rather like ordinary Prolog terms and the space it occupies must be managed rather like the standard WAM heap. Two options present themselves:

First, simply store the code on the existing heap, adjusting the WAM accordingly;

Second, store the code in a separate area, but add management facilities to the WAM which handle that code area in a manner similar to the heap.

At the moment, we lean towards the second, but feel that there is no vast difference between the two schemes. The differences will lie mostly in matters of detail. Our preference for the second is based on our estimate that it will provide for a somewhat cleaner system. We have designed most of the changes needed for this extension, and expect to start coding them into our present WAM after the end of the present academic semester.

For the metalevel extensions which focus on making the notion of theory explicit, we are convinced that, given the properly extended WAM, the problem of creating a compiler for the metalevel extension will be relatively straight-forward. Linguistically, the metalevel extension (which we call metaProlog) closely resembles ordinary Prolog with the "call(G)" built-in predicate replaced by the predicate "demo(T,G)" which recursively invokes the backchaining Prolog-type search mechanism in the context of the theory or program database T. Consequently, the metaProlog compiler will be a rather straight-forward extension of our present Prolog compiler(s).

### Related Concerns

One of the exciting fall-outs of the work has been the realization that making theories explicit first-class objects allows us to create clean implementations of a number of very popular artificial intelligence knowledge representation schemes. In particular, we are able to implement frames and semantic nets very cleanly and powerfully, including extensive inheritance of properties. We have also been led to discern the possibility of a rather direct implementation of message-passing. However, this latter will require some alteration of the search regime of the underlying deductive engine; we have not yet fully explored the consequences of this. These ideas are set out in more detail in two of the accompanying appendices.

As a secondary issue, we have devoted some attention to the question of extensions

which allow more flexible control of the deductive engine. Our approach here has been governed by the same philosophy which governed the addition of theories. Namely, rather than graft on a non-logical control language, we feel that the underlying concepts involved in control should be made explicit to some reasonable degree. Exactly what form this should take is not yet clear. However, it seems certain that abstract representations of the search space as a whole, the partially completed search, the discarded part of the space, and the as yet unsearched portion of the space should be available as reasonably first-class objects, amenable to direct reasoning by the program. However, while we have some very intriguing ideas as to how to accomplish this, the impact on the WAM seems more extensive and thus we are proceeding more slowly in this line.

### **Future Plans**

Our goals for the immediate future are the coding of the extended WAM and the installations of the accompanying changes to the compiler so as to achieve an operational metaProlog compiler system. When this is achieved, we will enter into an extended period of implementation of prototypical knowledge base maintenance schemes and expert systems using the metaProlog compiler. This will provide not only a testing of the constructed system, but should also iteratively feed back information leading to improvements of the system. We expect the efficiency and capacity of the system to be sufficient to explore some experiments of realistic scale. After some period of this endeavor, we will begin working closely with Professor P. Bruce Berra's effort to integrate the code management schemes of our extended WAM with the (software) systems his project is designing for the low-level management of very large knowledge bases. We will also continue our investigations of control extensions at a secondary level of effort.

### **Research Group Composition**

Besides the principal investigator, Professor Kenneth A. Bowen, the group is composed of a number of students at Syracuse University. Two of them, Ilyas Cicekli and Andrew Turk, are supported by the AIC contract. Another, Kevin Buettner, has support during the academic year from a Syracuse University Graduate Fellowship, but during the summer of 1985 was supported by the Post-Doctoral Program grant under which some of the work preliminary to this effort was conducted. The other student members of the group have support from other grants or University graduate assistantships. They are: Hamid Bacha, Aida Batarekh, Loren Fairbank, Rumi Gonda, and David Wolfram. Another student, Keith Hughes, is employed under Professor P. Bruce Berra's grant, and is working very closely with our group. He is devoting effort towards the integration of the logic programming system(s) with large-scale relational database managers. Currently he has been investigating the problem of interfacing the logic programming system(s) with the relational database system INGRESS as a preliminary step to integration with the systems being designed by Professor Berra's group. It is hoped that several new students will join the group in the course of the

academic year.

### Visitors and Travel

We benefited considerably from a number of visits during the year by colleagues from the logic programming community. Hideyuku Nakashima of the Electrotechnical Laboratory, Ibaraki, Japan, spent two months as a visiting CASE Center scholar. His work is closely related to our own and the regular discussions we had with him were quite enlightening. Jacob Levy of the Weizmann Institute of Science, Rehovot, Israel, visited under the USAF Window on Science Program. Our discussions of implementation techniques with him were extremely helpful. We also spent time discussing questions of concurrency, which, while not of foremost concern for our group at the moment, are of considerable long-term significance for us. Throughout the year we worked closely with the Theorem-Proving group directed by Ewing Lusk and Ross Overbeek at the Department of Energy Argonne National Laboratory, Argonne, Illinois. They have been deeply concerned with questions of Prolog implementation and during the first phase of the year, we made use of their implementation of the WAM, later replacing it with our own. Bowen made a trip to Argonne in October, 1984, for technical discussions, and during the year, both Lusk and Overbeek made separate trips to Syracuse to continue those discussions. Throughout the year, we were able to maintain very close contact via the connection between CSNET (Syracuse) and ARPANET (Argonne). The work would have benefited even more had there been a direct ARPANET connection between the two sites. In May of 1985, most of our group attended a *non-formal Workshop on Prolog Implementation hosted at Argonne*. The value of the discussions we had there was extremely high, leading to some of our key insights. Other visitors whose discussions contributed to our efforts include: Allen Brown of General Electric Corporate Research & Development Laboratories, Schenectady; Jean-Louis Lassaize of IBM Yorktown Heights Research Center; Lee Naish of Melbourne University; K. Ueda of ICOT; Maarten van Emden of the University of Waterloo; and Tobias Weinberg of Digital Equipment Corporation. In July of 1985, the group attended the Symposium on Logic Programming in Boston where we derived considerable benefit from a large number of informal conversations.

# Meta-Level Programming and Knowledge Representation

**Kenneth A. Bowen**

Logic Programming Research Group  
School of Computer & Information Science  
Syracuse University  
Syracuse, NY, 13210 USA  
CSNET: kabowen@syr  
ARPANET: kabowen%syr@csnet-relay

## Abstract

The nature of a metalevel extension of Prolog is outlined. The key features include the treatment of theories (databases) and metalevel names as first-class objects which may be the values of variables. The use of the power of these constructs in traditional knowledge representation is explored. In particular, it is shown how frames, semantic nets, scripts, message passing, and non-standard control can be represented.

## 1. Introduction

Beginning in Bowen and Kowalski [1982] and continuing in Bowen and Weinberg [1985], we have explored metalevel extensions of the basic first-order logic programming paradigm, and especially of Prolog. The aim of these investigations has been to provide a powerful programming language suitable for artificial intelligence applications, while at the same time recapturing a fully logical semantics which was lost through Prolog's addition of such primitives as `assert/retract` to the pure Horn clause paradigm. The use of `assert/retract` in Prolog reflects a real programming need in many artificial intelligence applications, namely the need to segment or modularize the clause database for a variety of purposes. For many applications, a static segmentation of the database would be inadequate, since database segments may need to appear to

---

This work supported in part by US Air Force grant AFOSR-82-0292 and by US Air Force contract F30602-81-C-0169. The author is very grateful to the following people for numerous valuable conversations on the topics of this paper: Hamid Bacha, Aida Batarekh, Kevin Buettner, Ilyas Cicekli, Hidey Nakashima, Andy Turk, Maarten van Ernden, and Toby Weinberg.

change in time or even come into and pass out of existence during run-time execution of the program. Combined with the desire to provide such facilities together with a logical semantics, we adopted the approach of ascending to a metalevel point of view. At its most simplistic, this would consist of a (first-order) metalevel axiomatization of the object level proof theory. That is, the programming system would axiomatize the notion of theory and proof for some object-level language, allowing theories (sets of formulae) to exist as first-class objects in the sense that they could be the values of variables, and thus could be passed into and returned from procedures; in particular, they could be dynamically created or destroyed, and new (modified) theories could be generated from previously existing theories. The basic connection between theories, formulae, and proofs would be provided by the proof predicate

demo(Theory, Formula, Proof)

which holds precisely when Proof is a proof (in the axiomatized system of logic) of Formula based on the axioms in Theory.

However, to exploit the full potential of the metalevel approach, we must avoid the stratification into levels (object, meta, meta-meta, etc.) that the simplistic approach entails. To do this, we must effectively amalgamate the basic object language with all of the upper meta-levels. One approach to accomplishing this is to (conservatively) extend an ordinary first-order predicate calculus language to one in which finite sets of formulas are represented by terms, and in which every syntactic item (from variables to sets of formulas) is named by a constant of the language <sup>(1)</sup>. Note that these constants participate in other terms and formulas, and these latter also are named by other constants. The relation

name\_of(<Item>, <Name>)

is incorporated as part of the basic first-order proof machinery of the extended language. This is accomplished by simply iterating the process of extending a language by adding constants to name all its items in a manner similar to pp. 43-45 of Shoenfield [1967]; in addition, all of the name\_of assertions are added at each iteration. The extension is conservative in the sense that if A is a formula expressed in the original language, and A is seen to be provable in the extension, then it follows that A is provable in the original system.

The facilities provided by such an extension turn out to be surprisingly powerful. Many of the constructs used for knowledge representation in AI programming turn out to have rather direct representations in terms of theories and names. These include frames, semantic nets, and scripts. In the following sections, we will outline an approach to the design of one particular metalevel extension, called metaProlog, and illustrate how frames, semantic nets, and scripts may be represented. In addition, we show how an object-oriented message-passing paradigm may be represented. However, to obtain the correct behavior from this paradigm, execution control other than the standard Prolog control must be available. This leads us to explore one method of implementing non-standard control by means of metalevel knowledge of control mechan-

isms. Given the representation of message-passing and appropriate control, knowledge representation schemes such as blackboards can easily be implemented.

It is important to stress that what we present here is *not* a description of how to implement these ideas in a fixed, precisely specified system, but rather an outline of how *some* extension of Prolog incorporating such metalevel ideas might deal with a large body of existing techniques for knowledge representation. It is an exploration of the power inherent in the metalevel notions. As such, it can be viewed as a presentation of design goals for a precise system.

## 2. The Nature of the metaProlog Extension.

The amalgamation of language and metalanguage described in the last section was originally suggested by Gödel's [1931] famous incompleteness proof in which he used the technique now called "Gödel numbering" to demonstrate that ordinary arithmetic could express substantial portions of its own formal syntax, including a definition of the basic proof relation. Thus, in particular, the amalgamated language allows object-level programs to ascend to the meta-level via calls on the demo predicate. The nature of the proof relation for the amalgamated system can perhaps be most easily seen via an ordinary Prolog axiomatization of the core of the proof relation for the amalgamated system. In the first (definitional) axiomatization below, the structure-sharing approach is expressed by making the necessary substitutions explicit as arguments to the appropriate predicates. The predicate

```
match(Left, Right, In_Substitution, Out_Substitution)
```

is the unifier; it attempts to extend the input substitution to a substitution unifying its first two arguments.

```

demo(Theory, Goal, [], Substitution, Substitution)
:-
  empty(Goal).

demo(Theory, Goal, [Reason | Rest_Proof], In_Subst, Out_Subst)
:-
  select(Goal, SubGoal, Rest_Goals),
  react(Theory, SubGoal, Reason,
        In_Subst, Inter_Subst, Continuation_Goals),
  merge(Continuation_Goals, Rest_Goals, New_Goal),
  demo(Theory, New_Goal, Rest_Proof, Inter_Subst, Out_Subst).

react(Theory, demo(New_Theory, Subsid_Goal, Subsid_Proof), sbs(Subsid_Proof),
      In_Subst, Out_Subst, true).
:-
  demo(New_Theory, Subsid_Goal, Subsid_Proof, In_Subst, Out_Subst).

react(Theory, current(Theory), current(Theory), In_Subst, In_Subst, true).

react(Theory, SubGoal, s(SubGoal, Rule), In_Subst, Out_Subst, Rule_Body)
:-
  find(SubGoal, Theory, Rule),
  parts(Rule, Rule_Head, Rule_Body),
  match(SubGoal, Rule_Head, In_Subst, Out_Subst).

```

Figure 2.1

We have provided two special forms (distinguished predicates) recognized by the system: `demo(, , )` and `current(, )`. (More will be indicated below, and certainly others will arise.) The first clause for `react` allows recursive calls on the `demo` predicate, while the second clause allows a clause to determine the theory under which it is being executed.

If we assume that the metavariables of the metaProlog system being axiomatized are to be identified with the variables of the underlying Prolog system, we can present a more compact axiomatization as follows:

```

demo(Theory, Goal, [])
:-
  empty(Goal).

demo(Theory, Goal, [Reason | Rest_Proof])
:-
  select(Goal, SubGoal, Rest_Goals),
  react(Theory, SubGoal, Reason, Continuation_Goals),
  merge(Continuation_Goals, Rest_Goals, New_Goal),
  demo(Theory, New_Goal, Rest_Proof).

react(Theory, demo(New_Theory, Subsid_Goal, Subsid_Proof),
      sbs(Subsid_Proof), true)
:-
  demo(New_Theory, Subsid_Goal, Subsid_Proof).

react(Theory, current(Theory), current(Theory), true).

react(Theory, SubGoal, s(SubGoal, Rule), Rule_Body)
:-
  find(SubGoal, Theory, Rule),
  parts(Rule, Rule_Head, Rule_Body),
  match(SubGoal, Rule_Head).

```

Figure 2.2

The role of assert/retract in ordinary impure Prolog, modifying the single global database, is replaced by the construction of new theories out of old ones using two primitive built-in relations:

```

add_to(<Old Theory>, <Assertion>, <New Theory>)

drop_from(<Old Theory>, <Assertion>, <New Theory>)

```

Figure 2.3

From a logical point of view, <New Theory> is obtained by first making a copy of <Old Theory> and then modifying this copy appropriately. From a programming language point of view, actual copying is too high a price to pay in general (though it may be necessary in certain restricted circumstances). Instead, the actual implementation must make it appear that such copying has taken place, even if it really hasn't.



This can be achieved by describing one of the old and new theories as a modification of the other. Thus, we could either describe <New Theory> as the result of adding <Assertion> to <Old Theory>, or vice-versa. Of course the implementation of the demo predicate must know how to manipulate such descriptions when retrieving axioms for use in a proof. As is the case with the classic AI frame problem, access to the axioms of the theory represented as a description becomes slow as the number of modifications increases. Since in the typical application of these facilities (add\_to and drop\_from) it is the new theory which will be most heavily accessed, the design decision for metaProlog has been to provide for fast access to the new theory. This is accomplished by letting <New Theory> be the actually modified representation of <Old Theory>, while, after execution, <Old Theory> is left as a description in terms of <New Theory> <sup>(2)</sup>. Versions of the predicates add\_to and drop\_from are also provided which carry out complete copying for occasions when this appears necessary for efficiency.

The intended logical interpretation of such a metalevel extension of Prolog is that of a (first-order) axiomatization of theories and proofs in the amalgamated language. In the usual use of logic, a first-order formal axiomatization has an *intended* interpretation outside of language. For example, the intended interpretation of Hilbert's axiomatization of geometry was a world of geometric figures, while the intended interpretation of Peano's arithmetic was a world of numbers. In these cases, it is almost an accidental after-effect of the formal process that (Herbrand) interpretations built out of linguistic elements can be constructed. In contrast, in this setting, the intended interpretation of a formal metalevel Prolog extension is in a world of language: the metalevel system is intended to talk about theories and proofs of a language. An alternate semantic possibility is to provide the system with a Hintikka-Kripke modal semantics. In either case, the interpretation of names is that they are proper nouns functioning as *rigid designators* in the sense of Kripke [1972]. Thus, if <n> names a certain theory <t>, it will point at some physical manifestation of <t>. If an operation add\_to(<t>, <a>, <t2>) is performed, <n> will subsequently point at the physical manifestation of <t2>. The situation is analogous to that for proper names in natural language. For example, assume 'John Jones' refers to a certain (physical) person. Then, after undergoing an operation for removal of the gall bladder of John Jones, the name 'John Jones' refers to the resulting physical person. These concerns raise difficult foundational issues which will be treated elsewhere.

The metaProlog system is related to the Prolog/KR and URANUS systems of Nakashima [1982] and [1985]: our theories are very similar to his "worlds". The exact logical semantic status of his system is not entirely clear, though he has indicated in conversation that he has considered an S4-type modal semantics for his system. One significant difference is that his system does not appear to provide for meta-level names of all objects. Another closely related system is MANDALA (Furukawa et al. [1984]) which also incorporates a version of the "world" notion and a version of our "demo" predicate. As with Nakashima's systems, the logical status of MANDALA is not clear, nor does it appear to incorporate meta-level naming. The metaProlog system is also closely related to the systems of Miyachi et al. [1984] and Kitakami et al. [1984] since they also grow out of the ideas of Bowen and Kowalski [1981].

### 3. Brief Syntax of metaProlog

The surface syntax of metaProlog is quite similar to that of Edinburgh Prolog, with ':' replaced by '<--' and conjunction indicated by '&' rather than comma. The most significant difference is the requirement that quantification be made explicit, rather than indicated implicitly by a variable convention. As described more fully in Bowen and Weinberg [1985], this requirement follows from two points:

- The desire to allow manipulation of partially instantiated theories, as would follow from the principle that they are to be treated as fully first-class objects;
- The desire to provide a correct semantics for `add_to` and `drop_from`.

