Kautz
780

780
13989

A Formal Theory of Plan Recognition

Henry A. Kautz*
Department of Computer Science
University of Rochester, Rochester, NY 14627

TR 215
May 1987

This report reproduces a thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy. The work was supervised by Dr. James F. Allen.

* Current address:
AT&T Bell Laboratories
Murray Hill, N.J. 07974

Curriculum Vita

Henry Alexander Kautz was born in 1956 in Youngstown, Ohio. After obtaining the highest score statewide on the 1974 New York State Regents Scholarship Examination, he entered the Case Institute of Technology in 1974, and transferred to Cornell University a year later. He received both an B.A. in English and one in mathematics from Cornell in 1978, graduating with highest honors. Mr. Kautz worked as a systems analyst for a year before winning a fellowship to the creative writing program at the Johns Hopkins University in 1979. During this time he wrote two professionally produced plays, and was awarded a M.A. by the Writing Seminars in 1980.

That fall Mr. Kautz returned to computer science, enrolling at the University of Toronto in 1980, supported by the Connaught Fellowship for foreign students. He produced his Master's thesis, *A First-Order Dynamic Logic for Planning*, under the supervision of Professor Ray Perrault, and received an M.S. in Computer Science in 1982. The National Science Foundation selected Mr. Kautz for a three-year fellowship that year, and he returned to the United States, entering the Department of Computer Science at the University of Rochester. Mr. Kautz was a teaching assistant for professors Patrick Hayes and James Allen in the Fall of 1983 and Spring of 1984 respectively, and was a research assistant for James Allen, his thesis advisor, for 1982–1983, and 1984–1987. He held research appointments in the summer of 1983 at BBN Labs in Cambridge, and in the summer of 1984 at SRI International, in Palo Alto, California. Mr. Kautz published papers on a number of topics in Artificial Intelligence during his tenure as a graduate student, including three presented at the 1986 annual convention of the American Association for Artificial Intelligence. Mr. Kautz finished his PhD thesis while employed as a Knowledge Representation Consultant at Bell Laboratories in Murray Hill, New Jersey, in the Spring of 1987.

Acknowledgments

The National Science Foundation helped supported my graduate study under Fellowship Grant No. RCD-8450125 and Research Grant No. DCR-8502481. This work was also supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441, and the Air Force Office of Scientific Research, Bolling Air Force Base, Washington, D.C. 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC). I thank the Xerox Corporation University Grants Program for providing equipment used in the preparation of this paper. The final document was produced on an Apple Macintosh© and Laserwriter©.

I thank my thesis advisor, James Allen, for the countless hours we spent in research meetings and seminars over the years. He was always quick to suggest new ideas, sketch solutions to problems I ran into, and to help separate the wheat from the chaff. The faculty and staff at the University of Rochester provided an exceptionally supportive and collegial environment for graduate students. Thanks for interesting classes and wide ranging discussions go to professors Patrick Hayes, Henry Kyburg, and Jerry Feldman.

Giddeon Frieder helped me get my first enjoyable summer job, working for the University of Buffalo Computer Science Department, and was fundamental in my decision to go to graduate school in computer science. Alex Borgida first introduced me to work in A.I. at the University of Toronto, where I was fortunate enough to become the student of Ray Perrault, both of whom taught me to think clearly and rigorously. Mention must be made of Gordy McCall, who hosted the Canadian A.I. Conference in Saskatoon; afterwards I felt like an honorary Canadian citizen for life. I am extremely grateful to Ron Brachman and Bell Laboratories for putting me on the payroll while I finished this document.

Time spent with fellow students Leo Hartman, Diane Litman, Josh Tenenberg, Jay Weber, Paul Kates, and Robin Cohen provided food for thought, as well as many fine meals. Jim Mayer taught me everything I know about making pasta, for which I more than forgive his utter skepticism about "artificial intelligence".

My parents were a constant source of moral and financial support during the many years of graduate study. Thanks for always encouraging me to try for the best.

This thesis is dedicated (of course) to Christine, who helped me celebrate the highs and saw me through the lows of working on a PhD. Her constant support, understanding, encouragement, and love have made this not only possible, but truly worthwhile.

A Formal Theory of Plan Recognition

Henry Alexander Kautz

Abstract

Research in discourse analysis, story understanding, and user modeling for expert systems has shown great interest in plan recognition problems. In a plan recognition problem, one is given a fragmented description of actions performed by one or more agents, and expected to infer the overall plan or scenario which explains those actions. This thesis develops the first formal description of the plan recognition process.

Beginning with a reified logic of events, the thesis presents a scheme for hierarchically structuring a library of event types. A semantic basis for non-deductive inference, called "minimum covering entailment", justifies the conclusions that one may draw from a set of observed actions. Minimum covering entailment is defined by delineating the class of models in which the library is complete and the set of unrelated observations is minimized. An equivalent proof theory forms a preliminary basis for mechanizing the theory. Equivalence theorems between the proof and model theories are presented. Minimum covering entailment is related to a formalism for non-monotonic inference known as "circumscription". Finally, the thesis describes a number of algorithms which correctly implement the theory, together with a discussion of their complexity.

The theory is applied to a number of examples of plan recognition, in domains ranging from an operating system advisor to the theory of speech acts. The thesis shows how problems of medical diagnosis, a similar kind of non-deductive reasoning, can be cast in the framework, and an example previously solved by a medical expert system is worked out in detail.

The analyses provides a firm theoretical foundation for much of what is loosely called "frame based inference", and directly accounts for problems of ambiguity, abstraction, and complex temporal interactions, which were ignored by previous work. The framework can be extended to handle difficult phenomena such as errors, and can also be restricted in order to improve its computational properties in specialized domains.

Table of Contents

Curriculum Vita ii
Acknowledgments iii
Abstract iv
List of Figures x
Poèmes humoristiques sur l'AI xi

Chapter 1
Introduction 1

1.1. Motivation 1
1.2. Overview of Thesis 2
1.3. Related Work on Plan Recognition 6
1.3.1. Story Understanding 6
1.3.1.1. Psychological Modeling 6
1.3.1.2. Script Based Systems 7
1.3.1.3. Abduction 8
1.3.2. Discourse 10
1.3.2.1. Allen and Perrault 10
1.3.2.2. Extended Discourse 11
1.3.2.3. Cohen and Levesque 13
1.3.3. Intelligent Computer Environments 14
1.3.3.1. The MACSYMA Advisor 14
1.3.3.2. A Smart Operating System: Plan Parsing 15
1.4. Related Work on Medical Diagnosis 16
1.4.1. INTERNIST and CADUCEUS 18
1.4.2. A Set Covering Model 19
1.5. Related Work on Non-Deductive Inference 19
1.5.1. Probability Theory 20
1.5.2. Default Logic 23
1.5.3. Circumscription 23

Chapter 2
Representing Event Hierarchies 27

2.1. Language 27
2.2. Representation of Time, Properties, and Events 28
2.3. The Event Hierarchy 30
2.4. Components of Event Tokens 31
2.5. Acyclic Hierarchies & Compatible Types 32
2.6. Example: The Cooking World 32
2.6.1. Diagrammatic Form 33
2.6.2. The Abstraction Hierarchy 34

2.6.3. The Decomposition Hierarchy 35
2.6.4. Describing Instances of Events 36
2.7. Conditional Actions 37
2.7.1. Representing Conditional Actions 37
2.7.2. An Example 39

Chapter 3
Covering Models 40

3.1 Model Minimization 40
3.2 Completing the Abstraction Hierarchy 41
3.2.1. Theorem 3.1 (Exhaustiveness) 42
3.2.2. Theorem 3.5 (Disjointedness) 42
3.2.3. Theorem 3.7 (Unique Basic Types) 42
3.2.4. Theorem 3.9 (Abstraction Completeness) 42
3.3 Completing the Decomposition Hierarchy 43
3.3.1. Definition of Covering Model and C-Entailment 43
3.3.2. Theorem 3.10 (No Useless Events) 43
3.3.3. Theorem 3.11 (Component/Use) 43
3.3.4. Theorem 3.13 (No Infinite Chains) 44
3.3.5. Theorem 3.14 (Decomposition Completeness) 44
3.3.6. Theorem 3.15 (Computability of C-Entailment) 44
3.3.7. Theorem 3.16 (Not Predicate Completion) 44
3.4 Circumscription 45
3.4.1. Theorem 3.17 (C-entailment and Circumscription) 45
3.5 Example: The Cooking World, Continued 46
3.5.1. Exhaustiveness Assumptions (EXA) 46
3.5.2. Disjointedness Assumptions (DJA) 46
3.5.3. Component/Use Assumptions (CUA) 47
3.5.4. A Simple Recognition Problem 49

Chapter 4
Minimum Covering Models 51

4.1. Cardinality Minimization 51
4.1.1. Theorem 4.1 (Minimum Cardinality Defaults) 52
4.2. Cardinality Circumscription 53
4.2.1. Circumscription with Variables 53
4.2.2. Theorem 4.2 (Cardinality Circumscription) 54
4.2.3. Example of Cardinality Circumscription 54
4.3. Example: The Cooking World, Continued 56

Chapter 5
Incremental Recognition 58

5.1. Deficiencies in the MC Model 59
5.1.1. The Combinatorial Problem 59
5.1.2. The Persistence Problem 59
5.2. Refining the Model 61
5.2.1. Incrementally Minimize Cardinality 61
5.2.2. Sticky Covers 62

5.2.3. Discourse Clues 62
5.2.4. Likelihood Associations 62
5.3. Model Theory for Incremental Minimization 63
5.3.1. Minimum Covering Submodels 63
5.3.1.1. Definitions 64
5.3.1.2. Theorem 5.1 (Non-Universal Conclusions) 64
5.3.2. Monotonic Incremental Recognition 65
5.3.2.1. Definitions 65
5.3.2.2. Theorem 5.2 (Incremental Recognition) 66

Chapter 6
Examples 67

6.1. The Cooking World, Once More 67
6.2. Indirect Speech Acts 69
6.2.1. Representation 69
6.2.2. Assumptions 72
6.2.3. The Problem 72
6.3. Operating Systems: Multiple Events 75
6.3.1. Representation 75
6.3.2. Assumptions 77
6.3.3. The Problem 78
6.4. Medical Diagnosis 82
6.4.1. Representation 82
6.4.2. Assumptions 84
6.4.3. The Problem 85

Chapter 7
Algorithms for Plan Recognition 91

7.1. Directing Inference 91
7.2. Explanation Graphs 93
7.2.1. Basic Elements of an E-Graph 93
7.2.2. Roles of Event Types and of Nodes 94
7.2.3. Definition of an Explanation Graph 95
7.3. Computing the Uses of an Event Type 97
7.3.1. Use Abstraction and Specialization 97
7.3.2. The Uses and Direct Component Relations 98
7.3.3. Definition of Uses 98
7.3.4. Example 100
7.4. Constraint Checking 100
7.4.1. Equality Constraints 101
7.4.2. Temporal Constraints 101
7.4.3. Fact Constraints 102
7.5. Overview of the Algorithms 102
7.5.1. Explain 102
7.5.2. Match 103
7.5.3. Group 104
7.6. Pseudo-Code 104
7.6.1. Utility Functions 104
7.6.2. Explain-Observation 105

7.6.3. Match-Graphs 108
7.6.4. Group Observations 109
7.6.4.1. Non-Dichronic Version 109
7.6.4.2. Incremental Minimization Version 110
7.6.4.3. Sticky Version 111
7.7. Description of Operation 112
7.7.1. Explain 112
7.7.2. Match 114
7.7.3. Group 117
7.8. Completeness & Correctness 118
7.9. Complexity 119
7.9.1. Explain 119
7.9.2. Match 120
7.9.3. Group 120
7.10. Transcripts 121

Chapter 8
Conclusions 123

8.1 The Three Levels 123
8.2 Applicability 124
8.3 Generality & Extensibility 125
8.4 Two Unresolved Issues 126

References 128

Appendix A
Proofs for Chapter 3 134

Theorem 3.1 (Exhaustiveness) 134
Theorem 3.2 135
Theorem 3.3 136
Theorem 3.4 137
Theorem 3.5 (Disjointedness) 137
Theorem 3.6 139
Theorem 3.7 (Unique Basic Types) 140
Theorem 3.8 140
Theorem 3.9 (Abstraction Completeness) 141
Theorem 3.10 (No Useless Events) 142
Theorem 3.11 (Component/Use) 143
Theorem 3.12 144
Theorem 3.13 (No Infinite Chains) 145
Theorem 3.14 (Decomposition Completeness) 145
Theorem 3.15 (Computability of C-Entailment) 147
Theorem 3.16 (Not Predicate Completion) 148
Theorem 3.17 (C-entailment and Circumscription) 149

Appendix B
Proofs for Chapter 4 150

Theorem 4.1 (Minimum Cardinality Defaults) 150

Theorem 4.2 (Cardinality Circumscription) 151

Appendix C
Proofs for Chapter 5 155

Theorem 5.1 (Non-Universal Conclusions).. 155
Theorem 5.2 (Incremental Recognition) 156

Appendix D
Temporal Constraint Logic 157

Appendix E
Transcripts 160

Hunt/Rob Example 160
Cooking Examples 164
Operating System Examples 172
Language Examples 176

Appendix F
Details of Correctness Proof of **Explain** 184

List of Figures

1.1: Hunt/Rob Hierarchy 3

2.1: Cooking Hierarchy..34

6.1: Language Use Hierarchy..70

6.2: Operating System Hierarchy..76

6.3: Medical Hierarchy..83

6.4: Conclusions from Jaundice...86

6.5: Conclusions from Pallor..87

6.6: Conclusions from Jaundice and Pallor....................................87

7.1: E-Graph for MakeMarinara..92

7.2: Extended Cooking Hierarchy Detail......................................98

7.3: Multiple Inheritance Hierarchy...103

7.4: E-Graphs for MakeMarinara, MakeNoodles, and their Match............115

7.5: Combinatorially Explosive Hierarchy...................................120

8.1: Erroneous Events...125

E.1: E-Graph for Get-Gun...161

F.1: Relation of Ec(nc) to E0(n0)..185

F.2: Case 1..185

F.3: Case 2..186

F.4: Case 3..187

Poèmes humoristiques sur l'AI

If you're dull as a napkin, don't sigh;
Make your name as a "deep" sort of guy.
    You just have to crib, see
    Any old book by Kripke
And publish in AAAI.

A hacker who studied ontology
Was famed for his sense of frivolity.
    When his program inferred
    That Clyde ISA bird
He blamed – not his code – but zoology.

If your thesis is utterly vacuous
Use first-order predicate calculus.
    With sufficient formality
    The sheerist banality
Will be hailed by the critics: "Miraculous!"

If your thesis is quite indefensible
Reach for semantics intensional.
    Your committee will stammer
    Over Montague grammar
Not admitting it's incomprehensible.

Chapter 1
Introduction

1.1. Motivation

Perhaps the central concern of Artificial Intelligence is to devise methods for representing and reasoning about actions and plans. While plan synthesis has received careful formal analyses [McCarthy & Hayes 69], the inverse problem of plan recognition (or action interpretation) has appeared in mainly empirical and domain-specific programs of research. These include work on story understanding [Bruce 81, Wilensky 83], psychological modelling [Schmidt 78], natural language pragmatics [Allen 83, Litman 84], intelligent computer system interfaces [Huff & Lesser 82], and strategic planning. In each case, one is given a fragmented, impoverished description of the actions performed by one or more agents, and expected to infer a rich, highly interrelated description. The new description fills out details of the setting, and relates the actions of the agents in the scenario to their goals and future actions. The result of the plan recognition process can be used to generate summaries of the situation, to help (or hinder) the agent(s), and to build up a context for use in disambiguating further observations. This thesis develops a formal analysis of plan recognition. The analysis provides a firm foundation for much of what is loosely called "frame based inference" [Minsky 75], and directly accounts for problems of ambiguity, abstraction, and complex temporal interactions, which were ignored by previous approaches.

Plan recognition problems can be classified as cases of *intended* or *keyhole recognition* [Cohen, Perrault, & Allen 81]. In the first case, but not the second, the observer can assume that the agent is deliberately structuring his activities in order to make his intentions clear. Recognition problems can also be classified as to whether the observer has complete knowledge of the domain, and whether the agent may try to perform erroneous plans [Pollack 86]. This thesis concentrates on keyhole recognition of correct plans, where the observer has complete knowledge. We will consider, however, some examples from discourse, which are cases of intended recognition.

An important preliminary step is to define the scope of inferences which must be treated by a theory of plan recognition. Plan synthesis can often be viewed as purely hypothetical reasoning (i.e., if I did A, then P would be true). Some attempts have been made [Charniak 85] to formalize plan recognition as a similar kind of hypothetical reasoning: infer a plan P, such that if the agent did P, then he would do the observed action A. Only a space of *possible* inferences is outlined, and little or nothing is said about why one should infer one conclusion over another, or what one should conclude if the situation is truly ambiguous. (Such criticism also applies to work based on "plausible" inference [Allen 83, Cohen 84, Pollack 86].) We insist that a theory of plan recognition specify what conclusions are absolutely *justified* on the basis of the observations, our knowledge of actions, and other explicit assumptions. In fact, our framework allows one to draw conclusions based on the *class* of *simplest* plans which contain the observed actions. Finally, while planning can be formalized as pure deduction, to formalize plan recognition we must develop a non-deductive, non-monotonic[1] system of inference.