The difficulty with regard to the latter can be seen by considering the following two Edinburgh Prolog clauses:

```
h :- X = a, assert(p(X)), p(b).  
h :- assert(p(X)), X = a, p(b).
```

Figure 3.1

In standard Prolog, the first fails, while the second succeeds, since Prolog assumes `assert(p(X))` means that "if X is uninstantiated, add 'all [X] : p(X)' to the database." However, the logical reading of these clauses treats the comma as a conjunction connecting the literals, and logical conjunction is commutative. Thus the two clauses ought to produce the same result. In metaProlog, if the programmer wishes to achieve the effect provided by Prolog's default assumption, he or she must explicitly indicate the quantification:

```
...& add_to(t1, all [x] : p(x), t2) & ...
```

If, on the other hand, 'x' is quantified somewhere in a clause which has been applied, but at run-time, the interpreter (meta)variable replacing x has not yet been instantiated, the effect of the call

```
...& add_to(t1, p(x), t2) & ...
```

would be to add the partially instantiated assertion `p(x)` to `t1`, constructing `t2` as a partially instantiated theory. Later processing could cause x to become instantiated, resulting in more precise specification of the theory `t2`.

Since quantification must be made explicit in metaProlog and consequently, no variable convention is in force, all identifiers beginning with a letter (uppercase or lowercase) are now constants.

#### 4. First Uses of Theories and Names

Since theories are first-class objects, they can participate in clauses in the same manner as constants and other terms. Thus one theory T1 can contain an assertion

useful(T2)

about another theory T2. Thus, for example, consider the problem of defining a database management system. A simple approach to such a system might run as follows:

```

all [CurDB] : dbm(CurDB, []).

all [CurDB, request, requests, NewDB] :
    dbm(CurDB, [request | requests])
    <--
    process(request, CurDB, NewDB)
    & dbm(NewDB, requests).

all [assertion, proof, CurDB] :
    process(retrieve(assertion, proof), CurDB, CurDB)
    <--
    demo(CurDB, assertion, proof).

all [Assertion, Proof, CurDB] :
    process(insert(Assertion, present(Proof)), CurDB, CurDB)
    <--
    demo(CurDB, Assertion, Proof).

all [Assertion, Response, CurDB, NewDB, IC, UnAcProof, RR, RevProof] :
    process(insert(Assertion, revision(Response)), CurDB, NewDB)
    <--
    demo(CurDB, insert_constraints(IC), _)
    & demo(IC, unacceptable(Assertion, CurDB), UnAcProof)
    & demo(CurDB, revision_rules(RR), _)
    & demo(RR, revise(UnAcProof, CurDB, NewDB), RevProof)
    & Response = [Assertion, UnAcProof, RevProof].

all [assertion, CurDB, NewDB] :
    process(insert(assertion, done), CurDB, NewDB)
    <--
    add_to(CurDB, Assertion, NewDB).

```

Figure 4.1

The database manager is represented by the two-place predicate 'dbm'. The first argument of dbm is a theory representing the current state of the database, while the second argument is a stream (list) of requests to be processed against the database. As can be seen, dbm is a simple tail-recursive procedure, with all the real work being done by the three-place relation 'process'. The first argument of process is the request to be processed against the current database which is contained in the second argument of process, while the state of the database resulting after this processing is represented by the theory in the third argument of process. The first clause for process deals with simple retrieval (which includes implicit information deducible from the database), while the last three deal with addition of new assertions; depending on the nature of

the insertion constraints and revision rules used in the third clause, this can also handle updates. The second clause for process reflects a parsimonious philosophy on the part of this particular database manager: nothing is added which is already explicit or implicit in the database. Whether this is an appropriate philosophy, or just how much resources should be expended in the attempt to determine whether something is implicit, is a matter of the particular philosophy of each database management system. The fourth clause of process is a default: if none of the other clauses are applicable to a proposed addition, then add it to the database.

The third clause embodies the possibility of management of integrity constraints and reason maintenance. It also illustrates the flexibility possible in the use of theories. In this case, the database carries with it the collection of integrity constraints in the form of an assertion

insert\_constraints(IC)

in which IC is intended to be another theory embodying the integrity constraints to be used. It is assumed that IC contains clauses defining the two-place predicate "unacceptable". If the proposed assertion can be proved unacceptable under the rules embodied in IC, then the database retrieves yet another theory, RR, via the assertion

revision\_rules(RR).

These are clauses defining a three-place predicate "revise" which, using the information obtained from the proof of the unacceptability of the proposed assertion, revises the database. This could range from simple rejection of the proposed update to serious reason maintenance activities. The records necessary for reason maintenance can be represented as theories and recorded in the database using the same method of assertions.

Many natural insertion constraints can be expressed, such as:

```

all [person, amt, amt0] :
    unacceptable(salary(person, amt), DB))
    <--
    demo(DB, salary(person, amt0), _)
    & amt0 ≠ amt.

or

all [person, amt, amt0, Lim, percent_increase] :
    unacceptable(salary(person, amt), DB))
    <--
    demo(DB, salary(person, amt0), _)
    & demo(DB, raise_limit(Lim), _)
    & percent_increase(amt0, amt, percent_increase)
    & percent_increase > Lim.

```

Figure 4.2

Database demons can be implemented similarly by describing them in a subtheory of CurDB, and adding appropriate conditions in the appropriate clauses of process.

Note that since each database carries its own constraints, revision rules, and demons, these can also be modified as appropriate, as suggested below:

```

...
& demo(cur_DB, insert_constraints(IC), _)
& modify(IC, ..., NewIC)
& drop_from(cur_DB, insert_constraints(IC), inter_DB)
& add_to(inter_DB, insert_constraints(NewIC), new_DB) ...

```

Figure 4.3

Another interesting application shows that theories can be organized in a tree structure similar to that of many operating systems. Suppose that 'subtheory(...)' is taken to be a distinguished predicate. Let root, t1, t2, and t3 be theories containing at least the clauses indicated below:

```

root:
  subtheory(t1).
  subtheory(t2).
  parent_theory(root).
  ...
t1:
  subtheory(t3).
  parent_theory(root).
  ...
t2: ...
t3: ...

```

Moreover, assume that the clauses for the predicate `find` (from the definition of `demo/react`) contain the following clauses among their first entries (cf. Appendix B):

```

find(Goal, U/V, Rule)
:-
  find(subtheory(V), U, _),
  find(Goal, V, Rule).

find(Goal, '.'/V, Rule)
:-
  current(Theory),
  find(parent_theory(U), Theory, _),
  find(subtheory(V), U, _),
  find(Goal, V, Rule).

```

These clauses then would allow calls such as

```
demo(root/t1/t2, Goal, Proof)
```

to be effectively equivalent to

```
demo(t2, Goal, Proof).
```

As indicated above, *names* refer or point to the (syntactic item) which they name. Since they are just constants of the system, they are also first-class objects and hence can appear in assertions. Thus, one of the most common uses of names is to make assertions about clauses. This is perhaps most naturally done by recording assertions about the clauses of one theory, say `t1`, in another theory, say `t2`. Thus if we wish to attach confidence factors to the assertions of `t1`, we might do it as indicated in Figure 4.4.

t2	t1
conf( , 0.5)	assertion1
conf( , 0.3)	assertion2
...	....

Figure 4.4

Among the possible benefits of such an approach, we would list the ability to simultaneously attach differing confidence factors (using t2, t3, etc.) to the same set of assertions (t1), and the ability to modify the confidence factors associated with the assertions of t1 (by using drop\_from and add\_to to move from t2 to t2', etc.). Another application of this technique is the representation of information for a reason maintenance system. For example, in the context of the simple database manager sketched above, the basic database db might always contain an assertion

justifications(reasons)

where reasons is expected to be a theory recording the justifications for every item added to the database. The entries in reasons would be assertions about the clauses in db and about deductions (or default justifications) justifying the presence of the assertion in the database. The actual assertions in reasons would be facts whose arguments were names of clauses in the database, names of proofs, terms representing application of default rules, etc. Such information is truly meta-level knowledge about the database, and the facilities of a meta-level system such as metaProlog provide powerful and flexible means for the expression and manipulation of such meta-level knowledge.

## 5. Application to Representation of Frames

Frames (cf. Minsky [1975] and Kuipers [1975]) constitute a powerful and commonly used method for representing knowledge in AI programs. The meta-level approach advocated here provides a natural method for representing frames using theories and names, and sheds some interesting light on the issues raised by Winograd [1975]. First consider the notion of frame. In its most elementary form, a frame is rather like a record structure with named slots which can be filled with appropriate entries. For example, a portion of a room frame (suitable for a vision system) might appear:



number\_of\_walls: 4  
number\_of\_doors: 1  
color\_walls: blue  
type\_of\_floor: wood  
color\_floor: brown

Figure 5.1

From a simple logical point of view, the filling of such named slots with values amounts to making assertions about a room, and so the knowledge contained in the frame could be taken as being equivalent to a corresponding set of assertions, such as:

number\_of\_walls(room, 4).  
number\_of\_doors(room, 1).  
color\_of\_walls(room, blue).  
type\_of\_floor(room, wood).  
color\_of\_floor(room, brown).

Figure 5.2

Note that despite the slot-naming convention of the original frame, the logical equivalent could be expressed somewhat differently (and potentially more flexibly) by:

number\_of(walls\_of(room), 4).  
number\_of(doors\_of(room), 1).  
color\_of(walls\_of(room), blue).  
type\_of(floor\_of(room), wood).  
color\_of(floor\_of(room), brown).

Figure 5.3

Another alternative would be:

number\_of(walls\_of, room, 4).  
number\_of(doors\_of, room, 1).  
etc.

Figure 5.4

A system in which this frame occurred would typically have to deal with more than one concrete room, and so might associate some identifier with the frame as its identification, say <frame\_id>. Then, of course, the logical equivalents above would be formulated with 'room' replaced by <frame\_id>. In a meta-level logical system, the various logical equivalents above are each theories. Since theories are first-class objects, they also have names. Thus, an alternative to using a randomly generated frame identifier as a means of identifying the frame would be to use the theory name. Of course, in the various equivalent theories above, we would then replace 'room' by the name of the theory.

Once we have chosen to represent frames by certain logical theories, we can see an alternative to the approach sketched above. Given that any theory will automatically possess a name in the metaProlog system, it is somewhat redundant to include that name in all of the basic assertions of the theory. Alternatively, we could drop the inclusion of the frame-theory name in all of the assertions, assuming that the context (the frame-theory in which they are located) is always available. Thus the first logical representation above would become:

```
number_of_walls(4).  
number_of_doors(1).  
color_of_walls(blue).  
type_of_floor(wood).  
color_of_floor(brown).
```

Figure 5.5

And the last two versions would coalesce to:

```
number_of(walls_of, 4).  
number_of(doors_of, 1).  
color_of(walls_of, blue).  
type_of(floor_of, wood).  
color_of(floor_of, brown).
```

Figure 5.6

These approaches to representing frames provide the elementary advantages any frame system must possess. The information in the frame is physically grouped together, so that once the frame is accessed, all of this information is immediately available. Moreover, if names encode the address of the object named, then reference to the frame-theory by name provides fast access to the frame.

Further development of this representation of frames necessarily involves at least a

sketch of the development of a system for processing frames. We could certainly present such a sketch as a free-standing metaProlog program. However, we will show how such a system can be integrated with the metaProlog proof predicate demo, thus providing a built-in frame processing capability in metaProlog. This should be regarded as a design study for a metalevel Prolog which intends to provide built-in frame processing capabilities<sup>(3)</sup>. The development we will present will actually indicate how a free-standing system would be developed. We will assume that the first variation on the representation of frames by theories is being used, namely, that the frame name (i.e., the metalevel name referring to the physical manifestation of the frame) is repeated in all of the assertions recorded in the frame.

First consider the problem of retrieving a value *C*, where *C* is an uninstantiated variable, which is the color of the walls of the particular room in question. This would be tackled by solving the goal:

```
demo(db, color_of_walls(<frame name>, C), _).
```

One possibility is that a *color\_of\_walls* assertion about *<frame name>* has been directly recorded in *db*, and so the goal will be solved by *demo*. But the expected case is that the desired *color\_of\_walls* assertion has been recorded in the frame pointed at by *<frame name>*. To deal with this case, we must add the following clause to the definition of *demo/react*, before the last clause for *react* given earlier:

```
react(Theory, Goal, fr(Arg1), Continuation)
:-
  Goal =.. [Pred, Arg1 | Rest_Args],
  name_of(Arg1, Frame_Theory),
  find(Goal, Frame_Theory, Continuation).
```

Figure 5.7

The use of *find* in this clause for *react* provides for the possibility that the slot entry in the frame might *not* be a simple binary fact, but might be a rule providing a method for calculating the value of the entry, rather than simple recording of a value. Thus there will be no need to provide separately for "value-calculating" demons.

To show that this approach can lead to a realistic frame system, we must indicate how to incorporate three additional behaviors:

- inheritance
- demon processing
- frame updating

- default values for slot entries

Consider the first. Assume that we wish an inheritance hierarchy based on the predicate 'is\_a'; that is, frames may contain assertions of the form:

```
is_a(<Frame Name>, <Super Frame Name>).
```

The normal tracing of such a hierarchy can be accomplished by adding the following clause to the definition of demo/react following the first frame clause described above:

```
react(Theory, Goal, inh([Arg1 | Inh_Trace]), Continuation)
:-
  Goal =.. [Pred, Arg1 | Rest_Args],
  name_of(Arg1, Frame_Theory),
  find(is_a(Arg1, Super_Frame_Name), Arg1, _),
  name_of(Super_Frame_Name, Super_Frame_Theory),
  Super_Goal =.. [Pred, Super_Frame_Name | Rest_Args],
  react(Super_Frame_Theory, Super_Goal, Inh_Trace, Continuation).
```

Figure 5.8

Exceptions to the inheritance mechanism are accomplished by simply recording the exceptional assertion in lower frame. Thus if a particular room has five walls instead of the normal four provided by the generic room frame, the assertion

```
number_of_walls_of(<Frame_Name>, 5).
```

is simply recorded in the frame for the particular room. Its presence there overrides the inheritance mechanism.

To accomplish frame updating, we must first recognize that in the normal use of frames, the process of updating the frame over-writes the old frame, and so there is no notion of continued access to the old frame. Basically, we want to create a new theory representing the frame, but with the previous slot-value entry dropped and the new slot-value entry added. This is accomplished by adding the following clause to the definition of the definition of demo/react:

```

react(Theory, update(Frame_Name, Slot, New_Value),
      upd(Frame_Name, Slot, New_Value), true)
:-
Old_Assert =..[Slot, Frame_Name, Old_Value],
drop_from(Frame, Old_Assert, Intermed_Frame),
New_Assert =.. [Slot, Frame_Name, New_Value],
add_to(Intermed_Frame, New_Assert, New_Frame).

```

Figure 5.9

A potential difficulty lies in the fact that assertions in the database or in other frames may point to the current frame via the frame name; after updating, one must assure that these point to the new frame. But here our design decision regarding the nature of names as rigid designators comes into play. Since we actually physically update the representation of the old theory to create the new theory, and since names encode the physical address of the theory pointed at, after this update, all names which pointed at the old theory now point at the new theory.

Although the inheritance mechanism for frames provides the basic method of supplying default values for slots, it is sometimes desirable to provide such values locally in the frame itself. This is different than providing exceptions to inheritance, but proceeds by much the same mechanism, utilizing a standard method for providing default processing in Prolog. This method consists in exploiting the ordered processing of clauses under backtracking. Thus a clause which provides the default processing for a predicate *p* is provided as the last of the clauses making up procedure *p*. If new clauses for *p* are added to the database, one takes care to add them using "asserta" which always inserts the new clause before any previously recorded clauses. In our setting of frames, we use the same methodology. When a given frame *f* is initially created, if a slot *s* is to have a default value associated with it, a clause providing that value (e.g., *s*(default)) is added to *f*. So long as the entry for *s* in *f* is not updated, this clause will provide the desired default value. When the entry for *s* in *f* is updated, the mechanism described above simply replaces the default clause with the new entry. Thenceforth, the newly recorded value is provided. If it is desirable that the default value clause remains in the frame along with the newly recorded entry, a variant *update\_a* can be used which records the entry before any preceding entries, without deleting the existing entry. The processing for *update\_a* is similar to that for *update* above; it can simply record the value, or it can be designed to determine whether one or two entries for the slot have been recorded, and in the latter case, drop the first entry from the frame. In this way, only the default and most recently recorded entries would be preserved in the frame.

Finally, demon processing can be achieved by a variety of techniques. Perhaps the most direct is to assume that the frame contains rules whose heads are of the form

```
demon(<slot>, <old_value>, <new_value>, <frame>)
```

The intent of these rules is to provide procedural attachments to the particular slot, which may in fact affect the final state of the frame. Such processing can be invoked by modifying the clauses of demo defining frame processing. Thus, to provide for "on-update" demon processing, the clause defining frame updating would be modified to read:

```
react(Theory, update(Frame_Name, Slot, New_Value),
      upd(Frame_Name, Slot, New_Value), true)
:-
  Old_Assert =..[Slot, Frame_Name, Old_Value],
  drop_from(Frame, Old_Assert, Intermed_Frame_0),
  New_Assert =.. [Slot, Frame_Name, New_Value],
  add_to(Intermed_Frame, New_Assert, Intermed_Frame_1),
  demo(Intermed_Frame_1,
       demon(Slot, Old_Value, New_Value, Intermed_Frame_1), _).
```

Figure 5.10

Note that since the demon has access to the frame via *Intermed\_Frame\_1*, it can modify other slot values via updates. The power of these techniques lies in the treatment of theories and names as first-class objects.

## 6. Application to semantic nets and scripts

Since semantic nets and scripts have a high similarity to frames within inheritance hierarchies, it should be evident that we can easily capture these techniques in the present setting. Consider, for example, the simple fragment of a net shown in Figure 6.1 (drawn from Winston [1984], p. 262):

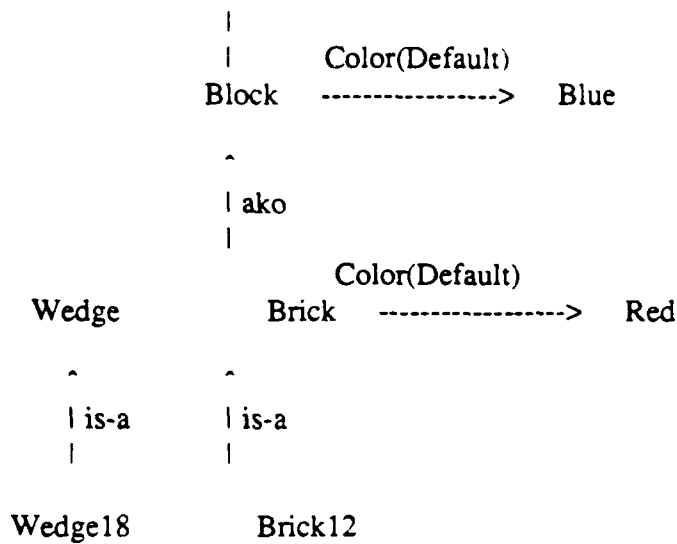


Figure 6.1

Each node can be represented by a small theory which contains an assertion

label(<node name>).

together with assertions representing the links to other nodes. Thus for example, the node for Brick would be a small theory containing the assertions:

```

label(Brick).
ako(<Name of Block node>).
default_color(<Name of Red node>).
  
```

Since the pointers to other nodes are metalevel names encoding the address of the theory pointed at, they really are pointers in the same sense as LISP pointers used in many implementations of semantic nets. Inference, say via spreading activation, can be programmed in a manner similar to more usual implementations. Note that since the nodes of the net are theories, they can contain a rich structure other than the simple representation of connections as indicated above. Besides providing for the possibilities of procedural attachments in a manner similar to that for frames described earlier, it would be possible for a given node to contain an entire separate semantic network inside of it, or to contain a rich metaProlog theory which could be made use of in procedural or declarative ways.

The representation of scripts can be flexibly accomplished using a combination of the techniques we have described for frames and semantic nets. Thus, for example, the scenes of a script can be represented as theories, as could the episodes of a scene, while the episodes could be represented as lists of individual acts, with special terms representing alternative branches to other episodes. Since terms (including lists) are constructed via pointers (as in LISP), the lists representing episodes can be so con-

structured that each episode is represented only once, though the lexical representation below does not show this. Consider the RESTAURANT script from pp. 43-44 of Schank and Abelson [1977]. The basic script is represented by a theory as follows:

```
Script(RESTAURANT).
Track(shop(coffee)).
Props([Tables, Menu, Food, Check, Money]).
Roles([S,W,C,M,O]).
Scene(1, Entering, <Name of Scene 1 Theory>).
Scene(2, Ordering, <Name of Scene 2 Theory>).
Scene(3, Eating, <Name of Scene 3 Theory>).
Scene(4, Exiting, <Name of Scene 4 Theory>).
Entry_conditions([ [S,is,hungry], [S,has,money] ]).
Results([ [S,has,less,money], [O,has,more,money],
[S,is,not,hungry], optional([S,is,pleased])]).
```

Figure 6.2

A minor alternative would be to represent the Entry conditions and Results by small theories, with assertions being used instead of the lists as above; e.g., has(S, money) instead of [S,has,money]. Scene 1, that of Entering, could be represented by the following theory:

```
scene_name(Entering).
episode(only, [ [S,PTRANS,S,into,restaurant],
[S,ATTEND,eyes,to,tables],
[S,MBUILD,where,to,sit],
[S,PTRANS,S,to,table],
[S,MOVE,S,to,sitting,position]
| next([scene(<Name of Scene 2 Theory>))] ]).
```

Figure 6.3

The predicate episode describes the entry episodes for the scene; in this case, there was only one. The end of an episode is signaled by by encountering a term

next([<destination>, <destination>, ...])

which indicates alternative following episodes within this script, or indicating exits from this scene to other scenes. As indicated, destinations can be represented by terms such as scene(...) whose argument is a theory name, or episode(...) whose entry is a (pointer to) a list, or 'exit', which would indicate the exit from the script. Note that episode pointers can be pointers to the beginning of an episode list or a pointer into the middle of such a list. In the presentation of the representation of Scene 2 below, we have used lower case labels to indicate the addresses of terms and have enclosed them in angle brackets when they are used in next terms.



```

scene_name(Ordering).
episode([menu,on,table], <e1>).
episode([W,brings,menu], <e1>).
episode([S,asks,for,menu], <e2>).

e1: [ [S,PTRANS,menu,to,S] | next([episode(<e4>)]) ]
e2: [ [S,MTRANS,signal,to,W],
      [W,PTRANS,W,to,table],
      [S,MTRANS,'need menu',to,W],
      [W,PTRANS,W,to,menu] | next([episode(<e3>)]) ]
e3: [ [W,PTRANS,W,to,table],
      [W,ATRANS,menu,to,S] | next([episode(<e4>)]) ]
e4: [ [S,MTRANS,food,list,to,CP(S)],
e8:  [S,MBUILD,choice,of,F],
      [S,MTRANS,signal,to,W],
      [W,PTRANS,W,to,table],
      [S,MTRANS,'I want F',to,W] | next([episode(<e5>)]) ]
e5: [ [W,PTRANS,W,to,C],
      [W,MTRANS,[ATRANS,F],to,W] | next([episode(<e6>),episode(<e7>)]) ]
e6: [ [C,MTRANS,'no F',to,W],
      [W,PTRANS,W,to,S],
      [W,MTRANS,'no F',to,S] | next([episode(<e8>),
                                     scene(<Scene 4 Theory Name>,[no,pay])]) ]
e7: [ [C,do,script(prepare,F)] | next([scene(<Scene 3 Theory Name>)]) ]

```

Figure 6.4

The remaining two scenes from this script are treated similarly.

## 7. Message passing and object-oriented programming

In its most basic form, object-oriented programming views computation as consisting of the passing of messages between entities called "objects" which, beyond the capacity to send and receive messages, possess local procedures whereby they can do work, including deciding what to do on receipt of a message and what further messages to send. Since theories contain procedures and can refer to other theories via names, it is natural to represent objects as theories. We can easily accomplish message passing by adding clauses to demo as above for frames. Let

```
send(<theory>, <message>, <response>)
```

be a distinguished predicate. Consider the following additional clause for the definition of demo/react:

```

react(Theory, send(Destination_Theory, Message, Response),
      send(Destination_Theory, Message, Response, Trace), true)
:-
demo(Destination_Theory, receive(Message, Response), Trace).

```

Figure 7.1

If `Destination_Theory` contains clauses defining the predicate 'receive', this clause for `demo` achieves the desired message passing. Note that this technique also captures the partially instantiated message idea of Concurrent Prolog (Shapiro [1983]), since both `Message` and `Response` could be partially instantiated terms.

In many applications of message passing, it is desirable to view certain objects (here, theories) as databases or blackboards accumulating assertions supplied by other objects. We can achieve this by adding a variant notion of sending a message. Let

```

assert(Theory, Assertion, Response)

```

be a distinguished predicate, and add the following clause to the definition of `demo/react`:

```

react(Theory, assert(Database_Theory, Assertion, Response),
      assert(Database_Theory, Assertion, Response, Trace), true)
:-
demo(Database_Theory,
      process(add(Assertion, Response),
              Database_Theory, New_Database_Theory), Trace).

```

Figure 7.2

Here it is assumed that `Database_Theory` contains clauses defining process similar to those described earlier in connection with the database manager example. Note that by varying the clauses defining process, different database theories can achieve different methods of processing assertions.

Finally, let us note that this logical construal of message passing provides an alternate, more logical method of construing input and output in this system. Simply, the user's terminal as well as files are viewed as theories to which messages are sent and from which responses are received. For example, if 'user' is a distinguished theory name, the process of writing can be viewed as simply sending a message to user and ignoring the response:

```
send(user, <output item>, _).
```

On the other hand, reading from the terminal can be seen as sending a message requesting a read, and expecting the item read to be supplied as the response:

```
send(user, read, Read_Item).
```

Note that as noted earlier, the response slot could be filled with a partially instantiated term indicating the nature of the item to be read, or the message slot could indicate more specifically what is to be read, as for example:

```
send(user, read(clause), Clause_To_Be_Read).
```

## 8. Metalevel control

The default control mechanism of *metaProlog* is that of ordinary *Prolog*: depth-first, left-to-right, backchaining exploration of the search space. There are, however, occasions on which some programmers would find it advantageous to be able to obtain and exploit other control mechanisms. In particular, to achieve the proper behavior for the message passing paradigm described in the previous section, it is necessary to have fair processing of the send and receive requests among the objects. A number of suggestions have been set forth over the years for extending the control of logic programming systems, and our own thoughts have touched on a number of schemes. We believe this is an area which will undergo considerable change and evolution in the future, since no one approach has yet become dominant. What we have done in the present setting is to provide one fairly flexible method for adding alternate control mechanisms, with the hope that it will provide a testing ground for alternative approaches. Examination of the two clauses defining *demo* proper indicate that *Goal* appears only as a variable -- that the *demo* clauses have no knowledge of the structure of Goals. In fact, the only two predicates which must have knowledge of the structure of goals are 'select' and 'merge' (as well as 'empty'). Consequently, it will be possible to package goals in manners that can possibly indicate to 'select' and 'merge' that non-standard control is to be utilized. Thus, instead of packaging goals *G1*, *G2*,... simply as *G1 & G2 & ...*, one might use a compound structure such as

```
notation(G1 & G2 & ...)
```

to indicate to 'select' some non-standard desire. Note that *notation* could also carry additional arguments beyond the actual goal. The meaning of some such notations could be built into 'select' (and 'merge'). On the other hand, if 'select' and 'merge' are given *Theory* as an extra argument, then they can also look for clauses defining the

non-standard control notation, much in the manner of Pereira [1984]. The predicate 'find' could also be treated similarly. The modified forms of the relevant clauses of demo/react together with fragments of the clauses necessary for 'select', 'merge', and 'find' are presented in Appendix B.

As an example, consider the circuit diagnosis problem sketched in Bowen and Weinberg [1985]. To follow Eshgi's [1982] approach, one finds it desirable to operate under a control regime which defers selecting subgoals of the form 'andGate(...)' or 'orGate(...)' until as late as possible. Under the facilities described above (cf. Appendix B), this could be accomplished by including the following clauses in the theory (C & D & P) which describes the circuit under diagnosis:

```
all [Gs, G, RGs] :
user_select(defer(defer(Gs)), G, RGs)
  <--
  user_select(defer(Gs), G, RGs).

all [Gs, G, RGs] :
user_select(defer( true & Gs ), G, RGs)
  <--
  user_select(Gs, G, RGs).

all [A, B, SG, RB] :
user_select(defer( A & B ), SG, defer( A & RB ) )
  <--
  table(A)
  & user_select(defer( B ), SG, RB).

all [A1, A2, B, SG, RA] :
user_select(defer( (A1 & A2) & B , SG, defer( RA & B ) )
  <--
  user_select(defer( A1 & A2 ), SG, RA).

all [A, B, Op] :
user_select( defer( A & B ), A, defer( B ))
  <--
  functor(A, Op, _)
  & Op ≠ '&'.

all [A, Op]:
user_select( defer(A), A, true)
  <--
  functor(A, Op, _)
  & Op ≠ '&'.
```

## 9. Conclusions

We have described one approach to creating a metalevel extension of Prolog which treats theories and names (of theories) as objects capable of being the values of variables, fully on a par with ordinary terms. The preceding sections have illustrated the tremendous power inherent in these ideas, showing that many of the common devices used for knowledge representation in AI can be cleanly captured in such an extension. It is likely that these metalevel techniques can be used in new and previously unforeseen ways to represent knowledge. The ideas considered in this paper raise many interesting theoretical and practical problems. To explore these problems together with the questions of applications to AI, our research group is actively exploring the construction of efficient interpreters and compilers for such metalevel systems. Through these efforts as well as the related research mentioned in Section 2, we expect to see exciting developments in this area in the near future.

## 10. Bibliography

Bowen, K.A., and Kowalski, R.A., *Amalgamating language and metalanguage in logic programming*, in **Logic Programming**, eds Clark, K.L. and Tärnlund, S.-A., Academic Press, London and New York, 1982, pp. 153-172.

Bowen, K.A., and Weinberg, T., *A meta-level extension of Prolog*, in **Proceedings of the 1985 Symposium on Logic Programming**, eds Cohen, J., and Conery, J., IEEE Computer Society Press, Washington, D.C., 1985, pp. 48-53.

Church, A. *A formulation of the simple theory of types*. **The Journal of Symbolic Logic**, 5 (1940), pp 56-68.

\_\_\_\_\_, *A formulation of the logic of sense and denotation*, in **Structure, Method, and Meaning (Essays in Honor of Henry M. Sheffer)**, New York, 1951, pp. 3-24.

Eshgi, K. *Application of meta-language programming to fault finding in logic circuits*, in **Proceedings of the First International Logic Programming Conference**, ed van Caneghen, M., Marseille, 1982, pp. 240-246.

Furukawa, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M., and Ueda, K., *MANDALA: A logic based knowledge programming system*, in **Proceedings of the International Conference on Fifth Generation Computer Systems**, ed ICOT, ICOT, 1984, pp. 613-622.

Gödel, K., *Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I*, **Monatshefte für Mathematik und Physik** 38, 1931, pp. 173-198.

[English translation in: **From Frege to Gödel**, ed van Heijenoort, J., Harvard Universi-

ty Press, Cambridge, Mass., 1967, pp. 596-616.]

Kitakami, H., Kunifuji, S., Miyachi, T., and Furukawa, T., *A methodology for implementation of a knowledge acquisition system*, in **Proceedings of the 1984 International Symposium on Logic Programming**, IEEE Computer Society Press, Washington, D.C., 1984, pp. 131-142.

Kripke, S.A., *Naming and necessity*, in **Semantics of Natural Language**, eds Davidson, D. and Harman, G., D. Reidel Pub. Co., Dordrecht, Holland, 1972, pp. 253-355.

Kuipers, B.J., *A frame for frames: representing knowledge for recognition*, in **Representation and Understanding**, eds Bobrow, D.G., and Collins, A., Academic Press, New York, 1975, pp. 151-184.

Minsky, M., *A framework for representing knowledge*, in **The Psychology of Computer Vision**, ed Winston, P.H., McGraw-Hill, New York, 1975, pp. 211-277.

Miyachi, T., Kunifuji, S., Kitakami, H., Furukawa, K., Takeuchi, A., and Yokota, H., *A knowledge assimilation method for logic databases*, in **Proceedings of the 1984 International Symposium on Logic Programming**, IEEE Computer Society Press, Washington, D.C., 1984, pp. 118-130.

Nakashima, H., *Prolog/KR - language features*, in **Proceedings of the First International Logic Programming Conference**, ed van Caneghen, M., Faculte des Sciences de Luminy, Marseille, 1982, pp. 65-70.

\_\_\_\_\_, *Knowledge representation in Prolog/KR*, in **Proceedings of the 1984 International Symposium on Logic Programming**, IEEE Computer Society Press, Washington, D.C., 1984, pp. 126-130.

Pereira, L., *Logic control with logic*, in **Implementations of Prolog**, ed Cambell, J., Ellis Horwood, Chichester, England, 1984, pp. 177-193.

Shapiro, E., *A subset of concurrent Prolog and its interpreter*, Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1984.

Shoenfield, J., **Mathematical Logic**, Addison-Wesley, Reading, Mass., 1967.

Warren, D.H.D., *An abstract Prolog instruction set*, Technical Report 309, Artificial Intelligence Center, SRI International, 1983.

Winograd, T., *Frame representations and the declarative-procedural controversy*, in **Representation and Understanding**, eds Bobrow, D.G., and Collins, A., Academic Press, New York, 1975, pp. 184-210.

## 11. Appendix A: Definition of demo without control

Here we have collected together all of the clauses for demo/react which do not account for metalevel control (for the latter, see Appendix B).

```

demo(Theory, Goal, [])
:-
  empty(Goal).

demo(Theory, Goal, [Reason | Rest_Proof])
:-
  select(Goal, SubGoal, Rest_Goals),
  react(Theory, SubGoal, Reason, Continuation_Goals),
  merge(Continuation_Goals, Rest_Goals, New_Goal),
  demo(Theory, New_Goal, Rest_Proof).

react(Theory, demo(New_Theory, Subsid_Goal, Subsid_Proof),
       sbs(Subsid_Proof), true)
:-
  demo(New_Theory, Subsid_Goal, Subsid_Proof).

react(Theory, current(Theory), current(Theory), true).

react(Theory, Goal, fr(Frame_Trace), Continuation)
:-
  Goal =.. [Pred, Arg1 | Rest_Args],
  name_of(Arg1, Frame_Theory),
  find(Goal, Frame_Theory, Continuation).

react(Theory, Goal, inh([Arg1 | Inh_Trace]), Continuation)
:-
  Goal =.. [Pred, Arg1 | Rest_Args],
  name_of(Arg1, Frame_Theory),
  find(is_a(Arg1, Super_Frame_Name), Arg1, _),
  name_of(Super_Frame_Name, Super_Frame_Theory),
  Super_Goal =.. [Pred, Super_Frame_Name | Rest_Args],
  react(Super_Frame_Theory, Super_Goal, Inh_Trace, Continuation).

react(Theory, update(Frame_Name, Slot, New_Value),
       upd(Frame_Name, Slot, New_Value), true)
:-
  Old_Assert =.. [Slot, Frame_Name, Old_Value],
  drop_from(Frame, Old_Assert, Intermed_Frame),
  New_Assert =.. [Slot, Frame_Name, New_Value],
  add_to(Intermed_Frame, New_Assert, New_Frame).