An advantage of this approach is that the model and proof theories apply to almost any situation. They handle disjunctive information, multiple concurrent (and unrelated) plans, steps "shared" between plans, abstract event descriptions, and both incremental and non-incremental recognition. The corresponding algorithmic theory may place restrictions on the form of information handled in order to gain processing efficiency. The search for more general and efficient algorithms can continue without the need to revise or reinvent the basic principles upon which the theory of recognition is founded.

The vocabulary that has been used to describe plan recognition varies considerably. We will speak uniformly of observations as *descriptions of events*. The observer's knowledge is represented by a set of first-order statements called an *event hierarchy*. The result of the recognition process is a description of the *end events*, those which are self-contained and self-justifying, which make up the situation. The wide applicability of the event vocabulary suggests that this work is relevant to areas of Artificial Intelligence not normally associated with planning or plan recognition, such as diagnostic reasoning. We will examine a simple medical-diagnosis problem using our logical machinery.

1.2. Overview of Thesis

An event hierarchy is a collection of restricted-form first-order axioms, used to define the abstraction, specialization, and functional relationships between various kinds of events. The functional, or "role/value", relationships include the relation of an event to its *component* subevents. There is a distinguished type-predicate, End, which

[1]A system is non-monotonic if some conclusions which may be drawn from a given set of assumptions may no longer be drawn from a larger set of assumptions.

holds of events which are not components of any other events. Recognition is the problem of classifying the End events which generate a set of observed events. The second half of this chapter reviews research in plan recognition in many different areas of Artificial Intelligence, as well as related work in non-monotonic inference.

Chapter 2 formally defines event hierarchies, and shows how they may be used to represent hierarchically-structured plans of action. It argues for the particular method of representation, and describes the expressive limitations of alternative systems. Event hierarchies can encode conditional actions without additional machinery. A hierarchy of actions involved in cooking is developed in detail, and serves as a basis for examples throughout the rest of the thesis.

An event hierarchy does not, however, by itself justify inferences from observations to End events. The lexical hierarchy justifies deductions from an event to its components; it does not rule out the possibility that those components may occur without reference to any of the End events mentioned in the hierarchy. Consider the following example. (The thick grey arrows denote abstraction or "isa", and the thin black arrows denote component or "has part".)

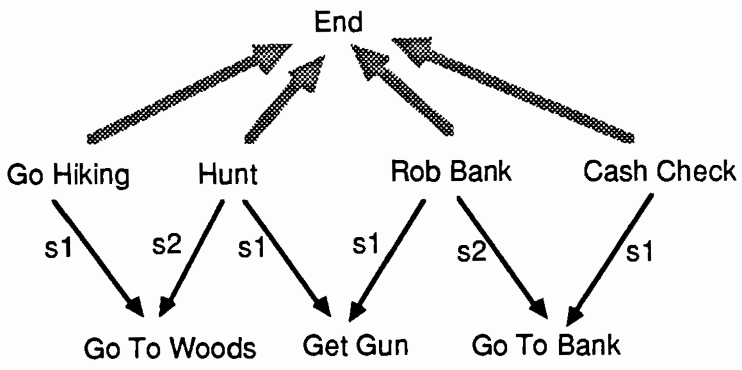


figure 1.1: Hunt/Rob Hierarchy

Suppose GetGun(C) is observed. This statement, together with the hierarchy, H, does not entail $\exists x \, . \, \text{Hunt}(x)$, or $\exists x \, . \, \text{Hunt}(x) \vee \text{RobBank}(x)$, or even $\exists x \, . \, \text{End}(x)$. There are models of $\{\text{GetGun}(C)\} \cup H$ in which none of these statements hold. For instance (where we describe a model by listing its positive atoms), none hold in $\{\text{GetGun}(C)\}$, and only the last in $\{\text{GetGun}(C), \text{CashCheck}(D), \text{GoToBank}(s1(D)), \text{End}(D)\}$.

Yet it does seem reasonable to conclude that someone is either hunting or robbing a bank, on the basis of the given hierarchy. This conclusion is justified by assuming that the event hierarchy is *complete:* that is, whenever a non-End event occurs, it must be part of some other event, and the relationship from event to component appears in the hierarchy. This completeness assumption can be expressed by defining a special subclass of models of H, called "covering models". For the example, {GetGun(C), C=s1(D), Hunt(D), GoToWoods(s2(D)), End(D)} and {GetGun(C), C=s1(D), RobBank(D), GoToBank(s2(D)), End(D)} are covering models of H, but none of the other models described above are.

Chapter 3 uses the notion of a covering model is used to define a new semantic relation, called *c-entailment.* In this example, hunting or bank robbing occurs in all covering models in which an agent gets a gun, or formally:

$$\mathrm{GetGun(C)}\ {}_{\mathrm{H}}\!\models_{\mathrm{c}} \exists x\ .\ \mathrm{Hunt}(x) \vee \mathrm{RobBank}(x)$$

The second part of that chapter relates c-entailment to ordinary entailment and deduction. An easily-computed "closure" function **cl** is defined, with the property that a statement is c-entailed by an observation if and only if that statement deductively follows from the observation and the closure of the event hierarchy. That is,

$$\Gamma\ {}_{\mathrm{H}}\!\models_{\mathrm{c}} \Omega \quad \text{if and only if} \quad \Gamma \cup \mathrm{cl(H)} \vdash \Omega$$

Thus cl(H) axiomatically captures the class of covering models; or, equivalently, explicitly states the closure assumptions in effect when one uses an event hierarchy for recognition. The theory of c-entailment is related to John McCarthy's system of non-monotonic inference known as *circumscription.*

When several events are observed, still stronger assumptions are commonly employed. Suppose that {GetGun(C), GoToBank(D)} is observed. This set does not c-entail an instance of robbery; the model containing an instance of hunting *and* an instance of check cashing provides a counterexample. By Occam's razor (do not multiply entities unnecessarily) we *would* be justified in concluding ∃x . RobBank(x); this principle can be realized by distinguishing the *minimum covering models* of the observations. These models define a final semantic relation between observations and conclusions, *mc-entailment.* In the example,

$$\{\mathrm{GetGun(C)},\ \mathrm{GoToBank(D)}\}\ {}_{\mathrm{H}}\!\models_{\mathrm{mc}} \exists x\ .\ \mathrm{RobBank}(x) \wedge \mathrm{C}=\mathrm{s1}(x) \wedge \mathrm{D}=\mathrm{s2}(x)$$

Chapter 4 develops the theory of minimum covering models and mc-entailment. Mc-entailment is shown to correspond to a particular case of *circumscription with variables.*

Chapter 5 extends the theory to describe *incremental* plan recognition. An incremental recognition process *cyclically* makes observations and infers consequences. The conclusions reached at the end of any particular cycle form the basis for those reached in the next cycle. Incremental recognition is *dichronic*: the conclusions reached may depend upon the order of the observations. While this may make it non-optimal for some recognition problems, we shall argue that it is a plausible description, on both computational and psychological grounds, of the kind of inference performed in plan recognition and its applications.

This formal framework shows how one can infer "up" an event hierarchy, and unify the explanations of several observations in order to reach a stronger conclusion. Very few restrictions are placed on the kinds of events which can be encoded: disjunctions may appear in plans or in observations, observations may be incomplete, and arbitrary temporal constraints may appear between events. Several examples of plan recognition are discussed in detail in Chapter 6, from the domains of cooking, natural language pragmatics, and operating system interfaces. The chapter concludes with a comparison of plan recognition to medical diagnosis. Diseases can be taken to be End events, and symptoms to be component events. A problem handled by a version of INTERNIST, a medical expert system, is recast in our framework.

Traditional work in artificial intelligence on high level[2] recognition problems has often relied on graph-matching algorithms. A lexical hierarchy is very naturally represented as a labeled digraph, and such graphs can be given a straightforward semantic interpretation. Chapter 7 describes some graph-based recognition algorithms which are justified by the notion of mc-entailment: that is, they compute structures which can be interpreted as statements true in all minimum covering models. The algorithms are useful and interesting in that they suggest what conclusions *should* be drawn, while the formal framework only specifies what conclusions *can* be drawn. There are three basic algorithms. The first constructs an AND/OR graph bottom up, from each observation to an instance of End. The second minimizes the set of End events, by performing a top-down match of the structures created in the previous step. The third algorithm controls the first two, and decides which observations are part of

[2]High level recognition problems involve the application of a great deal of specific world knowledge to a relatively small amount of symbolic (non-numeric) data. Low level recognition problems reverse this situation: there is a massive amount of quantitative data, interpreted by general principles about the physical nature of the domain.

the same of End event. Has the framework's power and generality been bought at the cost of computational intractability? We believe not. While it is easy to show that the *worst-case* cost of solving a plan recognition problem matches that of general deduction (namely, exponential on the size of the knowledge base), careful use of event *abstraction* significantly collapses the search space.

1.3. Related Work on Plan Recognition

Work in A.I. on plan recognition has concentrated on story understanding, discourse, and intelligent computer interfaces and environments. Each area is reviewed in roughly chronological order.

1.3.1. Story Understanding

1.3.1.1. Psychological Modeling

One of the first papers to explicitly invoke the phrase "plan recognition" was the report by [Schmidt, Sridharan, & Goodson 78] 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 and his colleagues conducted experiments in which human subjects were presented simple descriptions of sequences of actions by a single agent. The sequences were interrupted at various points, and the subjects were asked to summarize the 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.

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. Schmidt argued 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. Once the setting of the story was input, BELIEVER would retrieve a single parameterized plan from memory. As each observation was input, BELIEVER tried to match the description against an *expected* action (that is, an action in the plan whose preconditions were true). Failure of the observation to match an expected action would trigger "critics", that is, specialized pieces of code which revised the plan to account for errors and accidents by the agent.

The work on BELIEVER is quite different from our own. While BELIEVER was a "top down" inference system, we have concentrated on the "bottom up" aspects of plan recognition: if *many* different ultimate plans are possible, how do we home in on just a few possibilities? BELIEVER was offered as a rough psychological model, while we have concentrated on building a tight mathematical model of "ideal" performance. Schmidt's work began to deal with the critical issues of errors and accidents, which we have not examined. (Recent work by [Pollack 84] investigates problems in recognizing erroneous plans.) Both Schmidt and our work stress the use of a library of plans, and the importance of "sketchy" or abstract plans; our work extends BELIEVER's notion of abstraction, which involved plans with uninstantiated parameters, but not a hierarchy of abstract plan types.

After the BELIEVER project interest grew in devising systems that could understand stories which involved a number of different characters, each pursing goals and plans which could interact and conflict with the plans of the other characters. [Bruce 81] presented a detailed analysis of the interacting plans in a number of children's stories, including *Hansel and Gretel*. Unfortunately Bruce did not continue this line of research with a computational model. The greatest nexus of interest in story understanding grew up in Roger Schanks' group at Yale.

1.3.1.2. Script Based Systems

Schank'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 planning and plan recognition, [Wilensky 82] extended script theory to handle more dynamic domains. Wilensky argued that most everyday planning problems involve little or no search ("canned" plans are known for all possible goals) but do involve complicated interactions between goals. Furthermore, plan recognition is inextricably intertwined with plan synthesis. For example, given the following text:

> *John wanted the newspaper. It was raining outside, so John called for his dog Spot.*

Wilensky claimed one would 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 (the reader's beliefs about) John's wants. The fact that it is raining would invoke the *stay dry* goal, so this is added to John's wants. The reader 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 reader learns that John called for Spot. The effect of this is determined to be that Spot is with John. The reader 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.

While parts of this theory were implemented in many computer programs, much remained vague. A crucial feature were the so-called *text comprehension principles,* which were used to "tie together" the individual sentences of the story. These were *coherence* , meaning the character's plan is consistent; *least commitment* , meaning that a reader shouldn't prematurely assume that *any* particular explanation found is *the* explanation, and then have to undo it; and *parsimony* , meaning that a reader should "maximize the connections between the inputs". Specific versions of these general principles appear in our framework for plan recognition. Our model theoretic approach forces us to only consider consistent plans (inconsistent plans would have no model). Least commitment arises from concluding what holds in *all* minimum covering models, rather than a particular one. The principle of parsimony corresponds to our minimization of unrelated events.

1.3.1.3. Abduction

Recent work by [Charniak 85] views plan recognition, or "motivation analysis", as a kind of *abductive inference.* The term "abduction" comes from an early attempt in the philosophy of science to look at explanation as the process of finding the best hypothesis which logically entails the thing to be explained [Peirce 58].[3] For example, suppose we known that all men are mortal:

$$\forall x \, . \, \mathrm{man}(x) \supset \mathrm{mortal}(x)$$

The thing to be explained is that Socrates is mortal. An abductive inference would take us from mortal(Socrates) to man(Socrates). Thus Socrates is mortal because he's a man. Abduction is obviously a *very* unsound rule of inference, that can lead to ridiculous conclusions. Pierce never claimed abduction was all that was involved in scientific explanation: it was merely a method of *generating* hypotheses that could be tested and weighed by other means.

Charniak's system employs abductive techniques to generate hypotheses which entail the (logical forms of) the sentences in simple stories. In order to limit the amount of inference, only a small subset of the reader's knowledge is assumed to be available at any one time. For example, a story might be:

> *John went to the store. He walked down the aisle and picked up the milk.*

The individual words in the story, such as "store" and "milk", activate the semantically-related plan of *going grocery shopping* in the system's memory, as well as related axioms about how shopping carts and check-out lines work. The system then tries to generate a resolution-style *proof* that the story occurs, where it is allowed to make two kinds of assumptions:

1. An instance of any active concept may be assumed to exist. Thus the theorem prover may simply assume $\exists x$. Shopping-Trip(x), if such an assumption is needed to close off a branch of the proof tree.

2. Any two terms may be assumed to be equal, if the system cannot prove (within some finite amount of time) that the terms are not equal. Thus the system may assume, for instance, that the go-to-store step of the Shopping-Trip event is equal to the particular instance of going to the store that is mentioned in the story.

[3]Eventually Peirce abandoned the notion of abduction [Hacking 83].

Charniak's framework is much less "cautious" than our own. If several different plans could entail the observations, it must choose a single one, whereas our framework would simply conclude the disjunction. It is also not entirely clear how well the technique of controlling inference by drawing activation from lexical items really works: it would be possible to "tune" such an activation network so that it activated just the formulas needed for any *particular* example. No semantic basis for the system is given, or indeed seems possible. Despite these criticisms, Charniak's work is among the most principled in the "scruffy" area of A.I.[4]

1.3.2. Discourse

1.3.2.1. Allen and 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. [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*, but not valid. The control strategy determines which of these inferences should actually be accepted. An example of these rules is *precondition/action* rule, which states that if (agent) H believes (agent) S wants proposition P to hold, then H may plausibly conclude that S wants action Act to be performed, given that P is a precondition[5] for Act. Another is the*effect/action* rule, that states if S wants P to hold, and P is achieved by Act, then S *may* want Act to occur.

[4] Workers in Artificial Intelligence are often divided into the neat and scruffy camps [Schank 83], with the neats trying to create formal theories [Kautz 86c] which systematize the heuristics uncovered by the intuition-driven scruffies.

precondition/action

H **Bel** S **Wants** P ⇒ H **Bel** S **Wants** Act
given that P is a precondition of Act

effect/action

S **Wants** P ⇒ S **Wants** Act
given that P is an effect of Act

Rules such as these were applied to form a chain of inference from a single observed action (called the *alternative*) to one of a number of possible contextually-dependent goals (called the *expectations*). Allen's system tried to find the "most likely" (or perhaps "most obvious") chain by performing a best-first search, numerically scoring chains of inference[6] by rules like the following:

(H1) Decrease the rating of a chain of inference if it contains an action whose preconditions are false at the time the action starts executing.

(H3) Increase the rating of a chain of inference if it contains descriptions of objects and relations in its alternative that are unifiable with objects and relations in its expectation.

One of the most interesting rating heuristics is applicable only in *communicative* domains, where the observer can assume that the actor is *trying* to make the intentions behinds his acts obvious. It says to *lower* the rating of a chain of inference if a great many different rules all apply to its last step. Thus ambiguity is penalized.

Allen did not try to provide any theoretical underpinning for his rules of plausible inference, in terms of probability or model theory. Nor did his system deal with multiple observations, which is the main focus of attention in our work (as well as that of [Litman 84], discussed in the next section). The basic notion of forming a chain connecting an observation to a goal seems related to our idea of a covering model, in which every event is part of some End (goal) event.

1.3.2.2. Extended Discourse

While a number of researchers [Sidner & Israel 81, Carberry 83, Grosz & Sidner 87] have begun to extend Allen's work to deal with sequences of utterances,

[6]Allen called such chains of inference "partial plans".

[Litman 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. 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). Litman's work is notable in providing a deterministic, highly constrained recognition algorithm, and in cleanly accounting for plan suspension and resumption.

The plan recognition algorithm constructs a *stack* of partially-recognized plans. When it observes an action, the system attempts to attach it somewhere on the stack, according to the following *preferences:*

1. Attach (as a substep) 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.

Attaching an observed act to a partial plan may require that certain equalities be *assumed* to hold between parameters of the act and the plan. Litman calls this *consistency unification.* A similar result is obtained in our framework by different means. As will be described in Chapter 4, the assumption that End *events* are equal can lead to the *conclusion* that certain parameters of substeps of the event are equal.

Both Litman's algorithm and our own (as will be seen in Chapter 7) rely on constraint propagation to eliminate alternative interpretations. Her system exemplifies a backtrack-free, incremental recognition process, as described in Chapter 5. Our "sticky" incremental recognition theory is a very crude approximation to Litman's.