```

/\* Modified form to use to allow demon processing on update: similar modification should be made to other frame axioms if demon processing is desired there; e.g., on access, or on inheritance, etc.



```

react(Theory, update(Frame_Name, Slot, New_Value),
      upd(Frame_Name, Slot, New_Value), true)
:-
Old_Assert =..[Slot, Frame_Name, Old_Value],
drop_from(Frame, Old_Assert, Intermed_Frame_0),
New_Assert =.. [Slot, Frame_Name, New_Value],
add_to(Intermed_Frame, New_Assert, Intermed_Frame_1),
demo(Intermed_Frame_1,
      demon(Slot, Old_Value, New_Value, Intermed_Frame_1), _).
*/

react(Theory, send(Destination_Theory, Message, Response),
      send(Destination_Theory, Message, Response), true)
:-
demo(Destination_Theory, receive(Message, Response), _).

react(Theory, assert(Database_Theory, Assertion, Response),
      assert(Database_Theory, Assertion, Response), true)
:-
demo(Database_Theory,
      process(add(Assertion, Response),
              Database_Theory, New_Database_Theory), _).

react(Theory, SubGoal, s(SubGoal, Rule), Rule_Body)
:-
find(SubGoal, Theory, Rule),
parts(Rule, Rule_Head, Rule_Body),
match(SubGoal, Rule_Head).

```

## 12. Appendix B: Definition of demo with metalevel control

```

demo(Theory, Goal, [])
:-
  empty(Goal).

demo(Theory, Goal, [Reason | Rest_Proof])
:-
  select(Goal, SubGoal, Rest_Goals, Theory),
  react(Theory, SubGoal, Reason, Continuation_Goals),
  merge(Continuation_Goals, Rest_Goals, New_Goal, Theory),
  demo(Theory, New_Goal, Rest_Proof).

/* react clauses ... */

react(Theory, SubGoal, s(SubGoal, Rule), Rule_Body)
:-
  find(SubGoal, Theory, Rule),
  parts(Rule, Rule_Head, Rule_Body),
  match(SubGoal, Rule_Head).

select( SubGoal & SubGoals, SubGoal, SubGoals, _).

/* special built-in select control clauses */

select(Goal, SubGoal, Rest_Goals, Theory)
:-
  demo(Theory, user_select(Goal, SubGoal, Rest_Goals), _).

select(Goal, Goal, true, _).

find(Goal, T1 & T2, Rule)
:-
  find(Theory, T1, Rule).

find(Goal, T1 & T2, Rule)
:-
  find(Goal, T2, Rule).

find(Goal, U/V, Rule)
:-
  demo(U, subtheory(V), _),
  find(Goal, V, Rule).

find(Goal, '..'/V, Rule)
:-
  current(Theory),
  find(parent_theory(U), Theory, _).

```

```
demo(subtheory(V), U, _),  
find(Goal, V, Rule).
```

```
/* Other built-in specialized find rules and normal (Prolog) retrieval */
```

```
find(Goal, Theory, Rule)  
:-  
demo(Theory, user_find(Goal, Theory, Rule), _).
```

### 13. Footnotes

(1) There appears to be a viable alternative to the introduction of metalevel names which we have introduced here. Roughly, this approach regards *all* entities, including what are normally regarded as formulas, as terms, much in the manner of Church's systems of type theory and sense and denotation (Church [1940], [1951]). A subclass of these terms are contextually distinguished as formulas. Since formulas in reality are terms, they may *directly* appear in other formulas without the mediation of metalevel names. The details of this approach remain to be worked out.

(2) For those with knowledge of Prolog implementation methods following Warren [1983], we can indicate some of the details of how this is to be carried out. The basic representation of theories consists of a predicate-name hash table the entries of which can point to various kinds of buckets. The table and the buckets are allocated on the heap (global stack). [Alternatively, the code space can be managed as a separate (garbage-collectable) heap similar to the global stack used for terms.] Also, the trail mechanism is modified so as to allow recording of values other than undef to which a variable must be reset upon backtracking. (Several methods are available for this; on byte-addressable machines, one is particularly efficient.) Let T1 be a variable which has previously been instantiated to a theory, let T2 be an uninstantiated variable, and suppose the call

```
... & add_to(T1, p(8), T2) & ...
```

is to be executed. All the events which are to take place are trailed. If no clauses for p appear in T1, an entry for p is made in the hash table for T1. An entry for p(8) is made in the appropriate bucket. T2 is set to point to the hash table pointed at by T1. And finally, T1 is reset to point to a description based on (the variable) T2 and the trail entries; this description enables the system to recover the changes made when clauses from the original T1 are requested (though nothing is really reset unless backtracking occurs). T1 can be thought of as a virtual modified copy of T2. It is important that the description for T1 be in terms of the variable T2, instead of the actual physical representation of the updated theory, since further changes to the theory might otherwise invalidate that description.

(3) A powerful alternative to this approach would be to design the basic demo/react interpreter in such a way as to allow the user to add clauses for demo and react in his own theory; in this setting, they could be clauses implementing a particular style of frame processing. In essence, this would provide a powerful form of *logical reflection* (cf. Bowen and Kowalski [1982]). This line of investigation is being pursued by our research group. It appears to be quite straight-forward to provide such power in an interpreter approach to demo/react, but complications arise when one attempts to understand the nature of compilation in this situation.

It should also be noted that the notion of inheritance for frames which will be developed can be generalized to a notion of inheritance applicable to theories in general. Properly done, the frame notion is then simply a special case of the more general notion.

# Fast Decompile of Compiled Prolog Clauses

Kevin A. Buettner

Logic Programming Research Group  
School of Computer & Information Science  
Syracuse University

## Introduction

The programming language *Prolog* has a number of builtin predicates for operating on the global database. Among these operations, the predicates *assert*, *retract*, and *clause* are found in most Prolog systems. In the logic programming community, these predicates have gained a certain notoriety. In spite of the bad press that these operations have gotten, many practical and useful programs are written which need and require these operations although in theory it is possible to dispense with them.<sup>†</sup> The view of the author is that these operations are important and should be provided in modern Prolog systems. Even languages which claim a sounder logical foundation with respect to the database operations<sup>‡</sup> have implementation problems similar to those found in ordinary Prolog.

The first Prolog environments were interpreter based. The database operations are relatively easy to implement in these environments due to the similarity between the run-time data structures and the code structures. At the present, the trend seems to be towards compiler-based Prolog environments. Many if not most of the compiler-based implementations have their roots in the abstract machine of David Warren [1983]. With the advent of this compiler-based technology, implementation of the database operations has become more challenging due to the fact that the run-time data structures are quite a bit different from the code structures. The run-time data structures are much the same as they were under interpreter-based implementations, but the code structures are different; they are now sequences of instructions to execute. Implementation of the *assert* operation with this new technology isn't too difficult. All one needs to do is invoke the compiler on the clause to assert, obtain the sequence of instructions and link these into the indexing scheme.

It is the implementation of either *retract* or *clause* that is more interesting. These operations are similar in that we must be able to take fairly arbitrary patterns and find a clause in the database which matches these patterns. Again for interpreter-based systems, this isn't usually very hard due to the similarity between the run-time structures and the code structures. But matching clauses in a compiler-based system is more difficult. As observed by Clocksin [1985] in his discussion of the implementation of Prolog-X, there are basically three ways to attempt the matching:

- 1) Saving a copy of the clause in the run-time data structure format.
- 2) Decompile of clause code to obtain the original structure.
- 3) In addition to compiling the clause, compile the clause's source which is used explicitly for matching purposes. So in normal execution, one version of the clause is used; but when the clause is to be matched, the other version which will return the source structure is run instead.

<sup>†</sup> It is often possible to reorganize programs so that *assert* and *retract* are not used, but at the expense of efficiency and quite often clarity of expression.

<sup>‡</sup> e.g. *metaProlog* (cf. Bowen and Weinberg [1985]) provides the operations *addTo* and *dropFrom* which are used to create new theories from old ones.

Clocksin's Prolog-X system uses the third method, but has the disadvantage that essentially two copies of the clause are saved in the database. Fortunately, there is a way of using Warren's abstract instruction set which allows us to obtain the original clause structure from the object code with little effort. Furthermore, very few limitations exist on the types of compiler optimizations that may be performed. The limitations that do exist, however, may be easily circumvented with no loss in efficiency. In the ensuing discussion of the decompilation technique, familiarity with the abstract machine described by Warren [1983] is assumed.

### Decompilation

Suppose we have a source clause of the following form:

$$H :- G_1, \dots, G_m.$$

This translates (schematically) to:

```

match args of H
set up args of G1
call g1/n1.ES1
set up args of G2
call g2/n1.ES2
.
.
.
set up args of Gm
execute gm

```

In the above, call and execute are actual instructions. *match* represents the sequence of get and unify instructions that perform the head matching. *set up* refers to a sequence of put and unify instructions that install arguments for the call or execute instructions. In practice, the initial *match* and *set up* instructions are often merged as an optimization. This assumption will not affect the process which we are about to describe.

Before discussing the nitty gritty details, it should be noted that a fair amount of structure is created by the object code in normal execution. For example, if the clause is run with all uninstantiated arguments, by the time of the first call the original variables will be instantiated to structure representing the arguments in the head of the clause. Similarly, just before a call or execute, the A registers contain the arguments of the goal about to be called. To get back the structure of the entire clause entails taking these pieces and wrapping them up with the appropriate functors that represent the head, goals, commas between the goals, and the ":" between the head and the goals. The decompilation technique will be discussed in two stages. The first stage is to look at an object code transformation that when run by the underlying Prolog engine will give back the original clause structure. The second stage is rather easy; ways to avoid doing the actual transformation are considered.

A transformation that will take object code representing a clause (in the above form) and produce object code which can be run with a single argument (which may have varying levels of instantiation) and return the structure (or pieces of the structure as the level of instantiation requires) will now be described. The idea is to tack on a prologue to the beginning of the code which will create the uninstantiated variables for the head and set up the clause predicate name. Basically the prologue will create something of the form

$$h(\text{Arg}_1, \text{Arg}_2, \dots, \text{Arg}_{n_0}) :- G$$

The A registers will have pointers to the variables  $\text{Arg}_1, \dots, \text{Arg}_{n_0}$  respectively. Because the A registers contain pointers to variables, the get instructions in the *match* section of the clause code will be forced into write mode and will create structure. We don't want the call and execute instructions to run, so we replace them with code which will unload the argument registers and produce the appropriate goal structures.

Schematically, the transformation looks like this:

Original Code	Transformed Code	Comments
	<pre> get_structure :- 2, A1 unify_variable As unify_variable At get_structure h/n0,As unify_variable A1 . . unify_variable An0                     </pre>	Prologue
<pre> match args of H set up args of G1                     </pre>	<pre> match args of H set up args of G1                     </pre>	
<pre> call g1/n1,ES1                     </pre>	<pre> get_structure '1/2, At unify_variable As unify_variable At get_structure g1/n1,As unify_local_value A1 . . unify_local_value An1                     </pre>	Call Transformation
<pre> . . set up args of Gm                     </pre>	<pre> . . set up args of Gm                     </pre>	
<pre> execute gm                     </pre>	<pre> get_structure gm/nm,At unify_local_value A1 . . unify_local_value Anm proceed                     </pre>	Execute Transformation

Notes:

- The As and At in the above transformation refer to unused temporary (argument) registers.
- unify\_local\_value instructions are used to insure that pointers in structure built on the heap also point to objects on the heap. This is necessary due to the fact that an argument register may have a pointer to a variable on the local stack. If a unify\_value instruction is used in this situation, a pointer from the heap to the local stack is installed causing a dangling reference when the environment is deallocated.

The above transformation scheme will work for clauses with at least one goal where both the head and each of the goals has at least one argument. As it stands, it will not work for unit clauses or for clauses which have nullary goals. These cases, however, are not difficult to handle. For unit clauses, the prologue changes slightly. When it is necessary to work with a nullary goal, a get\_constant instruction is used instead of a get\_structure instruction. For example, the prologue for a unit clause whose head has no arguments is:

```
get_constant h,A1
```

The prologue for a unit clause with one or more arguments is

```

get_structure h/n0,A1
unify_variable A1
.
.
unify_variable An0

```

An execute instruction with no arguments translates to

```

get_constant gm/nm,At
proceed

```

It is left as an exercise for the reader to fill in the other variations. It is possible to define the translations so that there are no variations between unit clauses and rules, but in practice these variations cause little difficulty. The translation procedure given here has the added advantage that in attempting to match (unmatchable) partially instantiated clauses, failure can occur quite early. To make the translation procedure more uniform, it would be necessary to delay building the structure for the :- until the execute instruction. It would also be necessary to translate proceed instructions, which are currently untouched.

As an example, consider the clauses that make up part of a popular benchmark:

```

nrev([],[]).
nrev([H|T],L) :- nrev(T,RT), conc(RT,[H],L).

```

This compiles to the following object code:

```

nrev/2:
      switch_on_term      L434,
                          L440,
                          L464,
                          fail
L434:  try_me_else        L458,2 %
L440:  get_nil           A1      % nrev([],
      get_nil           A2      %   [])
      proceed          % .

L458:  trust_me_else     fail,2  %
L464:  allocate          %
      get_list          A1      % nrev(
      unify_variable    Y1      %   H |
      unify_variable    A1      %   T],
      get_variable      Y2, A2  %   L) :-
      put_variable      Y3, A2  % nrev(T,RT)
      call              nrev/2,3 % ,
      put_unsafe_value  Y3, A1  % conc(RT,
      put_list          A2      %   [
      unify_value       Y1      %   H |
      unify_nil         %       [],
      put_value         Y2, A3  %   L)
      deallocate        %
      execute           conc/3  % .

```

Performing the transformation on the first clause gives:

```

get_structure      nrev/2,A1
unify_variable     A1
unify_variable     A2
get_nil           A1      % nrev([],

```



```

get_nil      A2      %
proceed     %

```

The instructions of the code new to the clause are italicized. The switch and try\_me\_else instructions were not included because they form part of the procedure nrev/2 rather than the first clause of nrev/2. The second clause for the nrev procedure transforms to:

```

get_structure  nrev/2,A1
unify_variable A4
unify_variable A5
get_structure  nrev/2,A4
unify_variable A1
unify_variable A2
allocate      %
get_list      A1      % nrev(
unify_variable Y1      % H |
unify_variable A1      % T,
get_variable  Y2, A2   % L) -
put_variable  Y3, A2   % nrev(T,RT)
get_structure  nrev/2,A5
unify_variable A4
unify_variable A5
get_structure  nrev/2,A4
unify_local_value A1
unify_local_value A2
put_unsafe_value Y3, A1 % conc(RT,
put_list      A2      % [
unify_value   Y1      % H |
unify_nil     % [],
put_value     Y2, A3  % L)
deallocate    %
get_structure  conc/3,A5
unify_local_value A1
unify_local_value A2
unify_local_value A3
proceed

```

The reader should think of each of these transformations above as a unary unit clause. A1 should point to the structure to be matched or a variable. In the latter case, when the proceed instruction is reached, the variable will be instantiated to the clause structure. Of course, the original variable names will be missing; these may be filled in if desired by a number of different methods†.

In a real implementation, it will be undesirable to perform the actual translation. What should be done instead is to run the code for the prologue elsewhere, jump to the start of the clause code and interpret the call and execute instructions in a different manner. This interpretation of the call and execute instructions can be realized in at least two ways.

The first way of reinterpreting the call and execute instructions is to add another mode bit to the machine. This may in fact be worthwhile since *clause* and *retract* are, in some programs,

† If it is important to fill in the original variable names, another clause `vname(DBRef,VNameList)` may be asserted. DBRef is a reference to the clause in the database. VNameList is a list of the variables (as atoms) that occur in the clause from left to right with no duplications. To reinstall the variable names, the clause structure obtained from the decompilation process should be traversed from left to right. Every time a *hole* (variable) is found in the structure, it is filled in with the first element in the variable name list. After filling a hole, the first element of the list is discarded and the rest of the list is used for the remainder of the traversal.

quite heavily used. In one mode, the call and execute instructions behave normally; in the other mode, the sequence of *get* and *unify* instructions is performed. To set the mode bit, it may be desirable to create a new instruction which will also perform code for the prologue and branch to the start of the clause.

The second method involves setting breakpoints on the call and execute instructions. The break routine is responsible for performing the sequence of *get* and *unify* instructions corresponding to the call or execute instruction and for setting the next breakpoint if any. The very first breakpoint is set by the code that does the prologue. This second method is quite appropriate for a software implementation of the Prolog engine; implementing mode bits is quite expensive in software. On the other hand the first method, described in the preceding paragraph, may be more suitable for a hardware implementation of the Prolog engine.

In the C-based Prolog system written at Syracuse University, the second technique is used. The prologue is actually implemented as part of a builtin which returns the clause structure of a given clause.† This builtin also sets a breakpoint on the first call (or execute) instruction, if any, and sets the P register (program counter) to the start of the clause to decompile. When the clause is done "executing", it has decompiled itself.

### Limitations

The limitations of this method have to do with the implementations of `=/2`, `var/1`, `non-var/1`, and similar builtins.

Some compilers emit the following code for `=/2`:

```
put argument one, A1
put argument two, A2
get_value A1,A2
```

Provided that the `get_value` instruction doesn't fail, the above method described will work fine; the problem is that the resulting structure won't always be the same as the original structure. The meaning of the two clauses will be the same, but syntactically (modulo variable names), they won't be. The reason for this is that the transformation fails to take into account the fact that `get_value` is used in place of a call to the equality predicate. If the `get_value` instruction is replaced by a call to `=/2` or perhaps an instruction that performs the same function as `get_value A1,A2` then the decompilation method will work.

Even worse†, some compilers transform clauses with `=/2` in them. For example,

```
p(X,f(g(X),Y)) :- X=h(Y).
```

may be transformed to

```
p(h(Y),f(g(h(Y)),Y)).
```

Again, the decompilation procedure will work, but probably not as expected.

Another problem results in the way that `var/1` is implemented in some compilers.

```
p(c,X) :- var(X).
```

may be implemented as.

---

† The builtin is called `clause_structure(ProcName,Arity,DBRef,Struct)` where `ProcName` and `Arity` are the predicate name and arity of the clause referenced by `DBRef`. `Struct` is usually an output argument and represents the source structure of the clause. An additional builtin is provided for obtaining an initial data base reference to the first clause of a procedure given the module, predicate name, and arity. Another builtin is used to find the next clause in the indexing scheme, failing when it finds no more. With these builtins, it is possible to implement `clause/2`, `clause/3`, and `listing/1`. Additional builtins designed for removing references and inserting new ones are provided for implementing `retract` and `assert`.

† Or perhaps better depending on your point of view.

	get_constant	c.A1
	put_value	A2.A1
	switch_on_term	L1.fail.fail.fail
L1:	proceed	

Again, the problem is that a transformation isn't defined for the `switch_on_term` instruction. It would be possible to define a transformation, but unwise since this same instruction may be used to implement `nonvar` also. A better approach is to create instructions which implement the `var` and `nonvar` operations and define the obvious transformations on them for the decompilation process.

Similar problems exist (in some compilers) for other operations (usually builtins). Either by making explicit calls or by defining new machine instructions, these problems can usually be rectified.

### Conclusions

The methods described in this paper have applications beyond implementation of `clause`, `retract`, and `listing`. The method of setting breakpoints may be used to implement debuggers (in particular, the standard four-port debugger). In a nutshell: breakpoints are set on the next `call`, `execute` or `proceed` instruction and the next failure address (obtained from the top choice point). When the breakpoint is executed, the appropriate `call`, `redo`, `fail`, or `exit` message is printed. `Redo` and `fail` messages are printed when the failure breakpoint was reached. One or more `exit` messages are printed when a breakpoint corresponding to a `proceed` instruction is executed. `Call` messages are printed on `call` and `execute` instructions. Because of the generalized tail recursion optimizations in the Warren architecture, it is necessary to maintain debug frames. These frames contain such information as the previous debug frame, the parent debug frame, the call structure and whether the instruction that caused this frame to be created was an `execute` or a `call`. These frames may be safely kept on the heap; in fact keeping the frames on the heap permits a clever implementation of deciding how many `redo` messages to print when a failure occurs.

Both the decompilation and debugging methods have been implemented in a system constructed at Syracuse University. The underlying compiler is very fast, incremental, and has an interactive interface. All of the flexibility of an interpreted system is achieved in this compiler-based system. Moreover there are no interface or portability problems as are often found in systems which require both an interpreter and a compiler.

## References

- Bowen, K. A. and Weinberg, T. A Meta-level Extension of Prolog. *1985 Symposium on Logic Programming*. pp. Boston, IEEE, pp. 48-53, 1985.
- Clocksin, W. F. Implementation Techniques for Prolog Databases. *Software—Practice and Experience*, 15(7). pp. 669-75, 1985.
- Warren D. H. D., An Abstract Prolog Instruction Set. Technical Note 309. Artificial Intelligence Center, SRI International, 1983.

# The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler

Kenneth A. Bowen, Kevin A. Buettner, Ilyas Cicekli, Andrew Turk

Logic Programming Research Group  
School of Computer & Information Science  
Syracuse University  
Syracuse, NY, 13210 USA  
CSNET: kabowen@csyr

## Abstract

The design and implementation of a relatively portable Prolog compiler achieving 12K LIPS on the standard benchmark is described. The compiler is incremental and uses decompilation to implement retract, clause, and listing, as well as support the needs of its four-port debugger. The system supports modules, garbage collection, database pointers, and a full range of built-ins.

## 1. Introduction

In the course of exploring implementation techniques for metalevel extensions of Prolog (cf. Bowen and Kowalski [1982], Bowen and Weinberg [1985], Bowen [1985]), it became apparent that a fast flexible Prolog compiler would be a useful tool to serve as a starting point for developing experimental implementations of the extended systems. Consequently, in late 1984 we began exploring just such a project. We planned to base the system on the designs of Warren [1983], implementing a byte-code interpreter for the abstract machine in C, while implementing the compiler itself in Prolog. We worked initially in C-Prolog on the Data General MV/8000 which was the machine available to us at that time. We were fortunate to join forces with the group working at Argonne National Laboratory (Tim Lindholm, Rusty Lusk, and Ross Overbeek) who were interested in the implementation of Prolog on multiprocessor machines. They had already implemented a byte-code interpreter for a system which would support multiple versions of Warren's abstract Prolog machine (WAM), different machines running on different processors, but using shared physical memory and implementing appropriate logical memory spaces. The system was parameterized as to the

---

This work supported in part by US Air Force grant AFOSR-82-0292 and by US Air Force contract F30602-81-C-0189. The authors are very grateful to the following people for numerous valuable conversations on the topics of this paper: Hamud Bacha, Aida Batarekh, Jim Kajjya, Kevin Larue, Jacob Levy, Tim Lindholm, Rusty Lusk, Jon Mills, Hidey Nakashima, Ross Overbeek, Karl Puder, and Toby Weinberg.

number of physical processors, so that we could run a version with that parameter set to one, yielding a sequential byte-code interpreter for the abstract machine. Thus, in principle, we could focus our efforts on the construction of the compiler. Naturally, life being what it is was not quite that simple. The Argonne group had implemented their byte-code interpreter in C on a VAX 750. While they had striven for portability, one serious hardware assumption had crept into the code, namely that the underlying machine was byte-addressable. Since the MV/8000 is not a byte-addressable machine, we found that we had to devote considerable energy to porting the Argonne WAM to the MV/8000. However, the changes necessary to achieve this port were propagated back into the original Argonne code, so that the present Argonne WAM is in all likelihood an extremely portable system. The Argonne system includes a "WAM assembler" which will assemble and load "WAM assembly code". (This was revised by Cicekli to remove limitations on the sizes of programs which could be assembled and loaded.) Thus we were able to hand-compile and run test examples. We were disappointed in the resulting performance, the naive reverse benchmark (nrev) performing at only about 4K LIPs. We concluded that the relatively slow speed was due to a combination of the portability requirements and the data structures necessary for multi-processor implementation (even though we were making no use of those facilities)! Performance improved somewhat when we moved to a newly acquired VAX 780 running Berkeley UNIX 4.2, but was still disappointing. This disappointment, coupled with an interest in implementing a Prolog system on 68000-based machines, led Turk to begin exploring a new implementation of a byte-code interpreter written in C, while as a group we continued work on the compiler.

The need to devote resources to the port to the MV/8000 had slowed our development of the compiler, so it was not until late February of 1985 that we had a first version of the compiler itself constructed and operational in C-Prolog. While writing the compiler in Prolog was of course a joy, we found ourselves somewhat hampered by C-Prolog's restricted memory size and apparent lack of significant tail recursion optimization and garbage collection. Consequently, we were forced to somewhat unnaturally segment parts of the compiler, store intermediate results in files, etc. The compiler itself had grown fairly large, reflecting our explorations of various optimization techniques. When we began to attempt to boot the compiler on itself, we were frustrated to discover that we immediately overran the maximum allowable local and global stack spaces. While we found that by a combination of breaking the compiler into many small files and using Prolog assert/retract hacks to reclaim stack space we could begin jamming it through, we were quite upset by the butchery this was performing on what we originally regarded as relatively clean code. At this time, Buettner had been devoting some time to exploring the implementation of a Prolog compiler on 16-bit machines, in particular the design of a byte-code interpreter for that environment. In a burst of enthusiasm, he roughed out a new byte-code interpreter for the abstract machine coupled with an implementation of a moderately sophisticated compiler, all written in C, in the space of a month. We now found ourselves in the (perhaps enviable) position of possessing three distinct implementa-

tions of the abstract machine (all written in C) and two compilers, one written in C and the other in Prolog.

While there were some differences in structure between the compilers, they both operated on basically the same principles. On the other hand, our two home-grown implementations of the abstract machine appeared to use significantly different techniques, and of course differed markedly from the Argonne implementation. Since both of our local WAMs executed *nrev* at better than 6K LIPS and both authors asserted that not all opportunities for optimization had been exploited, we decided to pursue development of both machines and compilers in parallel. In the course of the summer of 1985, we saw both machines evolve towards a more common structure, and begin achieving speeds in nearing 10K LIPS on *nrev*. We also had the interesting experience of booting the Prolog-based version of the compiler using the C-based Prolog compiler. We were able to do this without introducing any of the ugly adjustments we had found necessary when using C-Prolog. Since the two abstract machines seemed to be evolving towards a common structure, we decided in July (at a breakfast meeting at the Logic Programming Symposium) to coalesce the two efforts, making a final incorporation of the remaining clever techniques of Turk's machine into Buettner's. From that point on, we focused most of our efforts on developing the C-based Prolog compiler and abstract machine. We did complete the Prolog-based version of the compiler and delivered a copy to the Argonne group in late August. It is expected that this version will be made publicly available along with the Argonne WAM sometime in the near future. The rest of this paper will be devoted to describing the design, structure, and facilities of the C-based system.

## 2. Organization of the System

We will assume familiarity with Byrd, Pereira and Warren [1980], Pereira, Pereira, and Warren [1978], and Warren [1983]. To the user, our system presents the appearance of a standard Edinburgh-style interactive interpreter. However, it is really an incremental compiler. Thus we have no need to support a separate interpreter with all the difficulties of consistency between compiler and interpreter which are normally entailed. Briefly, the major services provided by the system are as follows:

- The compiler is resident in the system, incrementally compiling original and added program clauses (including those added by *assert*) as well as goals.
- Programs may be organized into modules which are relatively independent of file structure in that multiple modules may be included in a single file (a single module can also be spread over several files); visibility of procedures is controlled by use of *import/export* declarations; clauses not appearing within a module declaration are stored in a default global module; constants and functors are globally visible; modules may appear as submodules within oth-

er modules:

- Garbage compaction of the global stack (heap) and trail is provided using a pointer-reversal algorithm of Morris[1975]; no garbage collection is provided for the code space:
- Run-time use of retract, clause, and listing is accomplished via a general decompilation technology (described in detail in Buettner [1985]); this technology is also used to support the debugging subsystem:
- A full four-port debugging model (cf. Byrd[1980]) is provided; it relies on the decompilation technology mentioned above and accomplishes its task by constructing linked lists representing local stack frame entry and exit on the global stack (heap); it is largely complete, though some standard commands remain to be implemented:
- Database pointers are supported; these exist as Prolog terms which can occur in other terms and predicates:

The system supports the full range of built-ins standard in Edinburgh-style Prolog systems. Some are implemented in C, with the rest being written in Prolog and compiled by the system.

The system occupies approximately 135K bytes of virtual memory (and 76K bytes of physical memory) when loaded. Performance of the system on the naive reverse benchmark is shown in Table 2.1 (measured in LIPS) for lists of length 100 and 1000. The slower figures for lists of length 1000 of course reflects the need to perform garbage collection.

	Unoptimized	Optimized
100	9.6K	12.0K
1000	8.5K	10.5K

Table 2.1. Benchmark performance.

The "unoptimized" column represents the performance of the system running with the output of the UNIX 4.2 C compiler unchanged. The "optimized" column represents the performance of the system with the output of the C compiler slightly hand optimized. The only optimization specific to a Prolog system is a tightening of the dereference loop. All of the rest of the optimizations are of a generic sort that could be performed by a highly optimizing C compiler, such as shortening branches to branches (to branches...). Another such optimization involves reclaiming poorly used machine registers. In the compiler output, the low numbered machine registers are only used for scratch values and are not saved on procedure entry/exit. The usage of these registers was reorganized and



code added before calls and exits to render them safe. Most of these optimizations were performed on the code simulating abstract machine instructions. A native code compiler could get it right from the beginning, while of course performing many other optimizations. It would not surprise us to see a speed increase factor of 3-4 resulting from native code compilation.

### 3. Compiler Organization

While the principles on which the two compilers operate are quite similar, their internal organization is somewhat different.

#### 3.2 Clause Compilation

The overall action of the Prolog-based compiler is divided into three major passes:

- (1) compilation of individual clauses to intermediate code.
- (2) organization of groups of intermediate clause code into procedures, and
- (3) generation of instructions for the abstract machine.

During the first pass, the compiler treats each clause for a procedure separately, producing intermediate code representing the action of that clause. This pass is organized into three phases: lexical analysis, parsing, and intermediate clause code generation. The lexical analysis phase outputs a list of annotated tokens. The parsing phase processes this list, more or less in a definite clause grammar style, to produce a complex Prolog term representing the clause; a considerable amount of variable analysis is also performed during this phase. The third phase processes this term, producing another Prolog term representing the required sequence of abstract machine instructions. Considerable use of difference lists and uninstantiated logical variables representing machine addresses is made during these phases. During the second pass, the intermediate code for the individual clauses constituting a procedure is connected using the indexing instructions. Our method of indexing, which differs from Warren [1983], will be described later. The output of the second pass is a complex Prolog term representing the procedure. Consequently, assembly amounts to a traversal of this term, calculating symbolic addresses as necessary, and linearizing the entire structure; loading is then straight-forward.

The C-based version of the compiler utilizes a standard Prolog reader to read the clauses as terms. It makes one pass through the term, performing its variable analysis and building appropriate tables. On a second pass through the term, this compiler generates and loads the instructions for the clause, linking them into the naive try-me-else indexing chain for the procedure (see Section 3.2). Full indexing for the procedure is generated when the module containing the procedure is sealed.

Examination of the examples supplied in Warren [1983] shows that the required *get*- and *put*- instructions occur in the order corresponding to the left-to-right ordering of the corresponding terms in the source clause. In an effort to minimize the number of instructions generated and to optimize A-register usage, our compilers reorder these instructions. They also make a very serious attempt to set up the arguments to the first call in the body while carrying out the head matching. They also perform the now-standard Warren-style optimization of permanent variable allocation by trimming environments. (The permanent variables used in implementing cut are also included in this optimization -- cf. Section 4.)

### 3.2. Indexing

Access to the block of clauses constituting a procedure is handled in the usual way with hash tables, though provision for modules and hiding of local procedures complicates this a bit. Within the list of clauses constituting a procedure, it is desirable to minimize the number of clauses attempted but failed due to failure to match the head of the selected clause against the incoming goal. Such a failure can occur only when the incoming goal contains instantiated variables; if all variables of the incoming goal are uninstantiated, the goal will match the head of each clause of the given procedure. Consequently, the indexing process has two major tasks to accomplish:

- (a) When the incoming goal contains uninstantiated variables in designated indexing argument places, it must provide a means of trying each clause of the procedure in order.
- (b) When the incoming goal contains instantiated terms in the argument places designated for indexing, it must provide a means of selecting only those clauses whose heads satisfy the following: for each argument position designated for indexing, the term occurring in the clause head must match the term occurring in the corresponding position of the incoming goal.

As with all other current Prolog systems known to us, ours only supports (or designates) indexing on the first argument of procedures. (However, our plans for the future include relaxing this restriction). We have not modified the indexing instructions of Warren [1983], but we do employ them in a different manner. Focusing on the first argument of procedures, a *block* of clauses is a maximal subset of the clauses for a procedure, contiguous in the given clause ordering, all of whose first head arguments are of the same type, where the allowable types are:

constant, compound term (other than list), variable, and list.

Roughly, one uses indexing instructions at the lowest level to control access to each block, coupling these with second-level indexing instructions to control transfers between blocks. A sequence of instructions of the form

try - retry - ... - retry - trust -

specifies a group of clauses to be tried in sequence, as does a sequence of the form

```
try_me_else - ... retry_me_else - ... retry_me_else - ... trust_me_else fail.
```

In the second case, the branch instructions must be physically interleaved with the code of the individual clauses, while in the first, the collection of branch instructions can be physically quite removed from the code of the clauses controlled. We refer to these as *try chains* and *try\_me\_else chains*, respectively. Note that in a *try\_me\_else* chain, the label of each instruction is the address of the succeeding *retry\_me\_else* or *trust* instruction. Consequently, this succeeding instruction and its following clause code need not physically follow the code of the preceding clause. Consequently, we can regard a *try\_me\_else* chain as a linked list of clauses. In the case of *try chains*, while the actual *try-retry-trust* instructions must physically follow one another (they constitute a vector of instructions), the actual code blocks of the clauses they control can be distributed in memory in any manner whatsoever. These code blocks need bear no physical relationship to one another nor to the controlling *try chain*, other than the fact that the *try chain* instructions reference the addresses of the clause code blocks. We exploit both of these observations in the implementation of *assert* and *retract*. Our method of indexing runs as follows. To cater to requirement (a) above, we create one master *try\_me\_else* chain linking all of the clauses of the procedure. In catering to requirement (b), we avoid using the *...\_me\_else* instructions, restricting ourselves to *try-retry-trust* to control sequential access to both clauses and blocks. Constant and compound term blocks are of course accessed using *switch* instructions, and overall access to the upper-level indexing is initiated with the *switch\_on\_term* instruction. Sequential ordering of groups of clauses as well as groups of blocks of clauses is indicated with *try chains*; no use of *try\_me\_else* chains is made in the upper-level indexing meeting requirement (b). Consequently, the indexing meeting requirement (a) is totally separated from the indexing meeting requirement (b). We feel this provides great flexibility for insertion and deletion of clauses (by *assert/retract* or by a run-time editor) while minimizing the number of choice points which must be created. Figures 3.1 and 3.2 schematically indicate the structure of this scheme.