A significant difference between her work and ours is our treatment of disjunction. Our theory justifies inference through disjunctions, allowing disjunctions to multiply[7] in the final conclusion. Litman argues that discourse should provide enough lin-

[7]Although in practice disjunctions often collapse again by abstraction, as will be seen.

guistic *clues* such as intonation, gesture, and keywords to eliminate most disjunctions. Following [Sidner 85], Litman claims that if the interpretation remains ambiguous, one should halt inference and simply wait for further utterances.

1.3.2.3. Cohen and Levesque

Recent work reported in [Cohen & Levesque 80, Cohen & Levesque 87] tries to derive speech act and discourse theory from general principles of rational interaction. The work axiomatizes (part of) the space of possible inferences available to an agent engaged in planning and plan recognition. They have not dealt with selecting between alternative interpretations of an observation.

Cohen & Levesque developed a version of dynamic logic [Harel 79] enriched with a logic of belief [Hintikka 62] to represent actions, beliefs, and intentions. Axioms in this language formally encode rules similar to those of Allen. For example, (one version of) the shared-recognition precondition/action axiom is

```
(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 (FINITELY-WAIT-FOR x q))))
        )
)
```

The formula (BMB y x ...) means that y believes that it is mutually believed between x and y that ... The axiom states that if y and x mutually believe that p enables q, and y expects that x will eventually want q, and y doesn't want not-q, and y is helpful to x, then *whatever* x does to make it mutually believed that x wants y to want p, will also make it mutually believed that x wants y to achieve q in the future (that is, x won't have to wait forever for q to occur).

Cohen & Levesque have concentrated on getting all the details of the representation of mental attitudes just right, and have deliberately ignored the issue of controlling inference. Our own work has gone to the other extreme, and completely ignored the representation of belief. Indeed, we have not distinguished between what the agent *wants* to come about and what *actually will* come about; we've assumed all plans are successful. On the other hand, our system tells us what conclusions are *justified*, rather than merely what conclusions are *possible* . (It is not entirely clear, in any case, what a "possible" conclusion is!) A sophisticated plan recognition system will eventually have to deal with the kinds of representational issues raised by Cohen & Levesque.

1.3.3. Intelligent Computer Environments

A natural application area for the discourse systems discussed above is the human/computer interface itself. 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.

1.3.3.1. The MACSYMA Advisor

One of the earliest automated consultants [Genesereth 79] helped people use MACSYMA, a powerful program for manipulating symbolic equations. Genesereth created a model, MUSER, of how a user typically breaks down a task when using MACSYMA. This model related the task, or plan, structure to the structure of the formulas being manipulated. Plans were represented as procedural nets [Sacerdoti 77], together with input/output links between various steps. The library contained 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. The advisor then built a possibly "buggy" plan graph which connected the two. The advisor then debugged the plan and told 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 applied deterministically: the partial plan was expanded only in unambiguous cases. An "escape hatch" existed in that the advisor would ask the user for clarification in case of ambiguity. 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 or concurrent multiple plans – issues central to our own work.

1.3.3.2. A Smart Operating System: Plan Parsing

An operating system consultant under development at the University of Massachusetts [Huff & Lesser 82] is notable for dealing with multiple concurrent plans, and for relating plan recognition to *parsing*.

The system tracks user's actions, and allows users to specify high-level commands which are disambiguated by context. A library of operating-system level tasks, such as *compile* or *edit,* is encoded by a set of grammatical rewrite rules, together with a list of *constraints*. The rules employ an extended version of regular expressions.[8] 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. 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 prefers the shuffle **eeeeffff** over **eeffeeff**, which is preferred over

[8]A regular expression is a string made up terminal and non-terminal symbols connected by the operators | (or), * (repeat 0 or more times), + (repeat 1 or more times), and blank (concatenation), possibly grouped by parentheses. A rewrite rule specifies that a non-terminal symbol on its left-hand side may be replaced by the expression on its right.

efefefef. Likewise for interleave, s is preferred over s$s, which is preferred over sss. 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.

Huff & Lesser's parsing heuristics yield conclusions similar to those obtained by the incremental recognition theories discussed in Chapter 5. The plan recognition algorithms described in Chapter 7 have a strong flavor of parsing, despite their origins in logical inference. Just as parsing can be viewed as logical inference[9], specialized inference can be cast as parsing. But there are at least two crucial differences between our work and Huff & Lesser's. First, we treat all temporal orderings between the components of an event as constraints. Any temporal relation may be specified: for example, one step may be during another, times may overlap, or one step may be required to occur *either* before *or* after another (without specifying which alternative). Second, our algorithms correspond to a particular model theory, while Huff & Lesser's are admittedly ad hoc. The inclusion of the shuffle operator, and the need to reach conclusions before the end of a sequence of observations, prevents Huff & Lesser from using any standard, well-understood parsing algorithms.

1.4. Related Work on Medical Diagnosis

One of the most active areas of applied research in A.I. has been in the area of medical diagnosis. Diagnosis and plan recognition are similar kinds of high-level recognition problems. (In fact, an early paper by [Pople 73], who was later to create the INTERNIST diagnostic system, explicitly made the connection.) In both cases one needs to find the best explanation for some phenomenon, using a hierarchically structured body of domain-specific knowledge. The vocabulary of events can be mapped to one appropriate for diagnosis in a straightforward way. (See Chapter 2 for the details of the Event vocabulary.) Events are replaced by *pathological states* of a patient. An *abstraction* hierarchy over pathological states is known as a *nosology*. The

[9]See any of the work on parsing with Prolog, such as [Pereira & Warren 80].

decomposition hierarchy corresponds to a *causation* hierarchy. If pathological state A always causes pathological state B, then B acts as a component of A. If only certain cases of A cause B, then one can introduce a specialization of A that has component B. The most basic specializations of End (unexplainable) events correspond to *specific disease entities*, while states which can be directly observed are *symptoms*. (Note that a symptom may also cause other other states: e.g., high blood pressure can be directly measured, and it can cause a heart attack.)

The pattern of inference in plan recognition and diagnosis is similar as well. Each symptom invokes a number of different diseases to consider, just as each observed event in our framework c-entails the disjunction of its uses. Once several findings are obtained, the diagnostician attempts to find a small set of diseases which accounts for, or covers, all the findings. This step corresponds to the minimization of End events in mc-entailment. A general medical diagnosis system must deal with patients suffering from multiple diseases; our plan recognition framework was designed to account for multiple concurrently executing plans. Finally, our work departs from previous work in plan recognition by explicitly dealing with disjunctive conclusions, which are winnowed down by obtaining more observations. These disjunctive sets correspond to the *differential diagnosis* sets which play a central role in medical reasoning.

There are some significant differences between our framework and those used in medical expert systems. In plan recognition it is necessary to distinguish the different roles that one event could play as a component of another, because a plan may have several different steps of the same type. One would need to distinguish the instance of the "meet with committee" event which *initiates* the plan to "obtain PhD degree" from the one which *terminates* it. Medical systems simply have undifferentiated "causes" links from diseases to symptoms. They make the assumption that a disease can't cause two different instances of the same symptom. In addition, the role of *time* is critical in plan recognition, but usually ignored in medical systems. But future medical systems may need to eliminate both differences. It certainly could be the case that a pathological state causes two different instances of the same *abstract* symptom. Thus AIDS often causes multiple different kinds of cancer in the same patient. [Patil, Szolovits, & Schwartz 82] suggests that diagnostic systems should incorporate a model of the time course of a disease, and the difference symptoms which are manifest at each stage.

Much of the research effort in medical expert systems has been on discovery procedures. The diagnostic system must take an active role, asking questions in order to narrow down the set of possible diseases. None of the work in plan recognition has dealt with this issue, yet it is critical if we are to build systems which can *actively* engage in a discourse or other task involving plan recognition.

1.4.1. INTERNIST and CADUCEUS

One of the best-known diagnostic systems is INTERNIST, an ambitious program which aimed to cover almost the entire field of internal medicine. The first version of INTERIST employed a causal hierarchy, and tried to chain from a group of observed symptoms to one or more specific disease entities. The succeeding versions of the program, INTERNIST-II and ultimately CADUCEUS, employed multiple abstraction (nosology) hierarchies as well. A detailed description of the project appears in [Pople 82].

A *task* for CADUCEUS was a data structure that explained a set of observations by causally linking them to one or more pathological states. The pathological states did not have to be ultimate, specific disease entities; they could be any states within the taxonomy. The task could contain certain kinds of *disjunctions*. For example, a task could link symptom A to causes B, C, and D, meaning that *either* B caused A, *or* C caused A, *or* D caused A. Such a disjunction was called a *differential diagnosis set.* The differential diagnosis sets could *collapse* within the task. For example, all of B, C, and D could be linked by a *subclassification* arc to a more general disease type E.

CADUCEUS created a task for each observed symptom, and then performed a heuristic search to attempt to combine the tasks into a smaller set. During the search, CADUCEUS could interactively prompt for information that could would be used to reduce differential diagnosis sets. An example of a operator that CADUCEUS could apply to two tasks in order to combine them is *Combined Causal/Subclassification:*[10]

```
Where P and Q are descriptors in tasks $T_1$ and $T_2$ respectively:
    IF P is (directly) caused by $S_1$ or $S_2$ or ... $S_n$ AND
        $S_1$, $S_2$, ... $S_n$ all (directly) sub-classify Q THEN
            Link $T_1$ and $T_2$ by adding caused-by arcs from P
            to each of $S_1$, $S_2$, ... $S_n$, and sub-classification arcs
            from each of $S_1$, $S_2$, ... $S_n$ to Q
```

In Chapter 7, our algorithm for plan recognition employs a data structure called an "e-graph", which links a set of observed events to some End event. E-graphs are

[10]The notation here is our own, not Pople's.

similar in structure to CADUCEUS's tasks. But because our e-graphs always contain an End node, two e-graphs can be combined by simply performing a top-down match starting at the End nodes. The resulting e-graph encodes *all* possibly ways of combining the original graphs. CADUCEUS performs only a heuristic search, and therefore could not guarantee completeness. There is no straightforward way to tell when all possible combinations have been found, or whether the heuristics need to run longer. The particular set of combining operators are (Pople admits) rather arbitrary and incomplete.

An interesting feature incorporated in CADUCEUS that our framework lacks are *planning links*, which connect one state to another if *some* specialization of the latter can cause the former. [Pople 82] shows how such links can be very useful for heuristically guiding search.

1.4.2. A Set Covering Model

Recent work by [Reggia, Nau, & Wang 83] has proposed that medical diagnosis can be viewed as a set covering problem. Each disease corresponds to the set of its symptoms, and the diagnostic task is to find a minimum cover of a set of observed symptoms.

While this idea has long been implicit in diagnostic systems such as INTERNIST, Reggia & Nau's work is unique in providing a provably correct and complete algorithm which not only finds all minimum covers, but also indicates when and how the diagnostician should request additional findings. The system is particularly strong in dealing with multiple simultaneous diseases.

Reggia & Nau have not yet expanded the work to cover many-level causal or abstraction hierarchies, although they mention that as a topic for future research. The lack of abstraction lets them stay within a purely propositional framework, which greatly simplifies the algorithms. Once causal and abstraction hierarchies are added, their framework and algorithms may prove to be very similar to our own.

1.5. Related Work on Non-Deductive Inference

Most recognition problems cannot be formalized entirely within a deductive system – unless one uses it as a meta-language to talk *about* a different, non-deductive system.[11] This section reviews the major formal systems of non-deductive inference used within A.I., concluding with *circumscription*, the system most closely related to our own work.

1.5.1. Probability Theory

Probability theory provides the basis for non-deductive inference in practically all the sciences *outside* of A.I.. Its success should at least cause the computer scientist to hesitate a bit before devising a new sort of theory. Indeed, a growing number of researchers are basing systems for such tasks as vision or medical diagnosis on classical, Bayesian, or Dempster-Shafer[12] theories of probability. In this section we will describe (part of) plan recognition in terms of very elementary statistics. Our 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 also addresses issues about which classical probability theory has little to say: most importantly, how hypotheses should be *constructed,* and when a likely statement should be *accepted* as fact.

Let A_1, A_2, etc. be various directly-observable and executable actions. A countably infinite number of plans, labeled P_1, P_2, etc. can be constructed from these actions. Each *basic hypothesis* B_i is of the form, "the agent intends to perform plan P_i." A *composite hypothesis* **H** is a boolean combination of basic hypotheses. For now, we will only consider composite hypotheses which are *conjunctions* of basic hypotheses.

What is the general form of a plan recognition problem? One might try to determine the most likely composite hypothesis. There is only a trivial answer to this problem: the empty hypothesis, that makes no assertion about the world. We cannot state a plan recognition problem as a crisp decision problem. The best one can do is to try to determine the sets of plans (composite hypotheses) which are "most likely" on the basis of the observed actions. (Later on we'll mention some of the problems in determining the exact point at which a hypothesis is likely enough to be accepted.)

Where P is the probability function, and **A** is the set of observed actions, the goal is find those **H** for which P(**H** | **A**) is large. By Bayes' theorem,

[11] For example, [Kyburg 74] uses first-order logic as a meta-language to axiomatize a system of probabilistic inference.
[12][Dempster & Shafer 76]

$$P(\mathbf{H} \mid \mathbf{A}) = \frac{P(\mathbf{A} \mid \mathbf{H})\, P(\mathbf{H})}{P(\mathbf{A})}$$

Given a fixed set of observations **A** and certain simplifying assumptions, it is possible to determine the *relative* probabilities of various candidate hypotheses. Assume that every plan (every B_i) has the same prior probability, and exactly one possible *decomposition* (that is, a breakdown into actions). Begin by considering the case where the actor performs exactly one plan -- that is, the B_i are disjoint. When calculating relative probabilities, the factors P(**A**) and P(**H**) can be ignored; the first because **A** is fixed, the second because every candidate **H** must be a basic hypothesis. The conditional probability of the observations given the hypothesis is simply:

$$P(\mathbf{A} \mid \mathbf{H}) = 1 \text{ if } \mathbf{H} \equiv B_i \wedge (\forall\ A_j \in \mathbf{A})\ .\ \text{component-of}(A_j, P_i)$$

$$= 0 \text{ otherwise}$$

So the most likely hypotheses are simply those which incorporate all the observations. 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** 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 described in this thesis corresponds to this search.)

Relax the assumption that the B_i are disjoint. Now it is harder to compute relative values for P(**H** | **A**). First, it is no longer reasonable to assign the same value to all P(**H**) – particularly since some composite hypotheses may strictly entail other composite hypotheses. Second, the conditional probability of the observed actions given the candidate hypothesis is much less constrained. Without making additional assumptions, all one may conclude is that

$$P(\mathbf{A} \mid \mathbf{H}) = 1 \text{ if } (\forall\ A_j \in \mathbf{A})\ (\exists\ B_i\ ).\ \mathbf{H}\ \vdash B_i \wedge \text{component-of}(A_j, P_i)$$

$$< 1 \text{ otherwise}$$

If one is comfortable assuming that the B_i are *independent*, so that

$$P(B_1 \wedge B_2 \wedge \ldots \wedge B_n) = P(B_1)\, P(B_2) \ldots P(B_n)$$

then note the following: if hypotheses **H** and **H'** contain the same number of conjuncts, 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 hypothesis has the higher probability. Our framework for plan recognition has similar characteristics.

We have suggested that in plan recognition it is appropriate to invoke Occam'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 *quite* fall out of the probabilistic account so far, even assuming the independence of the basic hypotheses: 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 partially remedied by constraining the conditional probability function P(**A** | **H**) to be very small for such an **H**.

$$P(\mathbf{A} \mid \mathbf{H}) \ll P(B_i)$$
$$\text{if } (\exists\, A_j \in \mathbf{A})\ (\forall\, B_i).\ \mathbf{H} \vdash B_i \supset \neg\ \text{component-of}(A_j, P_i)$$

A further complication awaits in the wings: it is not adequate to only consider conjunctive composite hypotheses. Much of this thesis will be concerned with abstract event (plan) types. An abstract plan stands for the class of its specializations; that is, it is logically equivalent to the *disjunction* of all the different ways it which it can be fully decomposed with constants assigned to all its parameters. The likelihood that the agent is performing an abstract plan is the probability assigned to the corresponding composite hypothesis, which may be equivalent to an *arbitrary boolean combination* of basic hypotheses. This more general formulation compounds the size of space of hypotheses.

In the end, then, probability gives a way to compare the relative merits of alternative hypotheses about an actors intentions, but does not automatically yield a way of *generating* or *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 our 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, one needs 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.* [Kyburg 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.

1.5.2. Default Logic

Much everyday reasoning employs "default rules", that allow conclusions to be drawn if there is no evidence to the contrary. For example, if all one knows about Tweety is that he is a bird, then one may conclude *by default* that Tweety can fly. The additional information that Tweety is a penguin and that penguins don't fly would disallow the previous conclusion. [Reiter 80] developed an extension to first-order logic, called *default logic*, to handle this kind of inference.

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, one could write, for example, the *precondition/action* rule as:

$$\frac{\text{Want(agt, P)} \wedge \text{precondition-of(P, act)} \;:\; \mathbf{M}\ \text{Want(agt, act)}}{\text{Want(agt, act)}}$$

This says that if an agent wants a precondition of an act to hold, and it is consistent that (**M**) he wants the act, then conclude that he wants the act. But little has been gained from the use of default logic. Default logic has 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 Cohen & Levesque: too much of the problem remains hidden in the strategy which orders the application of various rules of inference.

1.5.3. Circumscription

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, one might have a description of a bunch of blocks on a table, and want to synthesize 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 one cannot *prove* that there is anything on top of B, then, in general, one wants to be able to conclude, by circumscribing the predicate **on**, that *nothing* is on top of B.