#### 4. Abstract Machine Organization and Cut

The layout of the various machine regions is shown in Figure 4.1.

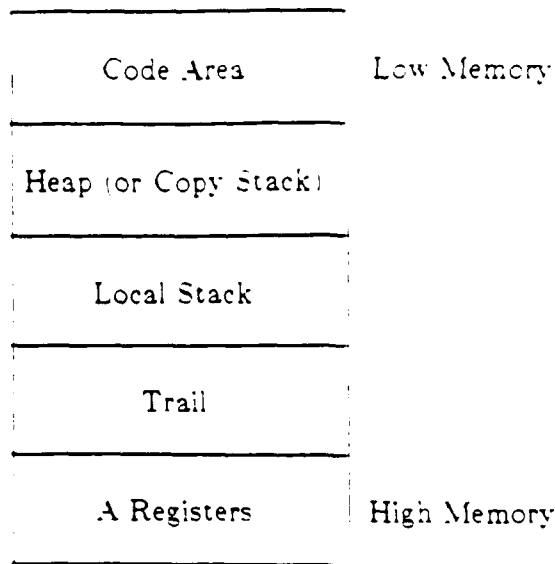


Figure 4.1. Abstract Machine Organization.

A bit table for garbage collection is also permanently allocated. For the most part, we have implemented the instruction set of Warren [1983] with only minor modifications. The most significant extension to date is the addition of a new machine register (called *cutpt*) and new instructions to allow us to compile *cut*. These instructions and their effects are listed below:

Instruction	Action
set_B_from_cutpt	B := cutpt
set_B_from Yn	B := Yn
save_cutpt_in Yn	Yn := cutpt
save_B_in Yn	Yn := B

Figure 4.2. Instructions Necessary for Cut.

The last instruction is only necessary for compiling the so-called "soft cut".

The difficulty in dealing with *cut* is that at compile time, it is impossible to know how many choice points will be created for a procedure before a clause of that procedure is entered. Consider the following trivial program.

f(a).  
f(b).

f/1:    switch\_on\_term    C1a.L1.fail.fail

L1:	switch_on_constant	2.[a:C1. b:C2]	
C1a:	try_me_else	C2a	f/1
C1:	get_constant	a..A0	a)
	proceed		.
C2a:	trust_me_else	fail	f/1
C2:	get_constant	b..A0	b)
	proceed		.

When the first clause C1 is executed, there can be one or zero choice points for the procedure f/1, but this cannot be detected at compile time because it depends on the incoming value in the first argument register A0. If the incoming value in A0 is the constant a, there will be no choice point created for the procedure f/1, but if a is an unbound variable, there will be one choice point created for the procedure f/1.

The new register *cutpt* is treated in the abstract machine as follows. The value of the last choice point register B is automatically stored in the *cutpt* register by a *call* or an *execute* instruction to record the address of the last choice point before the procedure is invoked. The current value of the *cutpt* register is saved in a choice point when the latter is created. The *cutpt* register is reset from the value stored in the last choice point when backtracking occurs.

The following examples illustrate how the compiler uses these instructions to compile cuts.

*Example 1*

p :- q1. !. q2.

*Code for the clause:*

allocate	1
save_cutpt_in	Y0
call	q1/0.1
set_B_from	Y0
deallocate	
execute	q2/0

*Example 2.*

p :- !.

*Code for the clause*

```
set_B_from_cutpt  
proceed
```

Notice that the clause doesn't have an environment and that the `cutpt` register contains a pointer to the last choice point before the procedure `p` is invoked.

*Example 3*

```
p :- q1, q2, !.
```

*Code for the clause:*

```
allocate          1  
save_cutpt_in    Y0  
call              q1 .1  
call              q2 .1  
set_B_from       Y0  
deallocate  
proceed
```

This approach can be optimized.

## 5. Conclusions

The abstract machine design of Warren [1983] together with the compilation techniques suggested by his examples are a sound piece of software engineering. We have filled in some gaps such as the implementation of `cut` which were omitted in his discussion, and have introduced modifications in the pursuit of refining and optimizing performance. The present system provides an excellent basis for our primary goal, the pursuit of implementations of meta-level Prolog systems. Our approach will be to introduce modifications to the abstract machine providing the required functionality, the primary one being a change in the treatment of the code space. This will be coupled with appropriate changes in the compilers. We expect this to lead to efficient implementations of the experimental systems.

## 6. References

Bowen, K.A., and Kowalski, R.A., Amalgamating language and metalanguage in logic programming, in *Logic Programming*, ed. K. Clark and S.-A. Tarnlund, 1982, pp 153-172.

Bowen, K.A., and Weinberg, T., A meta-level extension of Prolog, *1985 Symposium on Logic Programming*, Boston, IEEE, 1985, pp. 48-53.

Bowen, K.A., Meta-Level programming and knowledge representation, *New Generation Computing*, 3, 1985, pp. 359-383.

Buettner, K.A., Decompilation of compiler Prolog clauses, submitted.

Byrd, L., Prolog debugging facilities, in Byrd, Pereira, and Warren, 1980.

Byrd, L., Pereira, F., and Warren, D., *A Guide to Version 3 of DEC-10 PROLOG*, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1980.

Morris, F.L., A time- and space-efficient garbage collection algorithm, *Communications of the ACM*, 21, (1978), pp. 662-665.

Pereira, L.M., Pereira, F.C., and Warren, D.H.D., *User's Guide to DECsystem-10 PROLOG*, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1978.

Warren, D.H.D., An abstract Prolog instruction set, SRI Technical Report, 1983.

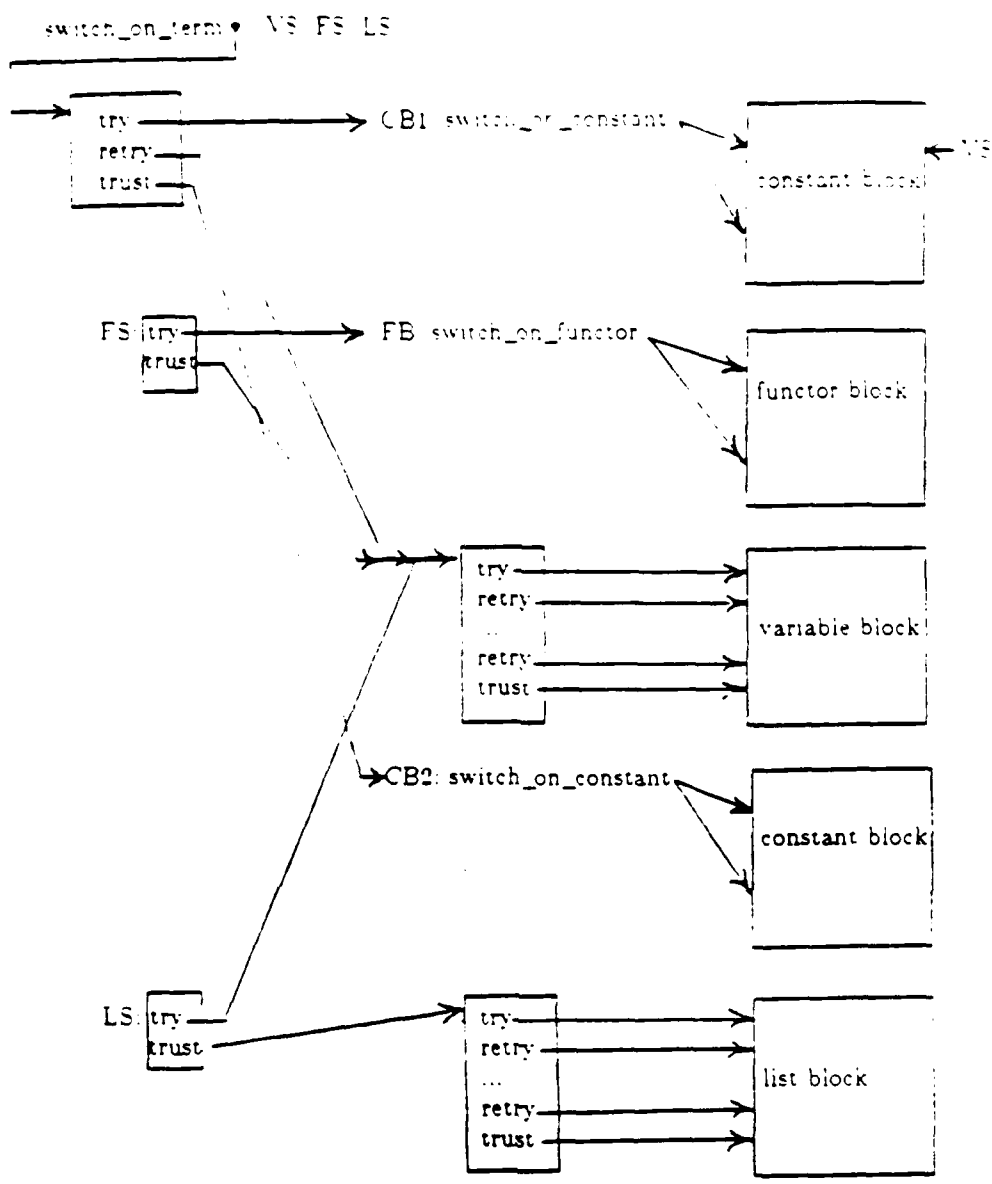


Figure 3.1. Overall indexing structure.



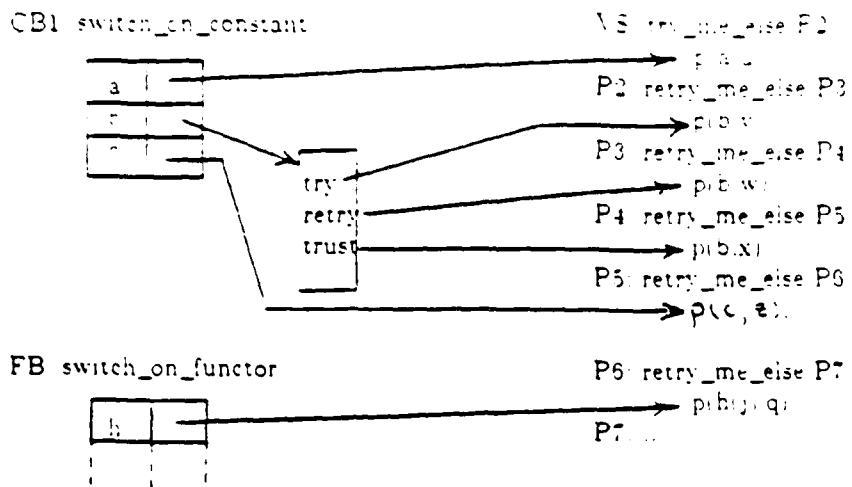


Figure 3.2. Detail of indexing structure.

# Compiler Optimizations for the WAM

Andrew K. Turk

Logic Programming Research Group  
School of Computer & Information Science  
Syracuse University  
Syracuse, NY, 13210 USA

## Abstract

A series of Warren Abstract Machine (WAM) implementation techniques are presented. These techniques and compilation strategies are designed for use in a highly optimized native code Prolog compiler. A thorough knowledge of the WAM and Prolog compilation is assumed.

### 1. Motivations

On conventional hardware, it is necessary to compile Prolog to native machine code for optimal performance. Ideally, a native code compiler can work along side one of the many byte-code compilers which already exist. This paper outlines a series of optimizations which can be used by an intelligent compiler to translate WAM byte-codes into the native machine language of a conventional machine. Most of these strategies need not be applied to every clause and procedure; the compiler is relatively free to decide which method is best, or at least take advice from the programmer. It is assumed that the optimizations will only be applied to static code.

### 2. An Overview of Native Code

WAM byte-codes can be naively translated into native code in a straightforward fashion. Each of the steps and conditionals which must be performed by the byte-code interpreter must also be executed by native code. When these steps are directly translated into a native code stream, the overhead inherent in the interpreter is eliminated. However, much more can be done than the simple elimination of the interpreter.

### 3. The Read/Write Mode Bit

Warren's original design incorporates a mode bit which tells the machine whether it's in read mode or write mode. Many hardware and software interpreters actually use a mode bit. However, mode bits are expensive and difficult to manage in a native code system. Fortunately, there is a straightforward method which eliminates the bit and retains its functionality.

---

This work supported in part by US Air Force contract number F30602-81-C-0169

The mode bit is set by *get* instructions and must be maintained through the execution of the following *unify* instructions. Instructions which change the mode (either *gets* or *puts*) cannot occur between a *get* and its *unifys*. The simplest way to eliminate the mode bit is to compile each *unify* sequence twice, once for write mode and again for read mode. First, the code to dereference the register in question is emitted. Following that is a conditional branch to the read mode sequence (which directly follows the write mode sequence). Right after the conditional is the first instruction of the write mode sequence. A branch around the read mode code is placed after the last write mode instruction.

No "continuation branch" is needed after the read mode code because the next instruction corresponds to rest of the clause. *Put* instructions force the machine into write mode. This is done only so that the *unify* instructions will work properly. Since the mode is always write after a *put*, there is no need to compile a separate sequence for read mode. Some software implementations achieve the same effect with a set of *unify* instructions that always work in write mode. The compiler should emit the write mode sequence first in order to make write mode propagation more effective.

#### 4. Write Mode propagation

Compiled constants and variables do not change the mode. However, when an unbound variable is passed into a clause at runtime, a *getStruct* or *getList* will force the machine into write mode. Not only will that particular *getStruct* or *getList* invoke write mode, but any substructure corresponding to that instruction must also run in write mode. This is because all of the *unifyVar* instructions in the original *unify* sequence run in write mode, and therefore create fresh unbound variables on the heap.

When write mode is propagated to a *getStruct F, An* or a *getList An*, the machine can infer that *An* contains a reference to an unbound variable on the heap. Because of this, the runtime dereference and tag check for *undef* can be skipped. This can amount to a significant savings for programs which spend a lot of time in head matching.

Unfortunately, there is no way to propagate read mode. Furthermore, without undue overhead, the write mode of an instruction can only be propagated to its "leftmost" subtree. This is due to the fact that there is no way to distinguish between a propagated write mode and a non-propagated write mode.

In order to implement this technique, the compiler takes advantage of the fact that the *unify* sequences are compiled once for write mode and once for read mode, with a branch in between. If the compiler detects that the mode can be propagated from some parent *get* instruction to a child *get*, the "continuation branch" in the parent's code will jump PAST the dereference and check for *undef*, directly into the write mode *unify* sequence of the child. If the mode for the child can be propagated, then its continuation branch will skip the dereference and check for *undef* of the next *get*, and so on.

Whenever the mode cannot be propagated, the parent's "continuation branch" will

enter the child's *get* at the beginning. In order to capitalize on this optimization, the byte-code should match head structure in long left-descending chains so that the mode can be propagated as far as possible.

## 5. Shallow Backtracking on the WAM

Other Prolog implementations distinguish between two different types of backtracking: deep backtracking and shallow backtracking. Shallow backtracking occurs when the current choicepoint is the topmost object on the local stack; everything else is deep backtracking. However, we will restrict the definition of shallow backtracking to cases where some clause has failed in head matching and another clause remains to be tried in the same procedure.

Most WAM implementations have a single global subroutine or label which resets the machine and argument registers after a failure. This routine is used in the compilation of a *get* instructions when an incoming argument doesn't match. However, in order to optimize for shallow backtracking, the native code compiler should compile these actions in-line along with every *retry* and *trust* instruction.

Seen in this light, each *retry* resets the argument registers, resets the machine registers, updates a field in the choicepoint, unwinds the trail and finally jumps to the next clause. Each part of the *retry* has an entry point, and if no argument registers were changed by the previous clause, the machine can simply jump past that part of the *retry*, just as if the mode were being propagated.

Suppose that a procedure contains two clauses, numbered 0 and 1. Failure in the head of clause 0 will always cause shallow backtracking. Each instruction in the head of clause 0 which might cause failure must have some way of causing the machine to fail (i.e., branching to a failure label). Furthermore, the compiler will know how many registers might have been modified in clause 0 prior to each instruction.

Thus, the *trust* separating clause 0 from clause 1 will be arranged so that the first register modified in clause 0 is the last one reset in the *trust*. The second register modified will be the second to last reset, and so on.

By doing this, the compiler can have the first possibly failing instruction in clause 0 jump past almost all of the compiled *trust* because only a few registers have been changed. The last possibly failing instruction in clause 0 will fail into the beginning of the *trust* in order to reset all the changed registers.

Since the compiler calculates where to jump during shallow backtracking, the *nextClause* field in the choicepoint record is only used for deep backtracking. Clause 1 is the last clause in the example procedure, and it can only cause deep backtracking. The compiler cannot calculate the failure label in this case, and must emit code to jump to the failure label stored in the choicepoint. Failure labels in choicepoints always point at the first instruction of a compiled *retry* or *trust* in order to reset all the machine registers.

## 6. Improved Choicepoints

Many procedures are compiled with a type of indexing which uses two *try* instructions. These procedures are composed of a series of blocks of clauses with a *try* for each block. In addition, there is a *try* which laces all the blocks together. Under certain circumstances the machine can detect at run-time that only one clause in a particular clause can possibly match. In this case, the inner *try* instruction is not executed and the second choicepoint is not created.

Since the creation of a choicepoint is a very expensive operation in terms of both space and time, the compiler should strive to avoid unnecessary choicepoints. A closer examination of the second choicepoint occasionally used in the indexing scheme shows that all its fields except for B and the address of the next clause are duplicated from the first choicepoint. In fact, the only interesting part is the address of the next clause because the B field simply points to the first choicepoint.

Two choicepoints can be collapsed into one if the next clause field of the hypothetical second choicepoint is included in the first. In particular, the two fields are allocated together in the first choicepoint and function like a small stack. A *try* instruction pushes an address on the stack, a *retry* changes the topmost entry and a *trust* pops the stack by one.