The *circumscription* of a predicate (or set of predicates) relative to a body of knowledge is a sentence of second-order logic which involves the entire collection of facts at hand. The following formulation is drawn from [Lifschitz 84]. Let $S[\pi]$ be a sentence containing the list of predicates π. The expression $S[\sigma]$ is the sentence obtained by rewriting S with each member of π replaced by the corresponding member of σ. The expression $\sigma \leq \pi$ abbreviates the formula stating that the extension of each predicate in σ is a subset of the extension of the corresponding predicate in π; that is

$$(\forall x \,.\, \sigma_1(x) \supset \pi_1(x)) \wedge \ldots \wedge (\forall x \,.\, \sigma_n(x) \supset \pi_n(x))$$

where each x is a list of variables of the proper arity to serve as arguments to each σ_i. The circumscription of π relative to S, written Circum($S[\pi],\pi$), is the second-order formula

$$S[\pi] \wedge \forall\, \sigma \,.\, (S[\sigma] \wedge \sigma \leq \pi) \supset \pi \leq \sigma$$

Chapter 4 shows how this formula is generalized to allow other symbols to *vary* during the minimization, as described in [McCarthy 84].

Consider the following example. Let $S[p] = p(A) \vee p(B)$, and compute the circumscription of p relative to S. This is:

$$(p(A) \vee p(B)) \wedge \forall\, \sigma \,.\, ((\sigma(A) \vee \sigma(B)) \wedge \sigma \leq p) \supset p \leq \sigma$$

Expanding the $\leq$ abbreviation gives

$$\begin{aligned}&(p(A) \vee p(B)) \wedge \\ &\forall\, \sigma \,.\, ((\sigma(A) \vee \sigma(B)) \wedge \forall x \,.\, \sigma(x) \supset p(x)) \supset \\ &\qquad\qquad \forall x \,.\, p(x) \supset \sigma(x)\end{aligned}$$

Now let us investigate what can be concluded from this statement. Instantiate the predicate variable σ with the lambda-expression $\lambda(x) \,.\, x{=}A$. (That is, σ is the predicate which is true of A and nothing else.) This yields

$$(p(A) \vee p(B)) \wedge$$
$$((A=A) \vee A=B)) \wedge \forall x \,.\, x=A \supset p(x)) \supset$$
$$\forall x \,.\, p(x) \supset x=A$$

This statement reduces to

$$(p(A) \vee p(B)) \wedge$$
$$p(A) \supset (\forall x \,.\, p(x) \supset x=A)$$

Instantiate the circumscription formula a second time with the expression $\lambda(x).x=B$. The result is

$$(p(A) \vee p(B)) \wedge$$
$$p(B) \supset (\forall x \,.\, p(x) \supset x=B)$$

Thus the final conclusion is that one of A or B is a p, and there is only one p thing. (The conclusion is not p(A) exclusive-or p(B) because of the possibility that A=B.)

$$(\forall x \,.\, p(x) \supset x=A) \vee (\forall x \,.\, p(x) \supset x=B)$$

This example illustrates the major stumbling block to the use of circumscription. There is no general *mechanical* way of determining how to instantiate the predicate parameters in the second-order formula. The work in this thesis overcomes this problem for a particular class of circumscriptions, and gives simple syntactic rules for computing a *first-order* version of the circumscription formula.

A *model* of a set of statements is *minimal* in π if there is no other model of those statements which is identical, except that π holds of some things in the first model but not in the second.[13] Section 3.1 formally defines this notion. [Etherington 86] includes proofs that the circumscription of a predicate relative to a formula is true in all models minimal in the predicate. The story of the completeness of circumscription relative to this model theory is a bit complicated. The notion of a minimal model is powerful enough to capture the standard model of arithmetic, which cannot be axiomatized [Davis 80]. While all *strong* second-order models of the circumscription formula are

[13]McCarthy's original version of circumscription did not make reference to a particular set of predicates to be minimized. Instead, the entire *domain* of the model was minimized; thus, a minimal model of formula was one which had no submodels of the formula. [Etherington 86] calls this version of circumscription *domain circumscription*, and has shown that it is *not* comparable in expressive power to predicate circumscription. In this thesis, the expression "minimal model" always means "minimal in some predicate".

minimal models, it is not necessarily the case that all *weak* second-order models are minimal. Another way of stating this is that the circumscription formula is *not* always complete if one views it as a *schema* for recursively generating a set of first-order statements.[14] [Minker & Perlis 85] show that circumscription *is* complete in this sense if in all models the circumscribed predicate has a finite extension. Chapter 3 presents a proof that formulas computed by our closure function **cl** *are complete* for the minimal model semantics.

[14]McCarthy's early versions of circumscription viewed the formula in just this way, as a first-order schema, rather than as a true second-order statement.

Chapter 2
Representing Event Hierarchies

2.1. Language

The representation language is first-order predicate calculus with equality. A model provides an interpretation of the language, mapping terms to individuals, functions to mappings from tuples of individuals to individuals, and predicates to sets of tuples of individuals. If M is a model, then this mapping can be made explicit by applying M to a term, function, or predicate. For example, for any model M:

Loves(Sister(Joe),Bill) is true in M if and only if
(M[Sister](M[Joe]), M[Bill]) $\in$ M[Loves]

Meta-variables (not part of the language) which stand for domain individuals begin with a colon. Thus, one may write:

Let :C be an event token in Domain(M).

Models map free variables in sentences to individuals. We write M{x/:C} to mean the model which is just like M, except that variable x is mapped to individual :C. Quantification is defined as follows:

$\exists$x . p is true in M if and only if
there exists :C $\in$ Domain(M) such that p is true in M{x/:C}
$\forall$x . p is true in M if and only if
$\neg\exists$x . $\neg$p is true in M

The propositional connectives are semantically interpreted in the usual way.

Certain finite models can be completely specified by listing all the positive atoms which hold in the model, other than trivial instances of equality (i.e., of the form a=a). For example, a model in which the only predicate with a non-empty extension is Fat, which has two elements, may be specified by writing:

{ Fat(A), Fat(B) }

FOL proofs in this thesis use natural deduction, freely appealing to obvious lemmas and transformations. It is convenient to distinguish a set of constant symbols called *individual parameters*, or just parameters, for use in the deductive rule of existential elimination. The rule allows one to replace an existentially-quantified variable by a parameter which appears at no earlier point in the proof. Parameters are distinguished by the prefix *. Technically, no parameters may appear in the final step of the proof: they must be replaced again by existentially-quantified variables (or eliminated by other means). This final step is omitted when it is obvious how it should be done. For example, we may write

The system concludes $E1(*C) \vee E2(*C)$.

instead of

The system concludes $\exists x \,.\, E1(x) \vee E2(x)$.

2.2. Representation of Time, Properties, and Events

Most formal work on representing action has relied on the situation calculus or dynamic logic [Harel 79]. While these formalisms are convenient for planning, they prove awkward for plan recognition: it is impossible (without extreme convolutions; see [Cohen 84]) to state that some particular action *actually occurred* at a particular time. We therefore adopt a "reified" representation of time and events.

Time is linear, and time *intervals* are individuals, each pair related by one of Allen's interval logic relations: Before, Meets, Overlaps, etc. [Allen 83b]. The names of several relations may be written in place of a predicate, in order to stand for the disjunction of those relations, each applied to the same argument pair. For example, the expression

BeforeMeets(T1,T2)

is an abbreviation for the formula

$\text{Before}(T1,T2) \vee \text{Meets}(T1,T2)$

As a special case, the predicates **Within** and **Disjoint** abbreviate common disjunctions:

$$\text{Within}(T1,T2) \equiv \text{Starts}(T1,T2) \vee \text{During}(T1,T2) \vee \text{Finishes}(T1,T2)$$

Disjoint(T1,T2) $\equiv$

Before(T1,T2) $\vee$ Meets(T1,T2) $\vee$ MetBy(T1,T2) $\vee$ After(T1,T2)

Event *tokens* are also individuals, and event *types* are represented by unary predicates. All event tokens are real; there are no imaginary or "possible" event tokens. Various functions on event tokens, called *roles*, yield parameters of the event. Role functions include the event's *agent* and *time*. For example, the formula

ReadBook(C) $\wedge$ agent(C)=Fred $\wedge$ time(C)=T2

may be used to used to represent the fact that an instance of booking reading occurs; the agent of the reading is Fred; and the time of the reading is (the interval) T2.

The reader may wonder why we have not used functions to *construct* event tokens from their parameters. For instance, one might want to specify the above action as:

ReadBook(Fred, T2)

Indeed, many researchers have taken the line that an event type, together with its parameters and time of occurrence completely specifies an event token [Pollack 86, Allen 84]. There are two objections to this alternative. First is that it is not always possible to specify *all* the parameters of an event beforehand. Should ReadBook also have a parameter for the thing being read? What about whether the agent is skimming or reading carefully? And so on.[1] A second problem occurs when we allow very abstract event types, like "ReadSomething", which holds of *any* reading event, or "GainInformation", which holds of any instance of reading, watching television, listening to the radio, etc. It is possible for several *distinct* instances of the same abstract event type to occur *simultaneously*. Thus it is not sufficient to only represent event types and times; explicit event tokens are needed as well.

Timeless propositions are called *facts*. Facts are represented by ordinary predicates. For example, the fact that John is a human may be represented by the formula

Human(John)

The predicate Holds relates a time-dependent *property* and a time interval over which the property holds. Properties are individuals, and are therefore represented by terms. For example, the fact that John is unhappy at time T1 may be represented by the for-

[1]This is the same problem people working on case grammar [Fillmore 68] ran into.

mula

Holds(unhappy(John),T1)

The term unhappy(John) denotes the proposition that John is unhappy. Every distinct property term represents a distinct property. This is enforced by a set of *property identity axioms.* The axioms take the following form.

For all distinct $\rho,\sigma \in$ Property Terms:

$$\forall x_1,\ldots x_n \,.\, \rho(x_1,\ldots x_n) \neq \sigma(x_1,\ldots x_n)$$

$$\forall x_1,\ldots ,x_n, y_1, \ldots y_n \,.\, \rho(x_1,\ldots x_n) = \rho(y_1,\ldots y_n) \supset x_1=y_1 \wedge \ldots \wedge x_n=y_n$$

All properties are dense: if one holds over an interval, then it holds over all subintervals of that interval. The predicate Never holds of a property and a time when the property holds over no subinterval of the time. The following axioms capture these constraints:

$$\forall p,t_1,t_2 \,.\, \mathrm{Holds}(p,t_1) \wedge \mathrm{Within}(t_2,t_1) \supset \mathrm{Holds}(p,t_2)$$

$$\forall p,t_1 \,.\, \mathrm{Never}(p,t_1) \equiv (\forall t_2 \,.\, \mathrm{Holds}(p,t_2) \supset \mathrm{Disjoint}(t_1,t_2))$$

In addition to the relations between individuals, it is sometimes convenient to talk about relations between predicates. These meta-relations, such as "abstracts" and "component", are not part of the logic itself.

2.3. The Event Hierarchy

An event hierarchy is a collection of restricted-form axioms, and may be viewed as a logical encoding of a *semantic network,* as in [Hayes 85]. These axioms represent the abstraction and decomposition relations between event types. An event hierarchy H contains the following parts, H_E, H_A, H_{EB}, H_D, and H_G:

• H_E is the set of unary event type predicates. H_E contains the distinguished predicates AnyEvent and End.

• H_A is the set of abstraction axioms, each of the form:

$$\forall x \,.\, E_1(x) \supset E_2(x)$$

for some $E_1, E_2 \in H_E$. In this case we say that E_2 *directly abstracts* E_1. The transitive closure of direct abstraction is abstraction; and the fact that E_2 is the same as or abstracts E_1 is written E_2 abstracts* E_1. AnyEvent abstracts* all event types.

• H_{EB} is the set of basic event type predicates, those members of H_E which do not abstract any other event type.

• H_D is the set of decomposition axioms, each of the form:

$$\forall x \,.\, E_0(x) \supset E_1(f_1(x)) \wedge E_2(f_2(x)) \wedge \ldots \wedge E_n(f_n(x)) \wedge \kappa$$

where $E_0, \ldots, E_n \in H_E$; $f_1, \ldots, f_n$ are role functions; and κ is a subformula containing no member of H_E. The formula κ describes the *constraints* on E_0. E_1 through E_n are called *direct components* of E_0. Sometimes we refer to a component by the name of its role function; e.g., E_1 is the f_1 direct-component of E_0. The type End never appears as a direct component of another type; nor does any type which End abstracts.

• H_G is the set of general axioms, those which do not contain any member of H_E. H_G includes the axioms for the temporal interval relations; the density axioms for Holds and Never; the property identity axioms; as well as any other facts not specifically relating to events.

2.4. Components of Event Tokens

The component relation may be applied to event tokens in a model M as follows. Suppose $:C_i$ and $:C_0$ are event tokens. Then $:C_i$ is a direct component of $:C_0$ in M if and only if

(i) there are event types E_i and E_0, such that $:C_i \in M[E_i]$ and $:C_0 \in M[E_0]$

(ii) H_D contains an axiom of the form:

$$\forall x \,.\, E_0(x) \supset E_1(f_1(x)) \wedge \ldots \wedge E_i(f_i(x)) \wedge \ldots \wedge E_n(f_n(x)) \wedge \kappa$$

(iii) $:C_i=M[f_i](:C_0)$

The component relation is the transitive closure of the direct component relation, and the fact that $:C_n$ is either the same as or a component of $:C_0$ is written $:C_n$ is a component* of $:C_0$.

Note that the component relation over event tokens does not correspond to the transitive closure of the direct-component meta-relation over event types. This is due to the fact that a token may be of more than one type.

2.5. Acyclic Hierarchies & Compatible Types

An *acyclic* hierarchy is one that can be exhaustively searched in finite time, and is formally defined as follows. Two event predicates E_1 and E_2 are *compatible* if there is an event type E_3 such that both E_1 and E_2 abstract* E_3. A hierarchy if acyclic if it contains no series of event predicates $E_1, E_2, \ldots, E_n$ such that:

(i) E_i is a direct component of E_{i+1} for odd i, $1 \leq i \leq n-1$

(ii) E_j is compatible with E_{j+1} for even j, $2 \leq j \leq n-2$

(iii) E_n is compatible with E_1

We will consider only acyclic hierarchies in this thesis, although most results should extend to cyclic hierarchies as well.

Roughly speaking, a lexical hierarchy is cyclic if an event token may have an event of the same type as a component. The presence of the abstraction hierarchy, however, makes this definition too generally applicable. All events are of type AnyEvent; therefore any event token will share at least the type AnyEvent with its components. The previous definition avoids this problem by consider only the meta-structure between event type predicates.

2.6. Example: The Cooking World

The actions involved in cooking form a interesting yet tractable domain for planning and plan recognition. The specialization relations between various kinds of foods are mirrored by specialization relations between the actions which create those foods. Decompositions are associated with the act of preparing a type of food, in the manner in which a recipe spells out the steps in the food's preparation. A good cook stores information at various levels in his or her abstraction hierarchy. For example, one knows certain actions which are needed to create any cream-based sauce, as well as certain conditions (constraints) which must hold during the preparation. The sauce must be stirred constantly, the heat must be moderate, and so on. A specialization of the type cream-sauce, such an Alfredo sauce, adds steps and constraints: e.g., one should slowly stir in grated cheese at a certain point in the recipe.

The cook and the observer have the same knowledge of cooking, a hierarchically-arranged cookbook. (Most real cookbooks are "flat", of course; years of experience are required to induce the hierarchical structure.) Actions of the cook are reported to the observer, who tries to infer what the cook is making. We do not assume that the reports are exhaustive – there may be unobserved actions – although such an assumption could be made, without changing our framework: we would simply add additional observational reports, of the form "nothing else happened" over certain time periods. A cook may prepare several different dishes at the same time, so it is not always possible to assume that all observations are part of the same recipe. Different End events may share steps. For example, the cook may prepare a large batch of tomato sauce, and then use the sauce in two different dishes.

We do *not* aim to capture a lot of domain-specific information about cooking: rather, cooking appears to be one of the most general "toy worlds" one can consider. (For interesting work on *planning* in the cooking world, see [Schmolze 86].) Techniques which work in the cooking world should translate to almost any other domain.

2.6.1. Diagrammatic Form

The following diagram illustrates a very tiny cooking hierarchy. Thick grey arrows denote the abstraction meta-relation, while thin black arrows denote the direct component meta-relation. All event types are abstracted by AnyEvent. As discussed above, Ends are a special kind of event, which are not components of any other event. Here there are two main categories of End events: preparing meals and washing dishes. It is important to understand that the abstraction hierarchy, encoded by the axioms in H_A, and the decomposition hierarchy, encoded by the axioms in H_D, are interrelated but separate. Work on hierarchical planning often confuses these two distinct notions in an action or event hierarchy.

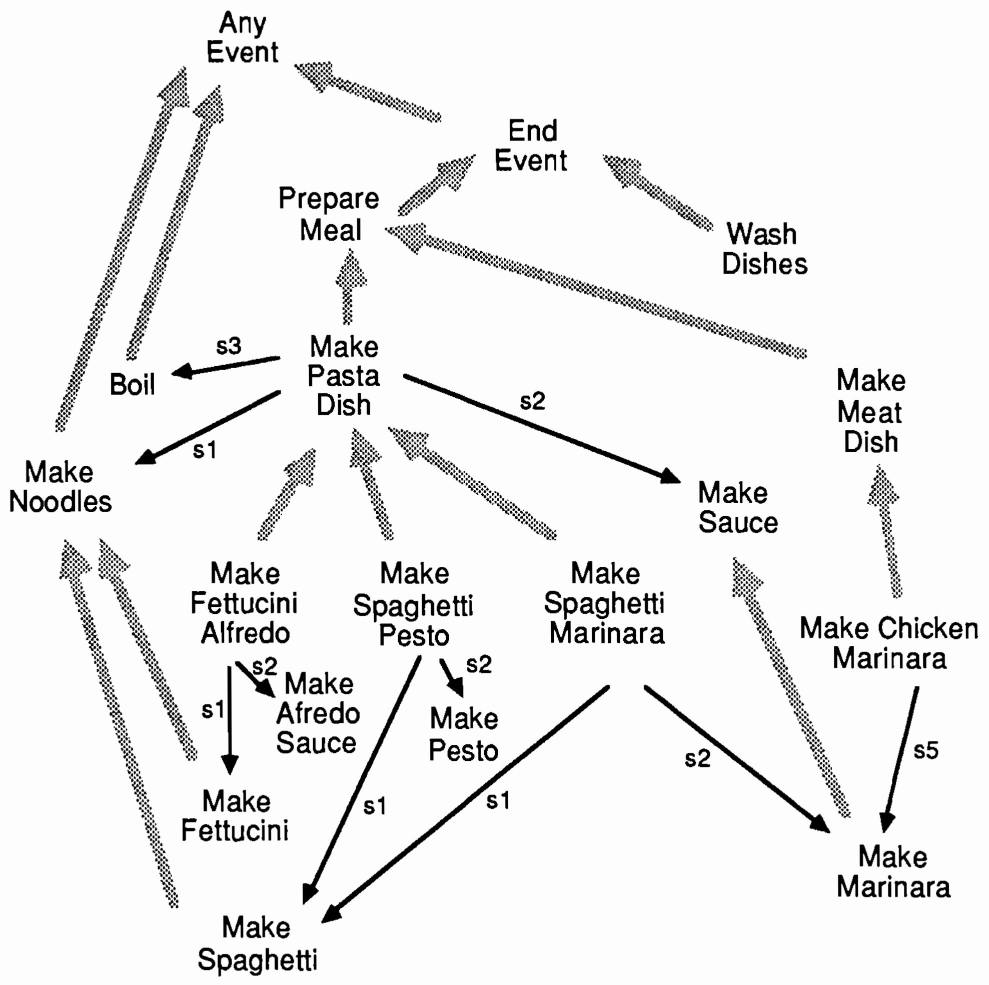


figure 2.1: Cooking Hierarchy

2.6.2. The Abstraction Hierarchy

The diagram suggests some of the elements which make up the lexical hierarchy.

• The set of event types, H_E, includes PrepareMeal, MakeNoodles, MakeFettucini, and so on.

• The abstraction axioms, H_A, assert that every MakeSpaghetti and every MakeFettucini is also a MakeNoodles, that every MakePastaDish is also a PrepareMeal, and so on. A traditional planning system might call MakeSpaghetti and MakeFettucini different *bodies* of the MakeNoodles plan.

$$\forall x . \text{MakeSpaghetti}(x) \supset \text{MakeNoodles}(x)$$

$$\forall x . \text{MakeFettucini}(x) \supset \text{MakeNoodles}(x)$$

$$\forall x . \text{MakePastaDish}(x) \supset \text{PrepareMeal}(x)$$

• The basic event types, H_{EB}, appear at the bottom of the abstraction (grey) hierarchy. These include the types WashDishes, Boil, MakeSpaghettiMarinara, MakeChickenMarinara, MakeFettucini, MakeSpaghetti, and MakeMarinara. Note that basic event types may have components (but no specializations).

2.6.3. The Decomposition Hierarchy

• The decomposition axioms, H_D, include much information which does not appear in the diagram. These axioms specify the role-functions which link an event to its components, and the constraints which hold between those steps and the event. Following is (an abbreviated version of) the decomposition axiom for the MakePastaDish event. This act includes at least three steps: making noodles, making sauce, and boiling the the noodles. The equality constraints assert, among other things, that the agent of each step is the same as the agent of the overall act; and that the noodles the agent makes (specified by the **result** role function applied to the MakeNoodles step) are the thing boiled (specified by the **input** role function applied to the Boil step). Temporal constraints explicitly state the temporal relations between the steps and the MakePastaDish. For example, the time of each step is during the time of the MakePastaDish, and the Boil must follow the MakeNoodles. The constraints in the decomposition include the preconditions and effects of the events. Preconditions for MakePastaDish include that the agent is in the kitchen during the event, and that the agent is dexterous (making pasta by hand is no mean feat!). An effect of the event is that there exists something which is a PastaDish, the **result** of the event, which is ready to eat during a time period **postTime**, which immediately follows the time of the cooking event.

∀x . MakePastaDish(x) ⊃

	MakeNoodles(step1(x)) ∧
Components	MakeSauce(step2(x)) ∧
	Boil(step3(x)) ∧
Equality	agent(step1(x)) = agent(x) ∧
Constraints	result(step1(x)) = input(step3(x)) ∧
Temporal	During(time(step1(x)), time(x)) ∧
Constraints	BeforeMeets(time(step1(x)), time(step3(x))) ∧
	Overlaps(time(x), postTime(x)) ∧
Preconditions	Holds(inKitchen(agent(x)), time(x)) ∧
	Dexterous(agent(x)) ∧
Effects	Holds(readyToEat(result(x)), postTime(x)) ∧
	PastaDish(result(x))

Note that the names of the component roles, step1, step2, etc., are arbitrary; they do *not* indicate temporal ordering. The event types which specialize MakePastaDish add additional constraints and steps to its decomposition. For example, the event type MakeSpaghettiMarinara further constrains its decomposition to include MakeSpaghetti (rather than the more generic MakeNoodles) and MakeMarinaraSauce (rather than simply MakeSauce). One could also add completely new steps as well.

∀x . MakeSpaghettiMarinara(x) ⊃
MakeSpaghetti(step1(x)) ∧
MakeMarinaraSauce(step2(x)) ∧ ...

2.6.4. Describing Instances of Events

Assertions about particular event instances take the form of the predication of an event type of a constant, conjoined with equality assertions about the roles of the event token, and perhaps a proposition relating the time of the event to that of other events. The English statement, "Joe made the noodles on the table yesterday" may be represented as follows:

MakeNoodle(Make33) ∧
agent(Make33) = Joe ∧
result(Make33) = Noodles72 ∧
Holds(onTable(Noodles72), Tnow) ∧
During(time(Make33), Tyesterday)

2.7. Conditional Actions

Plans of action often contain conditional actions. If some condition holds, then the plan (or event type) contains a certain step; otherwise, it contains a different step. At first glance, the form of a lexical hierarchy does not seem to allow for the presence of conditional steps in the decomposition of an event type. Consider the case where if a condition P holds, event type E0 should contain a step of type E_T; otherwise, it contains one of type E_F. One may want to write something like the following.

$$\begin{aligned} \forall x \,.\, E0(x) \supset \quad & (P \supset (E_T(s_T(x)) \wedge \kappa_T)) \wedge \\ & (\neg P \supset (E_F(s_F(x)) \wedge \kappa_F)) \wedge \\ & E2(s2(x)) \wedge \ldots \wedge \kappa \end{aligned}$$

The expressions κ_T and κ_F introduce constraints which only hold on the conditional part of the event. This formula is clearly not of the form specified for axioms in H_D. Does this mean that all the work in this thesis that depends on lexical hierarchies must be redone in order to account for conditional events?

2.7.1. Representing Conditional Actions

Fortunately, the answer is no. The form of a lexical hierarchy is sufficiently general to accommodate conditional events, through the introduction of additional abstract event types. Consider the case above. There are two clear ways of specializing the event type E0. In the first category, P holds, and E0 contains the component E_T. In the second category, P does not hold, and E0 contains the component E_F. This situation is captured by the following axioms.

$$\forall x \,.\, E0(x) \supset \quad E2(s2(x)) \wedge \ldots \wedge \kappa$$

$$\forall x \,.\, E0_T(x) \supset \quad E0(x)$$

$$\forall x \,.\, E0_F(x) \supset \quad E0(x)$$

$$\forall x \,.\, E0_T(x) \supset \quad E_T(s_T(x)) \wedge \kappa_T \wedge P$$

$$\forall x \,.\, E0_F(x) \supset \quad E_F(s_F(x)) \wedge \kappa_F \wedge \neg P$$

It is plain that these axioms are of the proper form to appear in H_A and H_D. The process that transformed the first statement into the second set of statements could be repeated, in case E0 contains more than one conditional expression. In fact, this forms a proof that conditional statements can always be "factored out" of decomposition axioms.

This is not the only possible way of representing conditionals in a lexical hierarchy. Another method would be to replace the conditional expression in the decomposition of E0 by a new event type, and then give two specializations for that event type. The result could be as follows.

$$\begin{aligned} \forall x \,.\, E0(x) \supset\ & E_P(s1(x)) \wedge \kappa_P \wedge \\ & E2(s2(x)) \wedge \ldots \wedge \kappa \end{aligned}$$

$$\forall x \,.\, E_{PT}(x) \supset E_P(x)$$

$$\forall x \,.\, E_{PF}(x) \supset E_P(x)$$

$$\forall x \,.\, E_{PT}(x) \supset E_T(s_T(x)) \wedge \kappa_T \wedge P$$

$$\forall x \,.\, E_{PF}(x) \supset E_F(s_F(x)) \wedge \kappa_F \wedge \neg P$$

The new constraint expression κ_P in the first axiom makes s1(x) have all the same roles as x, except for the new role s1 itself. This is necessary so the constraint expressions κ_T and κ_F in the final two axioms properly constrain $s_T(x)$ and $s_F(x)$. The advantage of this method is that it results in fewer axioms if the initial decomposition axiom contains many conditional expressions. Where n is the number of conditionals, the former method could introduce as many as $O(2^n)$ axioms, while the latter method only introduces O(n) axioms.

Some question might arise in the case where half the conditional is empty. Suppose that a certain action had to be performed only if P held:

$$\begin{aligned} \forall x \,.\, E0(x) \supset\ & (P \supset (E_T(s_T(x)) \wedge \kappa_T)) \wedge \\ & E2(s2(x)) \wedge \ldots \wedge \kappa \end{aligned}$$

Either transformation goes through as before, with the elimination of the atom $E_F(s_F(x))$. One of the axioms generated by the second method, however, looks a bit odd. The decomposition axiom for the event E_{PF} contains no event-types in its consequence:

$$\forall x \,.\, E_{PF}(x) \supset \neg P$$

What is intuitive meaning of the event type E_{PF}? It is a kind of "null event". One will never directly observe an instance of E_{PF}; however, from $\exists x.E0(x)$ and $\neg P$ one can deduce that E_{PF} occurred. Unlike some accounts of null actions, we do not insist that a null action occurs over all time periods in which the world is unchanged.

2.7.2. An Example

The PickUp action in the cooking world depends critically upon the temperature of the object being picked up. If the object is hot, the agent must wear a mit to avoid being burned. Therefore the plan for picking up an object must begin with the action of conditionally putting on a mit if the object is hot. This can be encoded in an event hierarchy as follows.

$$\begin{aligned}&\forall x \,.\, \text{PickUp}(x) \supset \\ &\qquad \text{Grasp}(s2(x)) \wedge \text{object}(x)=\text{object}(s2(x)) \wedge \ldots\end{aligned}$$

$$\forall x \,.\, \text{PickUpHot}(x) \supset \text{PickUp}(x)$$

$$\forall x \,.\, \text{PickUpCool}(x) \supset \text{PickUp}(x)$$

$$\begin{aligned}&\forall x \,.\, \text{PickUpHot}(x) \supset \\ &\qquad \text{Holds}(\text{Hot}(\text{object}(x)), \text{time}(x)) \wedge \\ &\qquad \text{PutOnMit}(s_1(x)) \wedge \text{BeforeMeet}(\text{time}(s1(x)),\text{time}(s2(x))) \wedge \ldots\end{aligned}$$

$$\begin{aligned}&\forall x \,.\, \text{PickUpCool}(x) \supset \\ &\qquad \text{Never}(\text{Hot}(\text{object}(x)), \text{time}(x))\end{aligned}$$

Every PickUp includes grasping the object, lifting it, and so on. PickUps are either of hot or cool objects. Picking up a hot object requires a preliminary step of putting on a mit. Picking up a cool object is constrained to occur only when the object is never hot during the time of the PickUp.

Chapter 3
Covering Models

We have seen that there are too many models of an event hierarchy to construct a semantic basis for recognition. A technique known as *model minimization* can be used to select a suitable subset, called *covering models.* In a covering model, any non-End event is a component of some End event. Each covering model for an observation serves as an *explanation,* in terms of End events, of the observation. While it would be unwise to arbitrarily adopt a *particular* covering model, it is reasonable to conclude whatever propositions hold in *all* covering models. These propositions are *c-entailed* by the observation. The sequence of model minimizations used to construct the covering models corresponds to a complex application of McCarthy's *predicate circumscription* schema. An easily-computed set of *completeness assumptions* provides a complete proof-theoretic description of the covering models.

Recall that a lexical hierarchy H contains two major parts: an abstraction hierarchy, H_A, and a decomposition hierarchy, H_D. Each of these hierarchies must be strengthened in order to be used for recognition. The abstraction hierarchy is strengthened by *assuming* that there are no event types outside of H_E, and that all abstraction relations between event predicates are derivable from H_A. The decomposition hierarchy is strengthened by *assuming* that non-End events occur only as components of other events.

These assumptions are reasonable because H encodes *all* of our knowledge of events. If the hierarchy is enlarged, the assumptions must be revised. (This thesis does not deal with learning, but may be compatible with various learning strategies. An obvious approach is to try to explain the observations while assuming H is complete; if there are no covering models of small cardinality in End, then relax some of the completeness assumptions.) At the conceptual level one can imagine computing all the completeness assumptions each time the hierarchy is modified. An implementation, of course, could incrementally build structures representing the assumptions as each part of the hierarchy is created.

3.1 Model Minimization

Let M_1 be a member of a class of models μ, and let π be a set of predicates. M_1 is **minimal in π among μ** if and only if there does not exist any other model M_2 such that:

1. M_2 is a member of μ.

2. M_1 and M_2 have the same domain.

3. M_1 and M_2 agree on the interpretation of all constants, functions, and predicates not in π.

3. The extension of every member of π in M_2 is a subset of the extension of that predicate in M_1.

5. The extension of some member of π in M_2 is a proper subset of the extension of that predicate in M_1.

If M_1 fails to be minimal because of such an M_2, we say that M_2 **defeats the candidacy** of M_1.

3.2 Completing the Abstraction Hierarchy

The following conditions incrementally define the **A-closed models of H**. In each case, M is a model of H_A.

- M is **closed under specialization** if M is minimal in H_E–H_{EB} among models of H_A. That is, all non-basic event types are minimized.

- M is **closed under abstraction** if M is minimal in H_E–{AnyEvent} among models of H_A which are closed under specialization.

- Finally, we define M to be an **A-closed** model of H just in case M is a model of H, and M is also a model of H_A which is closed under abstraction.

The following theorems describe the A-closed models in proof-theoretic terms. Proofs appear in the Appendix. Theorem 3.1 says that the given specializations of each type are assumed to be exhaustive. Theorem 3.5 shows that we have assumed that all event types are disjoint, unless H explicitly states otherwise. (Disjointedness assumptions are used heavily in recognition, as demonstrated below. We do *not* assert that different event types cannot occur simultaneously; only that a particular *token* cannot be of two non-compatible types.) Theorem 3.7 says that every event is of exactly one basic type. Theorem 3.9 presents a complete axiomatization of the A-closed models.

3.2.1. Theorem 3.1 (Exhaustiveness)

Suppose $\{E_1, E_2, \ldots, E_n\}$ are all the predicates directly abstracted by E_0 in H_A. Then the statement:

$$\forall x \,.\, E_0(x) \supset (E_1(x) \vee E_2(x) \vee \ldots \vee E_n(x))$$

is true in all models of H_A which are closed under specialization. The statement is also true in all A-closed models of H.

3.2.2. Theorem 3.5 (Disjointedness)

If event predicates E_1 and E_2 are not compatible, then the statement:

$$\forall x \,.\, \neg E_1(x) \vee \neg E_2(x)$$

is true in all models of H_A which are closed under abstraction. The statement is also true in all A-closed models of H.

3.2.3. Theorem 3.7 (Unique Basic Types)

If M_1 is a model of H_A closed under abstraction containing event token $:C_1$, then there is a unique basic event type E_b such that $:C \in M_1[E_b]$. Any event type which holds of $:C$ abstracts* E_b.

3.2.4. Theorem 3.9 (Abstraction Completeness)

Let EXA be the set of all statements which instantiate Theorem 3.1, and let DJA be the set of all statements which instantiate Theorem 3.5 for a particular H. M_1 is an A-closed model of H if and only if M_1 is a model of $H \cup EXA \cup DJA$.

3.3 Completing the Decomposition Hierarchy

Once the abstraction hierarchy has been closed, it is a simple matter to close the decomposition hierarchy, by minimizing the set of non-End event types. C-entailment is then defined in terms of the covering models. Theorem 3.10 says that every non-End event must be a component of some other event, and Theorem 3.11 allows one to infer the disjunction of possible uses of observed event token. By Theorem 3.13 every event is part of an End event, and by 3.14 the upward-inference assumptions exactly axiomatize the covering models. Theorem 3.15 states the obvious corollary that c-entailment is computable. Theorem 3.16 shows that one must consider all the possible abstractions and specializations of an event in order to account for all of its possible uses -- and therefore predicate completion in the style of [Clark 78] does not correspond to decomposition completion.