Since present compilation technology never requires more than two choicepoints per procedure, this sub-stack can be manipulated directly as an array. A *try* simply copies the first field into the second and a *trust* copies the second into the first. Sophisticated indexing techniques which require more than two choices may become available.

In this case, it will be advantageous to have another register which points to the topmost stack entry in the most recent choicepoint.

The only restriction is that the compiler must be able to know which *try* instruction will be executed first and which *trust* will be executed last. This is because the first *try* needs to allocate the choicepoint, while the last *trust* will deallocate it.

This technique avoids a great deal of space and time overhead by collapsing multiple choicepoints into a slightly larger initial choicepoint. More importantly, it means that the compiler can calculate exactly how much local stack space will be needed by the procedure at compile time. This allows it to allocate other objects below a procedure's choicepoint, such as an environment, and to share these objects between clauses.

## 7. Avoiding Environment Allocation

An environment is required for any clause that has more than one subgoal. The environment must be allocated before any use of a permanent variable and before the call to the first subgoal. Since most implementations do not have a top of stack (TOS) register, the TOS must be calculated dynamically. This involves a conditional test and possibly an indirect load and an addition. Many byte-code compilers emit code so that

the allocate instructions occur as late as possible in the hopes that the clause will not match and the environment need not be allocated.

However, procedures that use at least one *try* in their indexing always calculate the TOS in order to put a choicepoint on the stack. Thus, after a choicepoint has been laid down, the B register contains the TOS. The compiler can recognize this and compile references to permanent variables as offsets from B instead of E. Once the head of the clause has successfully matched, CP and E are copied into the local stack and E is updated to reflect the new environment.

This technique cannot be applied to procedures which optionally use zero or one choicepoint. Obviously, if no choicepoint was pushed on the stack, then B does not contain the TOS. In this case, there is a good chance that the procedure will be determinate anyway, and the compiler should not be overly concerned about optimizing for failure.

## 8. Improved Argument Registers

There are two drawbacks in the way argument registers are reset after failure in the WAM. First, many registers will be reloaded even though they were never modified in the first place. One solution to this problem is to make use of shallow backtracking. However, many registers are still reloaded from the choicepoint even though they will never be referenced due to early failure in the head. Fortunately, this problem can also be minimized by a careful compiler.

This technique involves reloading argument registers from the choicepoint of a non-determinate procedure only when absolutely necessary. Normally, the effective address of an argument register is either a host machine register or an offset into the argument array. However, the compiler can change the effective address of the first top-level occurrence of an argument register in the code to be an offset into the current choicepoint.

Thus, the compiled *retry* instructions will not include code to restore certain argument registers. The first effective address of such a top-level argument register corresponds to the stored value of that register. The machine should not change the contents of the choicepoint, but it is free to read whatever is necessary. This amounts to treating the choicepoint as a read-only cache of saved argument registers.

Two potential difficulties arise. Some of the instructions which are optimized away in byte-code (e.g., *getVar X1, A1*) cannot be eliminated with this scheme. Since the failure code will not reload certain argument registers, the compiler must explicitly restore all such registers that appear in the clause code.

Furthermore, *trust* instructions delete the topmost choicepoint and in the process remove the initial copies of the argument registers. To eliminate this problem, the compiler must emit code which only deletes a choicepoint after the head of a clause has matched or only after the head has failed. In the latter case, the machine would delete

the topmost choicepoint and then jump to the deep backtrack label of the previous choice.

A careful combination of this method and shallow backtracking should be more effective than either method alone. Register usage between clauses and the "determinateness" of the procedure will influence how the compiler emits code to reset the state after shallow backtracking.

## 9. Backtrackable Assignment

This next technique is not an optimization, nor is it unique to native code prolog implementations, but it was developed along with the other methods in this paper.

Backtrackable assignment appears to be a useful device for a number of different applications. It requires that extra information be stored on the trail for certain entries: those that were assigned to. The extra information is a copy of the particular cell before it was destructively assigned.

Naive implementations of backtrackable assignment either place a tag bit on each trail entry, or store a reset value with each reset address. The former requires that the tag bit be checked on each trail entry during failure and the latter doubles the size of the trail. Fortunately, a simple method exists which pays no overhead for normal trail entries.

The idea is to interleave two separate stacks, one for normal trail entries and one for reset-to-value entries. Each stack will have a pointer to the topmost element. TR points at the topmost normal trail entry, while TR' points at the topmost reset-to-value pair. This is similar to the way choicepoints and environments are interleaved on the local stack.

Each time a normal unification is trailed, an address is pushed onto the trail and TR is incremented. During a destructive assignment, a pointer to the cell, the contents of the cell, and the previous TR' are pushed on the trail with TR being incremented by. TR' is updated to reflect the new entry.

When backtracking requires cells to be reset, the machine compares the copy of TR in the backtrack frame with TR'; the higher of these two is copied into a temporary location Temp. The part of the trail between Temp and the current contents of TR represents a contiguous block of normal trail entries. The machine can loop through these, decrementing Temp and resetting variables to undef without checking any tags or worrying about strange objects in its path. If Temp is higher than the copy of TR in the backtrack frame, then Temp points at a reset-to-value entry which resets the variable and decrements Temp accordingly. Otherwise, no reset-to-value entries occur in the current trail segment.

This process continues by repeatedly untrailing a block of normal trail entries (possibly an empty block), and then untrailing a reset-to-value entry. When all the appropriate

trail entries have been removed, TR will point at the top of the trail and TR' will point at the most recent reset-to-value entry.

## 10. Conclusions

A number of optimized compilation techniques have been presented. An advanced compiler should be able to make use of them when it has determined that a particular procedure can be made more efficient. The optimizations are relatively independent so the compiler is not forced into a few rigid compilation models.

## 11. References

E. Tick  
Prolog Memory-Referencing Behavior,  
Research Paper 85-281, Computer Systems Laboratory,  
Stanford University, 1985.

D.H.D. Warren  
An Abstract Prolog Instruction Set  
Technical Note 309, Artificial Intelligence Center, Computer Science  
and Technology Division, SRI International, 1983.



Technical Report CIS-85-1  
School of Computer and Information Science  
Syracuse University

May 1985

**▲ Meta-Level Extension of Prolog\***

Kenneth A. Bowen  
Computer & Information Science  
313 Link Hall  
Syracuse University  
Syracuse, NY 13210  
kabowen%syrcsnet-relay

Tobias Weinberg  
AI Technology Group  
Digital Equipment Corp.  
77 Reed Road  
Hudson, MA 01749  
weinberg%logic.DEC@DECWRL

---

\* Work on this project was supported in part by AFOSR grant AFOSR-82-0292. The authors would like to thank Maarten van Emden and the students in Bowen's seminar at Syracuse University for helpful and stimulating discussions.

## I. Introduction

Prolog has many attractive features as a programming tool for artificial intelligence. These include code that is easy to understand, programs that are easy to modify, and a clear relation between its logical and procedural semantics. Moreover, it has proved possible to create clear and efficient implementations. Nonetheless, we perceive several shortcomings. Chief among these is difficulty representing dynamic databases (databases which change in time) and an apparent restriction to backward chaining, backtracking, depth-first search. Our intent in this paper is to present an extension to Prolog, called metaProlog, which preserves the virtues of Prolog while introducing powerful constructions to attack these problems. This work is a direct continuation of the investigation into meta-level programming in logic begun by Bowen and Kowalski [1982].

Many applications of artificial intelligence demand facilities which amount to the ability to dynamically manipulate databases. Databases are naturally represented in Prolog as a set of assertions and clauses. This exploits all the advantages of Prolog's inherent deductive machinery. However, the logical core of Prolog provides no conceptual basis for segmenting or modifying the database. Most implementations of Prolog have provided ad hoc extensions to the basic logic programming paradigm which allow for dynamic modification of the program database by the program itself. But since the database is the program, the use of these facilities introduces difficulties similar to those introduced by global variables and self-modifying code in conventional programming languages. The effect of these features on the virtues listed above is catastrophic. Programs become difficult to understand, reliable modification of the code is almost impossible, and the logical semantics is utterly destroyed. We know of no mathematical or philosophical definition of first-order proof where the collection of axioms is not fixed. We would suspect any such notion to be incoherent. We believe these difficulties can be overcome by the introduction of theories as first-class objects which can be dynamically created and passed as parameters. In standard Prolog, goals are invoked with respect to a single background theory. In metaProlog, goals must be proved in an explicitly identified theory. We regard this system as simply a first-order logical theory of axiom sets and proofs.

The means of indicating that a metaProlog goal  $G$  should be solved in a particular theory  $T$  is an explicit call on the proof predicate `demo`. From a logical point of view, the proof predicate is really a relation between three objects: the theory  $T$ , the goal  $G$ , and the proof  $P$  which attests to the solvability of  $G$  in  $T$ . But logic programming is not only concerned with the static existence of proofs, but also the process of discovering them. That is, it is also concerned with the notion of search space and search strategies. Thus, for logic programming, the deep central relation is the one which holds between a theory  $T$ , a goal  $G$ , and the complex object consisting of a proof for  $G$  in  $T$  seen as a portion of a search space explored by a particular search strategy. Our investigations have led us to the conclusion that all of these entities must be treated as first-class objects (metaProlog terms) capable of being manipulated and passed as values of parameters.

This approach appears to provide a logically sound programming formalism sufficiently

powerful to write clear reliable programs for experimental and applied artificial intelligence. We also believe it possible to construct efficient implementations of such a system, but will leave this question to a later paper. Although problem of efficient implementation has been of deep concern throughout our design process, our concern in this paper is with questions of conceptual and logical foundations. (Various portions of the system have been simulated by implementations in Edinburgh Prolog and parts of a prototype system have been written in C.)

Let us close this introductory section with an example illustrating the power of the approach. Suppose that one has two collections of goals,  $G_1, \dots, G_n$  and  $H_1, \dots, H_m$  and that one wishes to solve  $G_1, \dots, G_n$  in theory  $T_1$  under one search strategy  $M_1$ , and to solve  $H_1, \dots, H_m$  under another strategy  $M_2$  in theory  $T_2$ , where both  $M_2$  and  $T_2$  depend on the state of the computation resulting from the solution of  $G_1, \dots, G_n$ , as well as on  $T_1, G_1, \dots, G_n$ , and  $H_1, \dots, H_m$ . Let  $F$  be the problem to be solved by this work and let 'next\_strategy' be some procedure acting on theories, goals, and computation states which will be used to compute  $T_2$  and  $M_2$ . Then we could describe  $F$  as follows:

$F$  is solvable if

$G_1 \& \dots \& G_n$  is solvable in  $T_1$  using strategy  $M_1$   
 and  $S_1$  is the resulting computation state,  
 and next\_strategy acting on  $T_1, G_1 \& \dots \& G_n$ ,  
 $H_1 \& \dots \& H_m$ , and  $S_1$  yields  $T_2$  and  $M_2$ ,  
 and  $H_1 \& \dots \& H_m$  is solvable in  $T_2$  using strategy  $M_2$ .

Let  $v_1, \dots, v_k$  be the variables of  $G_1, \dots, G_n, H_1, \dots, H_m$ . Then, in the metaProlog formalism we will introduce below, this could be expressed by:

all  $\{v_1, \dots, v_k, T_2, M_2, S_1\}$ :  
 $F(v_1, \dots, v_k) \text{ —}$   
     demo( $T_1, G_1 \& \dots \& G_n, \text{strategy}(M_1) + \text{comp}(S_1)$ )  
     & next\_strategy( $T_1, G_1 \& \dots \& G_n,$   
                      $H_1 \& \dots \& H_m, S_1, T_2, M_2$ )  
     & demo( $T_2, H_1 \& \dots \& H_m, \text{strategy}(M_2)$ )

## II. Meta-Level Programming

It is important to make clear our notion of meta-level programming. Briefly, one distinguishes between the formal language being used to conduct some (unspecified) axiomatic investigation (the object language) and the language used to carry on any discussion about the object language (the metalanguage). For many purposes (including those of this paper), the metalanguage need only be powerful enough to discuss the combinatorial syntactic properties of the object language. The essential point is that the relations of the metalanguage are about the syntactic entities of the object language: the variables of the

metalanguage range over various syntactic entities of the object language. In contrast, the variables of the object language either have no specified range (when it is viewed as a formally uninterpreted language) or (when the object language is treated as being interpreted) range over the members (possibly extremely mathematically complex) of some specified set.

Properly viewed, an ordinary Prolog interpreter is already a meta-level object. The object level consists of a fragment of ordinary first-order logic, a language and proof predicate. The latter describes which formulas of the language are consequences of sets of other formulas of the language. The meta-level of a theorem-prover is concerned with the manipulation of sets of object-level formulas in the search for a collection of formulas which witnesses the derivability of a given goal formula from a given set of axiom formulas. The prover proper is a meta-level object because its variables range over formulas (and other syntactic classes) of the object level language.

Thus a Prolog interpreter really defines a relationship between sets of formulas (the program database), goal formulas, and proofs, namely the relation that the proof witnesses the deducibility of the goal formula from the program database. (Note that the standard Prolog interpreters return a portion of the proof to the user, namely that part of the substitution applying to the variables occurring in the goal). As commonly implemented, pure Prolog interpreters incorporate the program database as a fixed part of the interpreter. Thus, from a meta-level point of view, a standard Prolog interpreter provided with a fixed program database defines a certain meta-level unary predicate applying to goal formulas. This meta-level unary predicate holds for just those goal formulas which are deducible from the program database by the interpreter. The fundamental operator of standard Prolog systems is thus a one-place operator (usually written `call(...)`) which invokes a search for a deduction of its argument from the implicit program database parameter. The heart of the proposal set forth by Bowen and Kowalski was to utilize a system implementing the full deducibility relation described above. Such a system would have metavariables which not only range over formulas and terms, but would also allow the metavariables to range over sets of formulas (called theories). The fundamental operator of such a system is a three-place operator, usually written `demo(Theory,Goal,Proof)`, which invokes a search for a proof of the goal formula appearing as its second argument from the theory (or program) appearing as its first argument.

All metaProlog program databases are the values of metaProlog variables and are set up either by reading them in from files or by dynamically constructing them using system predicates. Besides the built-in predicate `demo/3`, the system predicates include

```
add_to(Theory, Axiom, NewTheory)
```

```
drop_from(Theory, Axiom, NewTheory)
```

which build new theories from old ones by adding or deleting formulas. Thus for example, one might find the body of a clause containing calls of the form

..., add\_to(T1, A, T2), demo(T2, D, P),... (\*)

where the theory which is the value of T1 has been constructed by the earlier calls. The effect of (\*) would then be to construct a new theory T2 resulting from T1 by the addition of the formula A as a new axiom, and then the invocation of a search for a proof of the formula D from the theory T2. Since demo implements the proof relation, such programs as (\*) preserve the logical semantics of Prolog while providing for the dynamic construction of new databases from old.

The correctness and completeness of an implementation of demo are expressed by what were called reflection rules by Bowen and Kowalski:

If demo(T, A, P), then A is derivable from T via proof P.

If A is derivable from T via proof P, then demo(T, A, P).

These rules provide the justification for the implementation of calls on demo in the abstract metaProlog machine as context switches. In essence, at most times the machine behaves as a standard Prolog machine with the current theory (the analogue of the usual fixed program database) indicated by a register. When a call demo(T, A, P) is encountered, the database (theory) register is changed to point to T and a new search for a deduction of A is begun. Thus the efficiency of standard Prolog computations is preserved and the overhead of meta-level computation is localized in the construction of new theories from old. This approach provides a meta-level programming methodology suitable for constructing other methods of exploring the search space of derivations of A from T besides the top-down depth-first approach of standard Prolog. Exploitation of this approach will ultimately provide the meta-level programmer with a library of search strategies which can be (programmatically) invoked depending on the particular problem and context.

In order for any language M to serve as a metalanguage for another language L, M must contain names for all the appropriate syntactic entities of L. Thus, since metaProlog is to serve as its own metalanguage, it must contain names for all of its own syntactic entities, just as any natural language does. To this end, constants act as names of themselves. For non-constant items, metaProlog provides structural or non-structural names (and sometimes both), where the former are compound terms whose structure reflects the syntactic structure of the syntactic item they name. Facilities for manipulating names are provided, such as methods of obtaining the name of a compound expression from names of its components. And methods for moving between a name and the thing it names are included, analogous to univ (= ..) of ordinary Prolog.

One further subtle point regarding variables must be treated at this point. The logical interpretation of Prolog's theorem prover stipulates that variables actually occurring in the

program's clauses are in fact implicitly universally quantified object level variables, even though they are syntactically indicated by metavariables. In using a clause, the interpreter replaces these universally quantified object level variables by existentially quantified meta-level variables. The syntactic conflation of object- and meta-level variables is acceptable for pure Prolog deductions, but causes difficulties as soon as `assert` and `retract` are added to the system. If an expression (say `p(X)`) contains a metavariable `X` which is uninstantiated at the time when `assert(p(X))` is executed, there is a natural sense in which the call `assert(p(X))` is incoherent: the formula to be added to the database is not fully specified. The Prolog approach to this problem is to once again conflate the existentially quantified metavariable `X` with a corresponding universally quantified object-level variable, actually asserting `(all X)[p(X)]`. This approach destroys the logical semantics of clauses in which such calls occur. Assuming that there are no clauses for the predicate `p` already in the database, the goal statements of the following two clauses should be logically equivalent:

`h : - X = a, assert(p(X)), p(b). (A1)`

`h : - assert(p(X)), X = a, p(b). (A2)`

But the first fails, since it only adds `p(a)` to the database, while the second succeeds, since it adds `(all X)[p(X)]` to the database. To avoid such difficulties, the metaProlog system requires that programmers be explicit about their intentions, clearly indicating universally quantified object variables. Thus, to add `(all X)[p(X)]` to a theory `T`, one would write

`add_to(Theory, allX : p(X), NewTheory).`

Note that in the above expression, the symbols `X`, `Theory`, and `NewTheory` are metaProlog constants. There is no way a metaProlog programmer can write the name of a metaProlog variable. He or she can only indicate the position of such variables to the metaProlog interpreter by using the explicit universal quantifier which is represented by the symbol `all/1`. From the syntactic point of view, `all/1` is just a function symbol used to form terms. The symbol `all/1` functions as a quantifier only when a term formed with it occurs as a clause in a theory or as an argument to certain meta-level predicates. As an example, consider the following two metaProlog clauses which achieve the same effects as the clauses (A1) and (A2) above:

```
all [X,T1,T2]:
  h(T1, T2) ←
    X = a & add_to(T1, p(X), T2)
    & demo(T2, p(b), -).(B1)
```

```
all [T1,T2]:
  h(T1, T2) ←
    add_to(T1, all X : p(X), T2)
    & demo(T2, p(b), -).(B2)
```

(N.B. There is no sense in which the symbol X as it occurs in (B2) is a variable - it is a metaProlog constant.)