3.3.1. Definition of Covering Model and C-Entailment

M is a **covering model** of H if M is minimal in H_E–{End} among A-closed models of H. Then Γ **c-entails** Ω, written

$$\Gamma_H \models_c \Omega$$

when Ω holds in all covering models of H in which Γ holds. If Ω holds for any Γ, Ω is **c-valid**.

3.3.2. Theorem 3.10 (No Useless Events)

Let M_1 be a covering model of H, containing event token $:C_1$. Then either $:C_1 \in M_1[\text{End}]$ is true, or there exists some event token $:C_2$ such that $:C_1$ is a direct component of $:C_2$.

3.3.3. Theorem 3.11 (Component/Use)

Let $E \in H_E$, and Com(E) be the set of event predicates with which E is compatible. Consider all the decomposition axioms in which any element of Com(E) appears on the right-hand side. The j-th such decomposition axiom has the following form, where E_{ji} is the element of Com(E):

$$\forall x \,.\, E_{j0}(x) \supset E_{j1}(f_{j1}(x)) \wedge \ldots \wedge E_{ji}(f_{ji}(x)) \wedge \ldots \wedge E_{jn}(f_{jn}(x)) \wedge \kappa$$

Suppose that the series of these axioms, where an axiom is repeated as many times as there are members of Com(E) in its right-hand side, is of length $m > 0$. Then the following statement is c–valid:

$$\begin{aligned}\forall x \,.\, E(x) \supset \quad & End(x) \vee \\ & (\exists y \,.\, E_{1,0}(y) \wedge f_{1i}(y)=x) \vee \\ & (\exists y \,.\, E_{2,0}(y) \wedge f_{2i}(y)=x) \vee \\ & \ldots \quad \vee \\ & (\exists y \,.\, E_{m,0}(y) \wedge f_{mi}(y)=x)\end{aligned}$$

3.3.4. Theorem 3.13 (No Infinite Chains)

If M_1 is a covering model of H such that $C_1 \in M_1[E]$, then there is a $:C_n$ such that $:C_n \in M_1[End]$ and $:C_1$ is a component* of $:C_n$.

3.3.5. Theorem 3.14 (Decomposition Completeness)

Let CUA be the set of all formulas which instantiate Theorem 3.11 for a particular H. M_1 is a covering model of H if and only if M_1 is a model of $H \cup EXA \cup DJA \cup CUA$.

3.3.6. Theorem 3.15 (Computability of C-Entailment)

There is a computable function **cl** which maps a hierarchy H into a set of axioms with the property that

$$\Gamma_H \models_c \Omega$$

if and only if

$$cl(H) \cup \Gamma \models \Omega$$

3.3.7. Theorem 3.16 (Not Predicate Completion)

Theorem 3.11 cannot be strengthened by considering only axioms in which E appears as a component, instead of all ones in which event types compatible with E appear as components.

3.4 Circumscription

Predicate circumscription [McCarthy 84] provides a proof theoretic realization of the model-theoretic minimalization operation used above. Direct use of the circumscription schema is difficult, however. Its most general form is a second-order, rather than a first-order statement. Techniques are known for automatically computing first-order circumscriptions for certain kinds of axiom sets; for example, for horn-clauses [Bossu & Seigel 85], or "separable" databases [Lifschitz 84]. None of these previously known techniques apply in the case under consideration here. Although the original hierarchy may be in horn-clauses, the result of the first minimization is a non-horn set of sentences, so the techniques based on predicate-completion are not applicable. None of the minimizations involve separable sets of predicates. However, the restricted form of lexical hierarchies has allowed us to directly compute a set of first-order statements which characterize all models resulting from a certain sequence of minimizations. This work thus describes a special but useful case in which circumscription can be efficiently computed.

Recall that the circumscription of the set of predicates π over a formula S, written Circum(S[π],π), stands for the second-order formula

$$S[\pi] \wedge \forall \sigma . (S[\sigma] \wedge \sigma \leq \pi) \supset \pi \leq \sigma$$

where the expression $\sigma \leq \pi$ abbreviates the formula stating that the extension of each predicate in σ is a subset of the extension of the corresponding predicate in π; that is:

$$(\forall x . \sigma_1(x) \supset \pi_1(x)) \wedge \ldots \wedge (\forall x . \sigma_n(x) \supset \pi_n(x))$$

3.4.1. Theorem 3.17 (C-entailment and Circumscription)

For a given a hierarchy H, a statement Ω is c-entailed by Γ if and only if Ω follows from the following schema:

$$\begin{aligned}\Gamma \wedge \text{Circum}(\ H \wedge \text{Circum}(\ &\text{Circum}(\ H_A\ ,\ H_E - H_{EB}\ ),\\ &\{\text{AnyEvent}\}),\\ H_E - \{\text{End}\})&\end{aligned}$$

Note that the two inner circumscriptions apply only to the abstraction hierarchy, while the final circumscription applies to all of H.

3.5 Example: The Cooking World, Continued

We return to the domain of the cooking, and discuss some of the statements which appear in the closure of the lexical hierarchy. These statements are then used to solve a simple plan recognition problem. The closure function, cl, generates three sets of axioms: the exhaustiveness assumptions (EXA), disjointedness assumptions (DJA), and component/use assumptions (CUA). We consider each set in turn.

3.5.1. Exhaustiveness Assumptions (EXA)

These axioms arise by minimizing non-basic event types. They justify inferences from an abstract type to the disjunctive of its specializations. In the cooking world, EXA includes the assertion that all End events are either instances of preparing meals or washing dishes:

$$\begin{aligned}&\forall x\ .\ \text{End}(x) \supset\\ &\qquad\qquad \text{PrepareMeal}(x) \vee\\ &\qquad\qquad \text{CleanHouse}(x)\end{aligned}$$

Similarly, all instances of preparing meals are either instances of making a pasta dish or making a meat dish.

$$\begin{aligned}&\forall x\ .\ \text{PrepareMeal}(x) \supset\\ &\qquad\qquad \text{MakePastaDish}(x) \vee\\ &\qquad\qquad \text{MakeMeatDish}(x)\end{aligned}$$

One such axiom appears for every event type not in H_{EB}.

3.5.2. Disjointedness Assumptions (DJA)

These axioms arise by minimizing all event types other than AnyEvent. This minimizes the set of types of each event token. By EXA, every event is of some basic type; by DJA, it is of no more than one basic type. The final effect of this is that types are disjoint, unless one abstracts the other, or they abstract a common type. It is important to reiterate that the assumptions do *not* assert that different event types cannot occur simultaneously; only that a particular *token* cannot be of two non-compatible types. The cooking world example includes the assumptions that preparing a meal and cleaning a house are disjoint; making a pasta dish and making a meat dish are disjoint; and so on.

$$\forall x \,.\, \neg \mathrm{PrepareMeal}(x) \vee \neg \mathrm{CleanHouse}(x)$$

$$\forall x \,.\, \neg \mathrm{MakePastaDish}(x) \vee \neg \mathrm{MakeMeatDish}(x)$$

It is simply to "block" a potential disjointedness assumption by adding a new type. For example, suppose we do not want to make the assumption that pasta dishes and meat dishes are disjoint. Then add a type to the original hierarchy which is abstracted by both; for example, MakeMeatRavioli.

adding to H_A

$$\forall x \,.\, \mathrm{MakeMeatRavioli}(x) \supset \mathrm{MakePastaDish}(x)$$

$$\forall x \,.\, \mathrm{MakeMeatRavioli}(x) \supset \mathrm{MakeMeatDish}(x)$$

would eliminate from DJA

$$\forall x \,.\, \neg \mathrm{MakePastaDish}(x) \vee \neg \mathrm{MakeMeatDish}(x)$$

3.5.3. Component/Use Assumptions (CUA)

The most important assumptions for recognition arise from minimizing non-End events. The assumptions in CUA let one infer the disjunction of the possible causes for an event from its occurrence. The axioms take us from an event to an event which has a compatible type as a component. The simplest case is when only a single type could have a particular event as a direct component. For instance, the hierarchy has only a single use for the action Boil, namely MakePastaDish.

$$\forall x \,.\, \mathrm{Boil}(x) \supset \\ \exists y \,.\, \mathrm{MakePastaDish}(y) \wedge x = \mathrm{step3}(y)$$

It is frequently possible to simplify the statements in CUA, by taking advantage of the abstraction axioms. For example, according the rule for construction of CUA, the upward-inference axiom for MakeNoodle is:

$$\begin{aligned} &\forall x\ .\ \mathrm{MakeNoodle}(x) \supset \\ &\qquad (\exists y\ .\ \mathrm{MakePastaDish}(y) \land x = \mathrm{step1}(y)) \lor \\ &\qquad (\exists y\ .\ \mathrm{MakeSpaghettiMarina}(y) \land x = \mathrm{step1}(y)) \lor \\ &\qquad (\exists y\ .\ \mathrm{MakeSpaghettiPesto}(y) \land x = \mathrm{step1}(y)) \lor \\ &\qquad (\exists y\ .\ \mathrm{MakeFettuciniAlfredo}(y) \land x = \mathrm{step1}(y)) \end{aligned}$$

The following axiom is equivalent, in light of H_A.

$$\begin{aligned} &\forall x\ .\ \mathrm{MakeNoodle}(x) \supset \\ &\qquad \exists y\ .\ \mathrm{MakePastaDish}(y) \land \mathrm{M1} = \mathrm{step1}(y) \end{aligned}$$

The simplification could not be made if MakePastaDish did not abstract all of the other event types, MakeSpaghettiMarina, MakeSpaghettiPesto, and MakeFettuciniAlfredo.

Another axiom that will be useful in our examples lets one infer the two known uses for making marinara sauce, namely making spaghetti marinara and making chicken marinara.

$$\begin{aligned} &\forall x\ .\ \mathrm{MakeMarinara}(x) \supset \\ &\qquad (\exists y\ .\ \mathrm{MakeSpaghettiMarinara}(y) \land x = \mathrm{step1}(y)) \lor \\ &\qquad (\exists y\ .\ \mathrm{MakeChickenMarinara}(y) \land x = \mathrm{step3}(y)) \end{aligned}$$

This statement has also been simplified. The original form should include the disjunct MakePastaDish, since MakePastaDish has the direct component MakeSauce, which is compatible with MakeMarinara. However, the exhaustiveness assumptions can be used to conclude that any MakePastaDish is either MakeSpaghettiMarinara, MakeSpaghettiPesto, or MakeFettuciniAlfredo. The step1 role in the latter two cases must be filled by an event of type MakePesto or MakeAlfredo respectively. The disjointedness assumptions can then be used to infer that MakeMarinaraSauce, MakePesto, and MakeAlfredo are mutually disjoint. Therefore the possibility that MakeMarinaraSauce could be a component of MakeSpaghettiPesto or MakeFettuciniAlfredo can be eliminated. We will use without further comment simplified versions of the assumptions in CUA.

Finally, the UPA axiom for MakeSauce demonstrates the importance of considering compatible types. The only *direct* use of MakeSauce is MakePastaDish; however, MakeSauce abstracts MakeMarinara, and *that* event type appears in the decomposition of MakeChickenMarinara. Therefore the axiom is:

$$\begin{aligned}&\forall x\ .\ \mathrm{MakeSauce}(x) \supset \\ &\qquad (\exists y\ .\ \mathrm{MakePastaDish}(y) \wedge x = \mathrm{step1}(y)) \vee \\ &\qquad (\exists y\ .\ \mathrm{MakeChickenMarinara}(y) \wedge x = \mathrm{step5}(y))\end{aligned}$$

3.5.4. A Simple Recognition Problem

The assumptions justified by c-entailment can be used to solve simple recognition problems. Let us suppose that the observer learns that the cook is either making spaghetti or fettucini. This is not enough information to conclude a particular basic dish is being created. It is safe to conclude, however, that *some* pasta dish is in the works. From this abstract description of the End event in progress, the observer can still make useful predictions. For example, he knows that the agent will eventually begin to boil water. A helpful observer might fetch a pot of water; a enemy observer might shut off the water supply!

Following is a proof sketch. The justification for each step appears in italics.

Observation

$\mathrm{MakeFettucini}(M1) \vee \mathrm{MakeSpaghetti}(M1)$

Abstraction

$\mathrm{MakeNoodle}(M1)$

Component/use, Existential Instantiation

$\mathrm{MakePastaDish}(*I1)$

Abstraction

$\mathrm{End}(*I1)$

Decomposition

$\mathrm{Boil}(\ \mathrm{step3}(*I1)\ ) \wedge \mathrm{After}(\mathrm{time}(M1), \mathrm{time}(\mathrm{step3}(*I1))\ )$

The final step is the predication, that a boiling event will occur at some time after the observation. As discussed in Chapter 2, the final logical form should replace the individual parameter *I1 by an existentially-quantified variable, and write

$$\exists y \,.\, \mathrm{Boil}(\ \mathrm{step3}(y)\ ) \wedge \mathrm{After}(\mathrm{time}(M1), \mathrm{time}(\mathrm{step3}(y))\ )$$

Chapter 4
Minimum Covering Models

C-entailment does not combine information from several observations. We would like to intersect possible explanations for each event. This is done by selecting covering models which minimize the *number* of end events. Minimization can be understood either as producing a set with as few elements as possible, or as producing a set with no redundant elements. The former sense is called cardinality minimization, and the latter, set minimization. (The use of the word minimization without qualification will always mean set minimization.) The model minimization schemes used in Chapter 3 are all instances of set minimization. The work in this chapter relies on cardinality minimization. We will show that certain cases of *circumscription with variables* correspond to numeric minimization.

4.1. Cardinality Minimization

Let M_1 be a member of a class of models μ, and let π be a predicate. M_1 **has minimum cardinality in π among μ** if and only if there does not exist any other model M_2 such that:

1. M_2 is a member of μ.

2. The size of the extension of π in M_2 is smaller than the size of the extension of π in M_1. That is,
$$|M_2[\pi]| < |M_1[\pi]|$$

If M_1 fails to be minimal because of such an M_2, we say that M_2 **defeats the candidacy** of M_1.

Let Γ be any sentence or set of sentences, and H a hierarchy. M is a **minimum cover** of Γ (relative to H) just in case

1. M is a model of Γ

2. M is a covering model of H

3. M has minimum cardinality in End among covering models of H

Suppose Γ is any sentence. Then P is **mc-entailed** by Γ, written

$$\Gamma_H \models_{mc} \Omega$$

if Ω holds in all minimum covers of Γ.

The proof-theoretic counterpart of cardinality minimization is to adopt the strongest statement which limits the number of distinct end events. Unfortunately, this step is not always effectively computable (the usual problem with default reasoning). In practice, one makes the strongest assumption possible; forward chains a limited amount; if a contradiction is detected, then makes the next weaker assumption, and repeats.

4.1.1. Theorem 4.1 (Minimum Cardinality Defaults)

Consider the following sequences of statements.

MA_0. $\forall x . \neg End(x)$

MA_1. $\forall x,y . End(x) \wedge End(y) \supset x=y$

MA_2. $\forall x,y,z . End(x) \wedge End(y) \wedge End(z) \supset (x=y) \vee (x=z) \vee (y=z)$

...

The first asserts that no End events exist; the second, no more than one End event exists; the third, no more than two; and so on. Suppose there is a minimum covering model in which the extension of End is finite. Then

$$\Gamma_H \models_{mc} \Omega$$

if and only if

$$\Gamma \cup cl(H) \cup MA_i \vdash \Omega$$

where i is the smallest integer such that left-hand side of the provability relation is consistent.

4.2. Cardinality Circumscription

At first glance it would seem that the theory of circumscription cannot handle the problem of minimizing the cardinality of the extension of a predicate. The circumscription schema states that there is no predicate which satisfies the axioms for the minimized predicate which has an extension which is a proper subset of that of the predicate. The extension must be minimal, not a minimum.

But consider the role of the non-minimized symbols in the circumscription. In the simplest version of circumscription, used in the previous chapter, these symbols must have the same denotation in comparable models. But the more general theory of circumscription [McCarthy 85] allows one to specify that certain predicates, functions, and/or constants *vary* during the minimization. Models may be comparable even if they do not agree on those symbols. The definition of minimum cardinality above compares models without regard to their agreement on any symbols. One could try to represent this process by circumscribing a predicate where all other symbols are allowed to vary. Surprisingly, this works! Under easily met conditions, minimizing cardinality is equivalent to setwise minimization where all predicates and functions vary.

Why should this be the case? Suppose we have two models, M_1 and M_2, where the cardinality of the extension of End is larger in M_1 than it is in M_2, yet the extension of End in M_1 does not contain the extension of End in M_2. M_1 and M_2 are not comparable. But there must be a model that "looks like" M_2, which *is* comparable to M_1. This model simply swaps the roles played by certain domain elements in all predicates and functions, so that its extension of End is a proper subset of M_1's. We can precisely define how this can be done.

4.2.1. Circumscription with Variables

The circumscription schema can be modified to allow symbols to vary. The **circumscription of π over a formula S where α varies** is written as

$$\text{Circum}(S[\pi,\alpha],\pi,\alpha)$$

which abbreviates the second-order formula

$$S[\pi,\alpha] \wedge \forall\, \sigma,\beta\ .\ (S[\sigma,\beta] \wedge \sigma \leq \pi) \supset \pi \leq \sigma$$

In the schema, α stands for either a single symbol or a sequence of symbols.

4.2.2. Theorem 4.2 (Cardinality Circumscription)

Let α include all the predicate, function, and constant symbols in our language other than End. Suppose that all models of H are infinite, and in some model of $\Gamma \cup \mathrm{cl(H)}$, End has a finite extension. If

$$\mathrm{Circum}(\Gamma \cup \mathrm{cl(H)}, \{\mathrm{End}\}, \alpha) \vdash \Omega$$

then

$$\Gamma_H \models_{mc} \Omega$$

where $\mathrm{Circum}(\Gamma \cup \mathrm{cl(H)}, \{\mathrm{End}\}, \alpha)$ means to circumscribe with α varying. The "if" is strengthened to "if and only if" if it is true that circumscription is complete in this case.

4.2.3. Example of Cardinality Circumscription

Consider the following simple example of circumscribing the cardinality of a predicate. The formula to be circumscribed contains only the predicate End, and three constant symbols. It asserts that either a is End, or both of b and c are.

$$S[\mathrm{End},a,b,c] = a \neq b \wedge a \neq c \wedge b \neq c \wedge (\ \mathrm{End}(a) \vee (\ \mathrm{End}(b) \wedge \mathrm{End}(c)\ )\ )$$

The circumscription of End in S where the constants do not vary simply strengths the disjunction to exclusive or.

$$\begin{aligned}&\mathrm{Circum}(\ S[\mathrm{End},a,b,c], \mathrm{End}\ ) \equiv \\ &\qquad a \neq b \wedge a \neq c \wedge b \neq c \wedge (\ \mathrm{End}(a) \oplus (\ \mathrm{End}(b) \wedge \mathrm{End}(c)\ )\ )\end{aligned}$$

This is because there is a minimal model where both b and c are End. Now try circumscribing with a, b, and c varying.

$$\begin{aligned}&\mathrm{Circum}(\ S[\mathrm{End},a,b,c], \mathrm{End}, \{a, b, c\}\ ) \equiv \\ &\qquad S[\mathrm{End},a,b,c] \wedge \\ &\qquad \forall p,x,y,z\ .\ (\ S[p,x,y,z] \wedge \forall w\ .\ p(w) \supset \mathrm{End}(w)\ ) \supset \\ &\qquad\qquad\qquad\qquad \forall w\ .\ \mathrm{End}(w) \supset p(w)\end{aligned}$$

Instantiations for p, x, y, and z are chosen.

Let $p = (\lambda\, u\, .\, u = b)$
$x = b$
$y = a$
$z = c$

The instantiated schema becomes:

$$S[End,a,b,c] \wedge$$
$$(S[(\lambda\, u\, .\, u = b),b,a,c] \wedge End(b)) \supset \forall w\, .\, End(w) \supset w=b$$

Expanding $S[(\lambda\, u\, .\, u = b),b,a,c]$:

$$S[End,a,b,c] \wedge$$
$$(b \neq a \wedge b \neq c \wedge a \neq c \wedge (b=b \vee (a=b \wedge c=b)) \wedge End(b)) \supset$$
$$\forall w\, .\, End(w) \supset w=b$$

Simplify, by eliminating the atoms in the second main conjunct which are implied by the first:

$$S[End,a,b,c] \wedge$$
$$End(b) \supset \forall w\, .\, End(w) \supset w=b$$

Now reason by cases. Suppose End(b). Then

$$\forall w\, .\, End(w) \supset w=b$$

Since $c \neq b$, this means $\neg$End(c). Together with S[End,a,b,c], this yields End(a). On the other hand, suppose $\neg$End(b). Then again, it must be the case that End(a).

So far we've shown that End(a) is a consequence of the circumscription. Reinstantiating the schema strengthens the conclusion to that a is the only member of End.

Let $p = (\lambda\, u\, .\, u = a)$
$x = a$
$y = b$
$z = c$

It is straightforward to show that the instantiated schema:

$$S[End,a,b,c] \wedge$$
$$(S[(\lambda\, u\, .\, u = a),a,b,c] \wedge End(a)) \supset \forall w\, .\, End(w) \supset w=a$$

implies

$$End(a) \supset \forall w \,.\, End(w) \supset w=a$$

Thus, circumscribing End with a, b, and c varying forces End to have cardinality 1:

$$Circum(\ S[End,a,b,c],\ End\ ) \vdash End(a) \wedge \forall w \,.\, End(w) \supset w=a$$

4.3. Example: The Cooking World, Continued

We continue with the example of plan recognition in the cooking world. There are two observations. The first is described in the previous chapter: the cook is reported to be making spaghetti or fettucini. We concluded that the cook was making some kind of pasta dish. The second observation is that the cook is making marinara sauce. From the second observation alone one could conclude that one of the dishes involving marinara sauce, namely spaghetti marinara or chicken marinara, was being prepared. If we allow the possibility that the two observations are unrelated, then all that one can conclude is the conjunction of the two conclusions; namely,

$$\begin{array}{l} \exists x \,.\, MakePastaDish(x) \wedge \\ \exists y \,.\, (MakeSpaghettiMarinara(y) \vee MakeChickenMarinara(y)) \end{array}$$

But consider what holds in all the minimum covering models. In such a model, there is only one End events (in the present example). Each of the variables x and y must be interpreted as End event, since the types MakePastaDish, MakeSpaghettiMarinara, and MakeChickenMarinara all specialize End. Therefore x and y must be interpreted as the same entity. The disjointedness assumptions tell us that an entity cannot be both a MakePastaDish and a MakeChickenMarinara. Therefore the cook must be making spaghetti marinara in all minimum covering models. This is the conclusion mc-entailed by the observations.

Following is a proof sketch. The first part of the proof appears in the previous chapter.

Second Observation

$$MakeMarinara(M2)$$

Upward Inference, Existential Instantiation

$$MakeSpaghettiMarinara(*I2) \vee MakeChickenMarinara(*I2)$$

Abstraction

$\mathrm{MakePastaDish}(^*I2) \lor \mathrm{MakeMeatDish}(^*I2)$

Abstraction

$\mathrm{EndEvent}(^*I2)$

Conclusion from first Observation (Chapter 3)

$\mathrm{EndEvent}(^*I1)$

Strongest Minimality Assumption

$\forall\, x,y\ .\ \mathrm{EndEvent}(x) \land \mathrm{EndEvent}(y) \supset x=y$

Universal Instantiation & Modus Ponens

$^*I1 = {}^*I2$

Conclusion from first Observation (Chapter 3)

$\mathrm{MakePastaDish}(^*I1)$

Substitution of Equals

$\mathrm{MakePastaDish}(^*I2)$

Disjointedness Assumption

$\forall\, x\ .\ \neg\mathrm{MakePastaDish}(x) \lor \neg\mathrm{MakeMeatDish}(x)$

Disjunction Elimination

$\neg\mathrm{MakeMeatDish}(^*I2)$

Abstraction

$\mathrm{MakeChickenMarinara}(^*I2) \supset \mathrm{MakeMeatDish}(^*I2)$

Modus Tolens

$\neg\mathrm{MakeChickenMarinara}(^*I2)$

Disjunction Elimination

$\mathrm{MakeSpaghettiMarinara}(^*I2)$

Chapter 5
Incremental Recognition

A recognition problem is *incremental* when the recognizer is presented with a temporal *sequence* of observations, $\Gamma_1, \Gamma_2, \ldots$. At any point in the sequence, the recognizer can be queried as to the consequences of the observations made so far.

Suppose mc-entailment is used to model the recognition process. Then after observation Γ_n, the recognizer can conclude Ω just in case

$$\Gamma_1 \cup \ldots \cup \Gamma_n \ {}_H\!\models_{mc} \Omega$$

The mc-entailment relation is non-monotonic. As Γ_i are added to the left-hand side, the set of conclusions may no longer include Ω. Consider the following specific case.

Each observation is of the occurrence of a single event. Each observed event must be a component of an End event; call these End events $N_1, N_2, \ldots N_n$. Let there be a minimum covering model for $\Gamma_1, \ldots \Gamma_n$ in which the extension of End has cardinality 1; that is, $N_1=N_2=\ldots=N_n$. If the number of observations is of reasonable size, it will quite often be the case that there is only a single specialization of End which could account for all of the observations. In particular, suppose that

$$\Gamma_1 \cup \ldots \cup \Gamma_n \ {}_H\!\models_{mc} \exists\, x \,.\, E_k(x)$$

Now suppose the n+1st observed event cannot be part of the same End event as all the previous ones. A minimum cover for $\Gamma_1, \ldots \Gamma_{n+1}$ must be of size 2. The recognizer can no longer assume that $N_1=N_2=\ldots=N_n$. Instead, it may be the case that $N_1=N_{n+1}$, and all the other N_j are equal; or that $N_2=N_{n+1}$; and so on. There may be as many as 2^n different ways of grouping the observations, and each could correspond to a different minimum covering model. The conclusion $\exists x.E_k(x)$ must be withdrawn, and replaced with a long disjunction of different specializations of End.

5.1. Deficiencies in the MC Model

This kind of non-monotonic behavior is justified as a normative theory of what an agent should come to believe, if the minimization of End events is the only basis for combining information from different observations. Objections can be raised on the grounds of complexity and psychological plausibility. The theory of mc-entailment can be adjusted, however, to both have a more efficient computational realization, and to more accurately reflex some of our intuitions about incremental recognition processes.

5.1.1. The Combinatorial Problem

Once the minimum cardinality default is weakened to allow two or more distinct End events, the number of ways of *grouping* the observations together as components of the same End event grows exponentially. In some domains the various constraints associated with event types could quickly rule out most of these possibilities. But it seems clear that additional principles will eventually be needed to deal with realistically-sized problems.

5.1.2. The Persistence Problem

Related to the combinational problem is the failure of mc-entailment to match up with our intuitions about the way a person would go about solving an incremental recognition problem. Once several pieces of information are "tied together", it seems unnatural to break that connection, simply because some seemingly *unrelated* piece of information comes along.

Consider the following (very!) short story, bearing in mind the event hierarchy in the beginning of Chapter 1.

> *Leo needed some cash. He took out his shotgun. Then he went to the bank. Later that day, he was seen in the woods.*

As we read along, the descriptions of the Leo's getting a gun and going to the bank, together with his state of needing money, invoke the bank robbery event (or plan or script or ...). Walking in the woods is not part of robbing a bank. None the less, the story still seems to be about a bank robbery; we don't consider the possibility that Leo got his gun in order to go hunting in the woods. After the first three sentences, we conclude something like

$$\exists\, x \,.\, \mathrm{RobBank}(x) \land \mathrm{agent}(x)=\mathrm{Leo}$$

and this belief *persists* even after the last sentence. Mc-entailment would justify this conclusion after the first three sentences, but after the last would only justify the weaker conclusion:

$$\exists\, x,y\,.\ (\mathrm{RobBank}(x) \wedge (\mathrm{Hunt}(y) \vee \mathrm{Go\text{-}Hiking}(y))) \vee (\mathrm{CashCheck}(y) \wedge \mathrm{Hunt}(x))$$

What is going on? One explanation is that we are minimizing End events in a incremental fashion. If we have to increase the number of End events to account for the story, we do, but we try not to *discard* any previous conclusions. Of course, new information could *force* us to withdraw a previous conclusion. So if the story continued

> *Leo had spent a long time in line at the bank, waiting to cash his paycheck. He was glad to finally be out stalking the wild moose.*

the earlier conclusion is eliminated. But such stories appear either awkward or deliberately misleading (as a mystery story *should* be); making an analogy with syntax, one might call them *garden-path* stories.[1]

This manner of reasoning is understandable if one views the times *between* observations as points at which the recognizer *accepts* his non-deductive conclusions as *full beliefs*. These beliefs, together with whatever else the recognizer believes, are subject to *minimal revision* as contradictory information is learned. But new *non-deductive* conclusions are not (in this model) the basis for revising old beliefs.

The difficulty with this model is that the minimum cardinality defaults are *always* revised when the number of End events must be increased. How then can conclusions *based* on these assumptions be retained? Section 5.3 below develops a model theory which exhibits this behavior, by separating *existential* and *universal* conclusions: the former may *persist* while the latter are rejected.

5.2. Refining the Model

This section considers a number of factors that could go into a theory of incremental recognition.

5.2.1. Incrementally Minimize Cardinality

[*] A garden-path sentence is a syntactically "correct" sentence which is difficult to understand, because the hearer makes incorrect decisions about its constituent structure before the end of the sentence is reached. The standard example is the sentence, "The horse raced past the barn fell."

Both the persistence problem and some aspects of the complexity problem are addressed by incrementally minimizing the number of End events, as described above. Most of the plan recognition systems described in the literature perform an incremental minimization, usually together with some of the following heuristics and constraints. Chapter 7 describes an algorithm for a plan recognizer which implements this heuristic.

5.2.2. Sticky Covers

A heuristic for grouping observations described in [Huff & Lesser 82] is to associate each new observation with the *most recent* observation with which it can consistently grouped. We call this the *sticky covering* assumption, since each observation tries to "stick" to the previous one. The sticky heuristic solves the combinatorial problem, as discussed in detail in Section 7.9.3, but is even more sensitive to garden-path type errors. An algorithm for calculating conclusions true in all sticky covers is given in Chapter 7, but we have not tried to work out a corresponding model theory.

5.2.3. Discourse Clues

Much work in computational models of discourse attempts to find clues which indicate how the various utterances which make up an extended discourse should be grouped. These take into consideration such features as "clue words" (such as "but" and "therefore") [Grosz 77] and intonation [Pierrehumbert 77].

5.2.4. Likelihood Associations

Eventually we'll need to deal with the frequency with which an event appears as a component of its different uses. The less common uses may be ignored if possible. Such a quantitative approach is not antithetical to the work described here. Once our qualitative methods choose all the possible interpretations of the data under some set of simplicity heuristics, probabilistic methods can be applied to rank and choose between the alternatives. This is exactly the approach taken by expert systems which employ notions similar to covering models, as described in Sections 1.4 and 6.4.

5.3. Model Theory for Incremental Minimization

The section defines a weaker version of minimum covering model, and then uses it to define an incremental operator, imc-entailment.

5.3.1. Minimum Covering Submodels

Mc-entailment justifies some kinds of conclusions that one would not ordinarily draw from a set of observations. Suppose that one observes A and B, and C is the only End act which entails both A and B. Then would seem reasonable to conclude C, and mc-entailment justifies this conclusion. However, mc-entailment also justifies the conclusion that no End act other than C occurs; and that assumption was the basis for the previous conclusion. Yet the statement that only one event will ever occur, at all times now and in the future, is almost surely false. The problem is that our intuitively acceptable conclusion C, has the same status as the dubious conclusion that nothing else will ever happen.

Simply eliminating the cardinality minimization would eliminate the doubtful conclusion, but at the cost of any combination of information from multiple observations. A more attractive approach is to weaken the notion of mc-entailment. Instead of minimizing the number of End events per se, we really only need to minimize the number of events which account for all the observations. There may be other End events which occur at other times and places, but they do not involve the current set of observed events. Instead of drawing conclusions based on the class of minimum covering models, one should draw conclusions based on the larger class of models which contain a minimum covering submodel.

Theorem 5.1 states that all the *existential* conclusions that one can obtain by considering minimum covering models can also be obtained by considering the larger class. This will provide the basis for retaining conclusions such as

$$\exists\, x \,.\, \mathrm{RobBank}(x) \wedge \mathrm{agent}(x)=\mathrm{Leo}$$

while dropping the assumption that

$$\forall\, x,y \,.\, \mathrm{End}(x) \wedge \mathrm{End}(y) \supset x=y$$

5.3.1.1. Definitions

M_1 is a **submodel** of M_2 if the domain of M_1 is a subset of M_2; M_1 and M_2 agree on all predicates for tuples consisting only of elements in M_1; and M_1 and M_2 agree on the interpretation of all functions, restricted to the domain of M_1. If M_1 is both a covering model of Γ relative to H, and has a submodel M_2 which is a minimum cover for Γ relative to H, we say M_1 **contains a minimum cover for Γ relative to H.**

Ω is **mcs-entailed** by Γ, written

$$\Gamma \;_{H}\models_{mcs} \Omega$$

if Ω holds in all models which contain a minimum cover of Γ relative to H. For brevity, we will call a model which contains a minimum cover an **mcs-model**.

5.3.1.2. Theorem 5.1 (Non-Universal Conclusions)

Let Ω be a sentence which, when written with all quantifiers in initial position, contains no universal quantifiers. Then

$$\Gamma \;_{H}\models_{mcs} \Omega$$

if and only if

$$\Gamma \;_{H}\models_{mc} \Omega$$

5.3.2. Monotonic Incremental Recognition

The intuition behind a monotonic theory of incremental recognition is that after the recognizer obtains a piece of evidence, he believes forever the consequences of that evidence. This section defines the final operator which relates a sequence of observations to its *incrementally minimum covering entailed* conclusions. Theorem 5.2 states that this operator has the desired properties of acting like mc-entailment when there is only one observation, and allowing existential conclusions to persist as new observations are made.

5.3.2.1. Definitions

Let $\Gamma = (\Gamma_1, \Gamma_2, \ldots \Gamma_n)$ be an **observation sequence**. Following is a recursive definition of the class of **incremental minimal covers** of Γ. Let us say that M is a **n-candidate** if M is a covering model of Γ relative to H, and (if n>1) M is an incremental minimum cover of $(\Gamma_1, \Gamma_2, \ldots \Gamma_n)$. Then M is an incremental minimal cover of Γ relative to H if M is an n-candidate, and:

(i) n=1, and M contains a minimum cover for Γ relative to H.

(ii) or n > 1, and M contains a submodel M_2 which has minimum cardinality in End among covering models of Γ which are submodels of n-candidates.