### III. The metaProlog System.

The metaProlog system is syntactically similar to the Edinburgh system. We use  $\leftarrow$  for the implication symbol (instead of  $:-$ ) and use  $\&$  as the conjunction operator, rather than comma. The major difference lies in our treatment of variables and constants. For the reasons we discussed above, we require that the implicit universal quantifiers on clauses be made explicit. Quantification is indicated by applying the function symbol `all/1` (which is parsed as a prefix operator) to a term whose principal functor is the binary infix operator `:/2` and whose first argument is either a metaProlog constant or a list of metaProlog constants and whose second argument is an arbitrary metaProlog term (we call such things "indicated terms"). Thus, for example, the Edinburgh clause

```
append([Head | Tail], Rt, [Head | R_Tail]) :-
    append(Tail, Rt, R_Tail).
```

could be written

```
all [Head, tail, Rt, r_Tail]:
    append([Head | tail], Rt, [Head | r_Tail])  $\leftarrow$ 
        append(tail, Rt, r_Tail).
```

If the clause contains only one variable, the list brackets in the quantifier can be dropped. (Dropping the convention of regarding symbols beginning with upper case as variables reduces the need for single quotes and the awkwardness that entails.)

The set of built-in predicates of pure Prolog exists as a subset of the metaProlog built-ins. (Indeed, pure Prolog is a subset of metaProlog, modulo the conventions regarding variable naming and quantification.) As is clear from the preceding sections, the three predicates `demo`, `add_to`, and `drop_from` constitute the core built-ins for manipulating theories and proofs (replacing `call`, `assert`, and `retract` from Prolog). We have already discussed `add_to` and `drop_from`. We need to discuss `demo` in somewhat greater detail.

Calls on `demo` support a convenient idiom for describing implicit unions of theories. Specifically, a call of the form

```
demo(Theory1&Theory2, Goal, Search_Info)
```

is logically equivalent to the call

```
demo(Theory3, Goal, Search_Info)
```

where `Theory3` is the ordered union of `Theory1` and `Theory2` in the following sense: If theories are regarded as the ordered list of their axioms, then `Theory3` satisfies

```
append(Theory1, Theory2, Theory3).
```

However, the system does not physically create Theory3, but regards the expression Theory1 & Theory2 as a description of a virtual theory. In effect, when searching for a rule or fact to apply to a selected subproblem of the current goal, it first searches Theory1 for a candidate, and only on failing to find such a candidate in Theory1, it then searches Theory2. Another usage supported is the explicit indication of the axioms of the theory. Namely, if it is desired to search for a deduction of G from A1, ..., An, this is achieved by the call

demo([A1, ..., An], G, Search\_Info).

The two usages can be combined, as in the calls:

demo([A1, ..., An] & Theory2, Goal, Search\_Info).

demo(Theory1 & [A1, ..., An], Goal, Search\_Info).

We have stated earlier that the proof predicate demo is a three-place relation holding between a Theory, a Goal, and a Search Space/Proof. We need to explain further the nature and use of the third argument. It may be used for a variety of purposes. These include extracting pieces of the proof or search space, controlling the search strategy, and introducing or extracting annotations to the proof, such as confidence factors. We intend this facility to be user-extensible. As a first step in this direction, search information expressions can be combined using the infix operator +/2, as in

demo(Theory, Goal, Search\_Info1 + Search\_Info2).

Two examples of search information annotations are proof(P) and branch(B). The proof(P) annotation causes the system to accumulate a representation of the proof branch in the (uninstantiated) variable P, allowing the programmer to extract a successful proof for further processing, such as providing explanations, etc. We see no reason to prevent the programmer from passing a partially constructed proof cum search\_space to demo through the use of proof(P). The branch(B) expression causes the call

demo(Theory, Goal, branch(B))

to succeed in all cases, binding the uninstantiated variable B to the left-most branch of the search tree. Note that in the case that the left-most branch is theoretically infinite, the call will still succeed due to depth bound limitations of the system. Backtracking into this call will cause B to be bound to successive branches of the search tree. As discussed in detail below, the call

setOf(B, demo(T, G, branch(B)), Branches)



would cause Branches to be bound to the (lazy) list of all branches of the search tree for G relative to T in the order that they are explored by the system.

As part of our program of providing powerful tools for AI programming, we seek to offer the programmer control of stream-based communication between concurrent processes, while still holding to our program of preserving the essential elements of Prolog semantics. In the logic programming context, this amounts to implementing some form of and-parallelism. The most straight-forward sort of and-parallelism to attack is simple producer - consumer computations. However, since the implementation of producer - consumer relations in which the producer is allowed to non-determinately reconsider the stream it has produced is difficult to say the least, we restrict ourselves to determinate and-parallel situations. Other approaches to parallelism in Prolog (e.g., Parlog (Clark and Gregory [198?]) or Concurrent Prolog (Shapiro [1983])) achieve this restriction by introducing committed choice. However, while preserving the correctness of the computations, this approach loses Prolog's deductive completeness. In contrast, we preserve both the correctness and completeness by restricting ourselves to running in parallel only producer - consumer computations in which the production of the stream is determinate. (Note that the computation of the stream may involve non-determinate aspects; it is simply at the point of adding a new element to the stream that the producer must act determinately. Also, consumption of the stream may be entirely non-determinate.) The essential point appears to us that it is not really the processes which must be forced to be determinate, but rather the communication between them. Thus our approach is to force the producing process to determinately fill the communication buffer; all else can be non-determinate.

We have identified two useful classes of producer - consumer computations which meet our requirement (and the possibility of others certainly exists). The first is the (lazy) production of sets via complete exploration of a search tree (i.e., the lazy form of Prolog's setof construct) and the production of streams by determinate tail-recursive procedures. These are indicated in metaProlog programs by the constructs

```
all_solutions(Template, Goal, Stream) and  
streamOf(Goal, Stream).
```

We see these as entirely encapsulated independent computations: their only method of communication with parent or sibling processes is via the stream variable. Every element of the stream must be ground. If the producing process would have otherwise produced a partially instantiated term as a stream element, that term must be converted to a ground term by use of the 'naming' or 'indicating' operator discussed above in conjunction with quantification. The same restrictions clearly must apply to the Goal argument of both stream\_of and all\_solutions. One method of implementation is that of producer variables. The first invocation of Goal binds the variable Stream to a buffer together with a description of Goal and its environment. Subsequent attempts to access the variable stream by the consumer causes Goal to be run through one cycle of its computation, binding Stream to a cons cell whose first element is the item produced and whose second element is a description of the rest of the buffer together with the current state of the

computation of Goal. It is important to recognize that the producer variable does not act like normal Prolog variable. Indeed, since any attempt to match a non-variable term against an element of the stream causes the stream element to be instantiated to a ground term by the producer, and since the producer is determinately committed to the binding it produces, producer variables behave for all intents and purposes as ground objects. Thus it is perfectly permissible for producer variables to appear in the Goal arguments of other producer processes. This allows for two-way communication between producers. Process synchronisation is achieved by requests for bindings passed from process to process. It is clear that the two communicating processes must be created simultaneously. The construct

`simultaneous(Process1, Process2)`

achieves this effect. It can be invoked with any number of arguments.

Because we see these processes as entirely sealed computations with their own environments, it is possible, in appropriate hardware settings, to run them truly in parallel, allowing the producing process to fill the buffer up to some pre-set limit or even run to completion when the stream is finite. On sequential hardware, the implementation is simple co-routining of the producer and consumer, with the additional overhead entirely localised in the communication - there is no slow down of the basic Prolog computation. In particular, the computational children of the Goal of one of these processes do not inherit the parallel mode: they run as normal Prolog processes. It should be possible to mix parallel and co-routined execution with no change to the program or its behavior. Finally, while we have not attempted to do so, it seems evident that or-parallelism could be introduced with a stream operator whose top level was expanded in an or-parallel manner. One might even introduce committed-choice versions of such an operator without disturbing the semantics of the rest of the system.

#### IV. A Programming Example.

In this section we describe approaches to fault-detection in digital circuits based on the ideas of Esghi [1982]. For the purposes of fault-finding, the devices must be described in some sort of predicate calculus formalism, for example `andGate(G, In1, In2, Out)`, which expresses that `G` is an and-gate with input lines `In1` and `In2`, and output line `Out`. Similarly for `orGate`. The topological description of the circuit is contained in the theory `c1`, which besides expressions such as those above, indicates the lists of input and output wires for the circuit. The behaviors of the circuit components are described in the theory `tt` (for truth tables) which contains such rules as:

```

all {Gate, In1, In2, Out}:
    andTable(Gate, In1, In2, Out) ←
        not(exceptional(Gate))
        & standardAnd(In1, In2, Out).

standardAnd(high, high, high).

all In2 :
    standardAnd(low, In2, low).

all In1 :
    standardAnd(In1, low, low).

```

The significance of the predicate "exceptional" will be described later. The topology and component behaviors can be used to predict the circuit outputs given the inputs as described in the theory laws which defines a predicate predict(InputValueList, OutputValueList) which calculates the output wire values by backchaining through the circuit from the output wires back to the input wires. Normal simulation of circuit function would be carried out by the call:

```
demo(c1 & tt & laws, predict(InList, OutList), -).
```

The fault detection problem consists of attempting to locate the source of the fault based on faulty input-output behavior. We will make the common simplifying assumption that the fault is caused by a single wire of a single gate being stuck at high or low. The basic method we will apply (due to Esghi) attempts, given a faulty input-output pair (If, Of), to determine a theory Tf obtained by minimal perturbation of the theory tt such that Tf correctly describes the behavior of the faulty circuit. Examination of Tf will then reveal the location of the fault. The basic algorithm runs as follows:

1. From tt and (If, Of), construct a set HYP of theories Hi such that:
  - i) For all i, demo(c1 & Hi & laws, predict(If, Of), -) succeeds;
  - ii) For some i, Hi correctly describes the faulty circuit.
2. If HYP contains only one element, halt and output HYP.
3. Otherwise, proceed as follows:
  - i) Choose distinct Hi and Hj from HYP;
  - ii) Construct a discriminating input Id which distinguishes Hi and Hj; if no such input exists for any choice of Hi and Hj, halt and output HYP.
4. Apply Id to the faulty circuit, obtaining output Od.
5. Delete from HYP all Hi for which the following call fails:

```
demo(c1 & Hi & laws, predict(Id, Od), -).
```
6. Goto 2.

Once the set HYP is constructed in step 1, the remainder of the algorithm is basically

straight-forward (though we will return to it below). The set HYP is constructed by first making the call:

```
setOf(B, demo(c1&tt&laws, predict(If,Of),
             user_choice+branch(B)), BadBranches)
```

First note that the goal of this setOf would fail without the control information since tt describes the correct circuit, while (If, Of) is faulty for this circuit. But the control branch(B) causes the setOf to produce the list of all branches of the search tree. The user\_choice forces these branches to be constructed according to the user choice theory uc which describes, in the style of Pereira[], a next-goal choice procedure which delays as long as possible selecting goals of the form

"andTable(→ → -)" or "orTable(→ → -)".

Next each of the failed proof branches on BadBranches is used to guide the generation of candidate theories Hi for HYP. Essentially, tt is modified so that failing calls of the form

"andTable(G, → -)" or "orTable(G, → -)"

become successful: essentially the failing call is added to tt together with the assertion "exceptional(G)" to produce Hi. HYP is then filtered by steps 2-6.

For any realistic circuits, the lists BadBranches and HYP will be unmanageably large if produced in their entirety. However, the lazy nature of setOf causes the production of BadBranches to be co-routined with the action of gen(BadBranches, HYP) which generates HYP from the elements of BadBranches. The procedure gen is defined as a tail-recursive streamOf construct, so that it can in turn be co-routined with the filter process implementing 2-6. The nature of the streamOf and simultaneous constructs allows the dynamic generation of processes. This permits filter to be organised in a manner analogous to the classic parallel implementations of the sieve of Eratosthenes. First, each Hi making it through the current filter is recorded on a working list. Next, as each pair Hi, Hj makes it through the filter, the discriminating pair (Id, Od) is generated, and used to produce a small check process check(Id, Od, H) which tests H to determine whether H correctly predicts the i-o pair (Id, Od). This check process is attached at the current end of the filter, much as the divisor test for the most recently generated prime is attached at the end of the sieve. Also, as each pair (Id, Od) is generated, check(Id, Od, H) is applied to each element of the current temporary scratch list, and any H on that list which fail the test are removed. The entire process gradually terminates as each of the processes, from the initial setOf call through last check process gradually close down (by seeing the streams they are consuming being closed). When the last of these processes closes down, the elements remaining on HYP all correctly predict the faulty i-o pair and pass all the tests for behavior of the real faulty circuit which have been generated. If HYP contains

more than one element, these hypothetical cannot be distinguished by i-o behavior. They are all candidate Hi descriptions of the possible source of the fault. Finally, let us note that these methods can be adapted to a setting of hierarchical diagnosis in the style of Genesereth [1982].

## V. Conclusion.

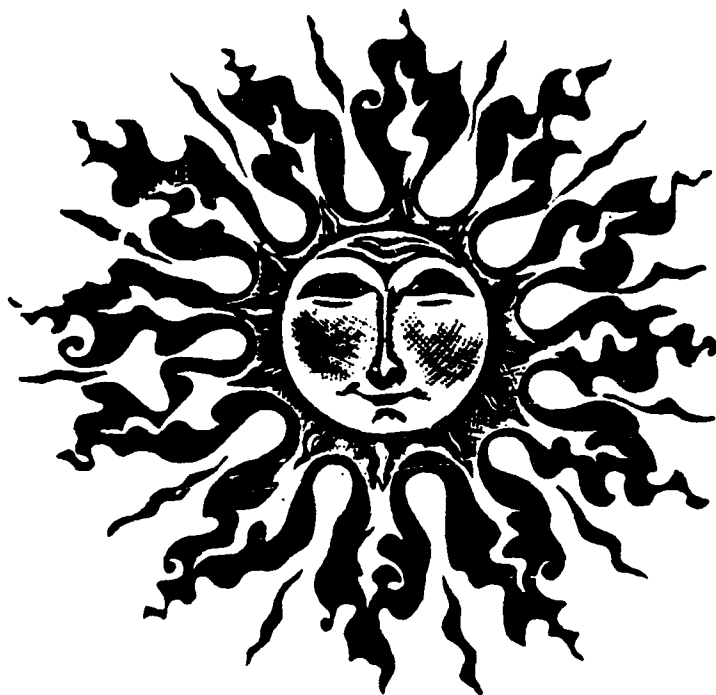
We have elaborated a system called metaProlog which, narrowly conceived, is an extension of Prolog. The real power (meta-power) of this system lies not in the specific system facilities we have described, but in the programming methodology they introduce. The example in the preceding section only beings to explore the possibilities of this system. Using this approach, we have begun to logically characterise frames and default hierarchies, generalised networks of theories and semantic nets, and more general control strategies such as bottom-up or breadth-first search. There is no logical requirement that the only notion of proof in metaProlog be the Horn clause-oriented demo predicate we have introduced. We see no reason why other methods of proof cannot co-exist with demo. We envisage the situation in which another method of proof would be rapidly prototyped using explicit recursive calls on the present demo, and later integrated into the system at a low level using the same bootstrapping methods we are adopting for the implementation of the basic metaProlog system.

By stepping up to the full meta-level point of view wherein all components of the system have become first-class objects, we have entered the realm of a logical construal of Theories, Goals, and SearchSpaces in which it is possible to axiomatically and programmatically characterize elements of the system previously regarded as parts of the implementation. This allows us to introduce powerful logical approaches to the construction of artificial intelligence systems.

## References

- Bowen, K.A. and Kowalski, R.A., *Amalgamating language and metalanguage in logic programming*, in *Logic Programming*, eds Clark and Tarlund, Academic Press, 1982, pp. 153-172.
- Clark, K., and Gregory, S., *Parlog: A parallel logic programming language*, Research Report DOC 83/5, Imperial College, March 1983.
- Eshghi, K., *Application of meta-language programming to fault finding in logic circuits*, in *First International Logic Programming Conference, 1982*, pp 240-246.
- Genesereth, M., *Diagnosis using hierarchical design models*, *Proc. Nat'l Conf. on Artificial Intelligence, 1982*, pp. 278-283.
- Pereira, F., and Warren, D.H.D., *Definite clause grammars for language analysis - a*

*survey of the formalism and a comparison with augmented transition grammars, Artificial Intelligence 13 (1980), pp. 231-278.*





*MISSION*  
*of*  
*Rome Air Development Center*

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C<sup>3</sup>I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C<sup>3</sup>I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END

DATE

FILMED

APRIL

1988

DTIC