Ω is **imc-entailed** by $(\Gamma_1, \Gamma_2, \dots \Gamma_n)$, written

$$(\Gamma_1, \Gamma_2, \dots \Gamma_n)\ {}_H \models_{imc} \Omega$$

if Ω holds in all models which are incremental minimum covers of (Q1, Q2, ... Qn) relative to H.

5.3.2.2. Theorem 5.2 (Incremental Recognition)

In the case of a single observation, imc-entailment is the same as mcs-entailment.

$$(\Gamma 1)\ {}_H \models_{imc} \Omega$$

if and only if

$$\Gamma 1\ {}_H \models_{mcs} \Omega$$

In the case of multiple observations, imc-entailment is monotonic.

$$(\Gamma_1, \dots \Gamma_{n-1})\ {}_H \models_{imc} \Omega$$

implies

$$(\Gamma_1, \dots \Gamma_{n-1}, \Gamma n)\ {}_H \models_{imc} \Omega$$

Chapter 6
Examples

6.1. The Cooking World, Once More

The examples at the end of Chapters 3 and 4 illustrated inference from a disjunctive observation, and the combination of observations to uniquely identify the plan in progress. The next example illustrates two other important features of our system: the ability to handle disjunctive hypotheses, and to draw conclusions that reflect an abstract description of a class of possible plans.

Suppose that the observer learns that the cook is preparing some kind of sauce. This is not enough information to conclude a particular basic dish is being created: one cannot (as appears to be the case in most of the plan recognition systems described in Chapter 1) imagine that a *single* plan is evoked. Instead, one is justified in concluding that the cook is making some Pasta Dish, *or* Chicken Marinara. This disjunction collapses, via the abstraction axioms, to the fact that an event of type PrepareMeal is occurring. This final piece of information, non-specific as it is, may still be of great interest: for instance, if we are hungry! The example also illustrates the importance of adopting a formal framework that guarantees completeness: all possible uses of an event are considered. A noted in Chapter 3, the only *direct* use of MakeSauce is MakePastaDish. It is not clear whether any of the earlier heuristic-based plan recognition systems would consider MakeChickenMarinara, or if they would immediately (and unjustifiably) jump to the first conclusion. The proof so far is:

Observation

MakeSauce(Obs1)

Component/Use, Existential Instantiation

MakePastaDish(*I1) ∨ MakeChickenMarinara(*I1)

Abstraction

MakePastaDish(*I1) ∨ MakeMeatDish(*I1)

PrepareMeal(*I1)

End(*I1)

Now let the next observation be that the agent is making Noodles, and assume the End event inferred from the first observation is the same as the End event inferred from the second. Mc-entailment lets us conclude that a plan to make some Pasta Dish is in progress. We do not need (cannot, in fact) conclude that a *particular* kind of Pasta Dish is under construction.

Second Observation

MakeNoodles(Obs2)

Component/Use, Existential Instantiation

MakePastaDish(*I2)

Abstraction

PrepareMeal(*I2)
End(*I2)

Strongest Minimality Assumption

$\forall$ x,y . End(x) $\wedge$ End(y) $\supset$ x=y

Universal Instantiation & Modus Ponens

*I1 = *I2

Substitution of Equals

MakePastaDish(*I1)

Disjointedness Assumption

$\forall$ x . $\neg$MakePastaDish(x) $\vee$ $\neg$MakeMeatDish(x)

Disjunction Elimination

$\neg$MakeMeatDish(*I1)

Abstraction

MakeChickenMarinara(*I1) $\supset$ MakeMeatDish(*I1)

Modus Tolens

$\neg$MakeChickenMarinara(*I1)

Disjunction Elimination

MakePastaDish(*I1)

Another example from this domain appears in Chapter 7 and in Appendix E (Transcripts). That example shows how one observation can constrain a different observation to be of specialized in a particular way. Specifically, a later observation of MakeMarinara constrains an earlier observation of MakeNoodles to be in fact an instance of MakeSpaghetti. Information can *flow* from one observation to another.

6.2. Indirect Speech Acts

A proper treatment of discourse requires careful attention to the representation of beliefs and intentions, as well as to the relation of an utterance to its meaning. An fully adequate treatment therefore necessitates the use of some sort of intensional or modal logic. None the less, we will try to approximate the problem of recognizing a speech act using the tools at hand, in order to suggest how the present approach might be extended to the discourse domain.

6.2.1. Representation

Recall that properties are represented by terms. The property that an agent A knows P is simply represented by applying the function **know** to terms for the agent and property P. The function **can** relates an agent and a property the agent can bring about. We assume that for every act (event) type, there is a property which holds just after an instance of the event occurs. Thus the fact that John knows at time T1 that Mary can give John the salt might be written:

$$\mathrm{Holds(know(John,can(Mary,gave(Mary,John,salt))),}\ T_1)$$

Finally we need the function **knowif**, which relate an agent and a property whose truth value the agent knows. Many objections can be raised to this notation, such as its referential transparency, and the omission of time indexes on the objects of beliefs, but it suffices for the example.

The following diagram shows part of the event hierarchy. The types SurfaceImperative and SurfaceQuestion classify utterances of commands and questions, respectively. Request and InformIf are speech act event type. A request can be specialized as a Command, a DirectRequest, or an IndirectRequest. Each of these specializations corresponds to different decomposition of the Request speech act, such as appears in [Litman 84]. Some of the discourse plans which employ speech acts are ObtainByAsking and FindOutByAsking. These discourse plans specialize more general methods for obtaining objects and finding out information. There may be a great deal of structure above, or the discourse plans themselves may be taken as End events, and thus terminate analysis. The example here does not depend on any higher structure in the library.

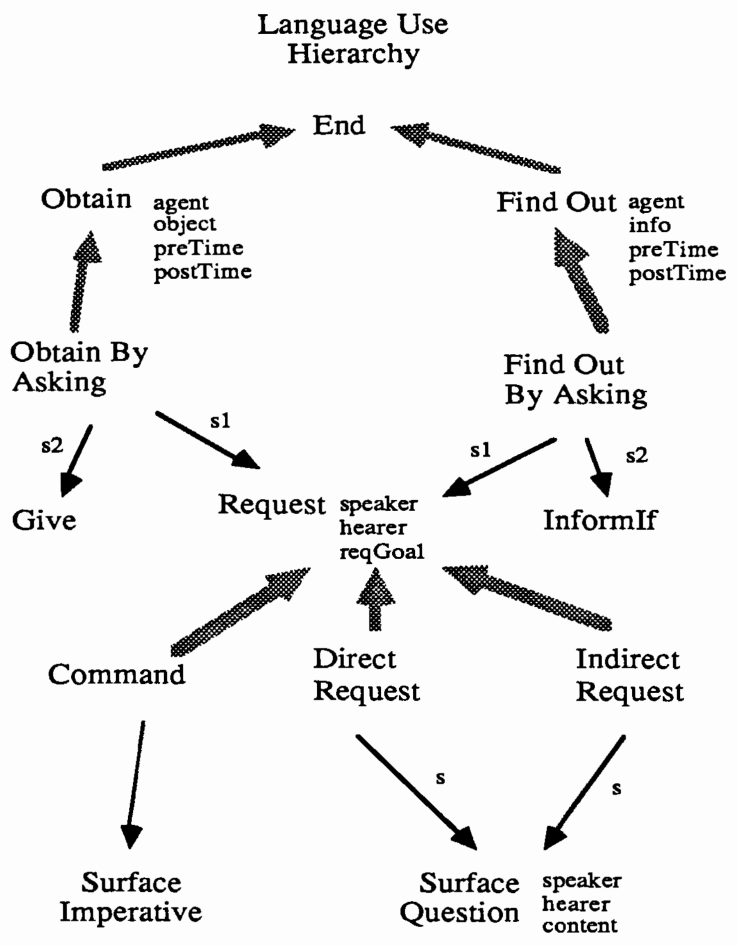


figure 6.1: Language Use Hierarchy

Following is an abbreviated list of the axioms for this hierarchy. All the actions have roles of **agent** and **time**, and other roles as specified.

• *Non-linguistic Acts:* Obtain has the role of the **object** to be obtained. During **pre-Time** the **agent** does not have the **object**; during **postTime** the **agent** does. Find-Out has the role **info**, a property, whose truth-value is to be determined. During **pre-Time** the **agent** does not **knowif** the **info** is true or false; during **postTime** the **agent** does.

$\forall$x . Obtain(x) $\supset$
    Never(have(agent(x),obj(x)), preTime(x)) $\wedge$
    meets(preTime(x), time(x)) $\wedge$ ...
    Holds(have(agent(x),obj(x)), postTime(x)) $\wedge$
    meets(time(x), postTime(x)) $\wedge$ ...

$\forall$x . FindOut(x) $\supset$
    Never(knowif(agent(x), info(x)), preTime(x)) $\wedge$
    meets(preTime(x), time(x)) $\wedge$
    Holds(knowif(agent(x), info(x)), postTime(x)) $\wedge$
    meets(time(x), postTime(x)) $\wedge$ ...

• *Discourse Acts:* The decomposition of ObtainByAsking states that the **agent** performs a Request, whose **reqGoal** is that the **hearer** gave the **agent** the **object**. This step, **s1**, is followed by step **s2**, in which the **hearer** gives the **agent** the **object**.

$\forall$x . ObtainByAsking(x) $\supset$
    Request(s1(x)) $\wedge$
    reqGoal(s1(x)) = gave(hearer(s1(x)), agent(x), obj(x)) $\wedge$
    Give(s2(x)) $\wedge$ ...

FindOutByAsking is very similar; the decomposition states that the **agent** performs a Request, whose **reqGoal** is that the **hearer** has informed the **agent** of the truth value of the **info**. In **s2** the **affected** performs an InformIf to the **agent**.

$\forall$x . FindOutByAsking(x) $\supset$
    Request(s1(x)) $\wedge$
    reqGoal(s1(x)) = informedIf(hearer(s1(x)), agent(x), info(x)) $\wedge$
    InformIf(s2(x)) $\wedge$ ...

• *Speech Acts:* Each of the specializations of a Request contains a single *utterance act* in its decomposition. The single step in a DirectRequest is a surface yes/no question. The **reqGoal** of the DirectRequest must be that the **affected** has informed the **agent** of the truth value of the (propositional) **content** of the SurfaceQuestion.

$$\begin{aligned}&\forall x \,.\, \mathrm{DirectRequest}(x) \supset \\ &\qquad \mathrm{SurfaceQuestion}(s(x)) \wedge \\ &\qquad \mathrm{reqGoal}(x) = \mathrm{informedIf}(\mathrm{hearer}(x), \mathrm{speaker}(x), \mathrm{content}(s(x))) \wedge \ldots\end{aligned}$$

The single step of an IndirectRequest is also a SurfaceQuestion; however, there is a different relation between the **content** of the question and the **reqGoal** of the request. The question must be a "can" question, of the form, "can the affected bring about the goal of the request?"

$$\begin{aligned}&\forall x \,.\, \mathrm{IndirectRequest}(x) \supset \\ &\qquad \mathrm{SurfaceQuestion}(s(x)) \wedge \\ &\qquad \mathrm{content}(s(x)) = \mathrm{can}(\mathrm{hearer}(x), \mathrm{reqGoal}(x)) \wedge \ldots\end{aligned}$$

6.2.2. Assumptions

The following component/use axioms are generated by assuming that the hierarchy is complete. Whenever a SurfaceQuestion occurs, it must be part of a DirectRequest or an IndirectRequest; and whenever a Request occurs, it must be part of an ObtainByAsking or FindOutByAsking.

$$\begin{aligned}&\forall x \,.\, \mathrm{SurfaceQuestion}(x) \supset \\ &\qquad (\exists y \,.\, \mathrm{DirectRequest}(y) \wedge x{=}s(y)) \vee \\ &\qquad (\exists y \,.\, \mathrm{IndirectRequest}(y) \wedge x{=}s(y))\end{aligned}$$

$$\begin{aligned}&\forall x \,.\, \mathrm{Request}(x) \supset \\ &\qquad (\exists y \,.\, \mathrm{ObtainByAsking}(y) \wedge x{=}s1(y)) \vee \\ &\qquad (\exists y \,.\, \mathrm{FindOutByAsking}(y) \wedge x{=}s1(y))\end{aligned}$$

6.2.3. The Problem

Suppose S says to H, "Can you give me the salt?" Real world knowledge (part of H_G) includes the statement that at all times, S knows whether or not H can give S the salt:

$$\forall t \,.\, \mathrm{Hold}(\mathrm{knowIf}(S, \mathrm{can}(H, \mathrm{gave}(H,S,\mathrm{salt}))), t)$$

An instance of a SurfaceQuestion, Q1, occurs, with the content, "H can give S the salt".

$\text{SurfaceQuestion}(Q1) \wedge \text{speaker}(Q1) = S \wedge \text{hearer}(Q1) = H \wedge$
$\text{content}(Q1) = \text{can}(H, \text{gave}(H,S,\text{salt}))$

Now apply the upward inference assumption for SurfaceQuestion. It must be case that Q1 is either a step of some DirectRequest (call it *R1) or some IndirectRequest. In either case the constraints can be used to determine the possible reqGoals. In the direct case, the goal must be for H to tell S whether or not H can give S the salt. In the indirect case, the goal must be for H to give S the salt.

$\text{DirectRequest}(*R1) \wedge Q1 = s(*R1) \wedge$
$\text{reqGoal}(*R1) = \text{informedIf}(H,S, \text{can}(H, \text{gave}(H,S,\text{salt})))$

$\vee$

$\text{IndirectRequest}(*R1) \wedge Q1 = s(*R1) \wedge$
$\text{reqGoal}(*R1) = \text{gave}(H,S,\text{salt})$

Neither alternative can (yet) be eliminated on the basis of inconsistent constraints. So we apply the second upward closure assumptions to this statement, yielding a four-way disjunction describing the act *R2 which has step *R1. Constraint information can now be used to eliminate all but one alternative.

FindOut Action

Direct Request : Precondition Fails

*R1 is a direct request, and the goal of *R1 is to be informed if H can pass the salt. *R1 is the first step of the FindOut action. However, the constraint that S does not know if H can pass the salt before the action occurs is provably false.

Indirect Request : Ill-Formed

> *R1 is an indirect request, and the goal of *R1 is for H to pass the salt. *R1 is the first step of the FindOut action. However, to be part of FindOut, the goal of *R1 must be of the form informedIf(–), instead of gave(–). Therefore this alternative is ill-formed.

Obtain Action

Direct Request : Ill-Formed

> *R1 is a direct request, and the goal of *R1 is to be informed if H can pass the salt. *R1 is the first step of the Obtain action. However, to be part of Obtain, the goal of *R1 must be of the form gave(–), instead of informedIf(–). Therefore this alternative is ill-formed.

Indirect Request : Accepted

> *R1 is an indirect request, and the goal of *R1 is for H to pass the salt. *R1 is the first step of the Obtain action. This alternative is well formed, and no violated constraints can be found.

The follow statement encodes this analysis. An ✗ appears at the point in each alternative where an inconsistency is detected. The ill-formed alternatives are ruled out by the property identity axioms discussed in Section 2.2.

```
     FindOutByAsking(*R2) ∧ *R1 = s1(*R2) ∧
     DirectRequest(*R1) ∧ Q1 = s(*R1) ∧
     reqGoal(*R1) = informedIf(H,S, can(H, gave(H,S,salt))) ∧
     reqGoal(s1(*R2)) = informedIf(H,S, info(*R2)) ∧
     info(*R2) = can(H, gave(H,S,salt)) ∧
  ✗  Never( knowif(S, can(H, gave(H,S,salt))), preTime(*R4))
∨
     FindOutByAsking(*R2) ∧ *R1 = s1(*R2) ∧
     IndirectRequest(*R1) ∧ Q1 = s(*R1) ∧
     reqGoal(*R1) = gave(H,S,salt)
  ✗  reqGoal(s1(*R2)) = informedIf(H,S, info(*R2)) ∧
∨
```

ObtainByAsking(*R2) ∧ *R1 = s1(*R2) ∧
DirectRequest(*R1) ∧ Q1 = s(*R1) ∧
reqGoal(*R1) = informedIf(H,S, can(H, gave(H,S,salt))) ∧
✗ reqGoal(s1(*R2)) = gave(H,S, obj(*R2)) ∧

∨

ObtainByAsking(*R2) ∧ *R1 = s1(*R2) ∧
IndirectRequest(*R1) ∧ Q1 = s(*R1) ∧
reqGoal(*R1) = gave(H,S,salt) ∧
reqGoal(s1(*R2)) = gave(H,S,object(*R2)) ∧
✓ object(*R2) = salt

The final disjunct must be true: The recognizer is justified in concluding that S is performing the plan to obtain the salt by asking for it.

ObtainByAsking(*R2) ∧ object(*R2) = salt ∧ *R1 = s1(*R2) ∧
IndirectRequest(*R1) ∧ reqGoal(*R1) = gave(H,S,salt) ∧ Q1 = s(*R1) ∧
SurfaceQuestion(Q1) ∧ content(Q1) = can(H, gave(H,S,salt))

6.3. Operating Systems: Multiple Events

Several research groups have examined the use of plan recognition in "smart" operating systems, which could answer user questions and/or watch what the user was doing, and make suggestions about potential pitfalls and more efficient ways of accomplishing the same tasks. A user often works on several different tasks during a single session at the terminal, and frequently jumps back and forth between uncompleted tasks. Therefore a plan recognition system for this domain must be able to handle multiple concurrent unrelated plans. The very generality of the present approach is an advantage in this domain, where the focus-type heuristics used by other plan recognition systems are not so applicable.

6.3.1. Representation

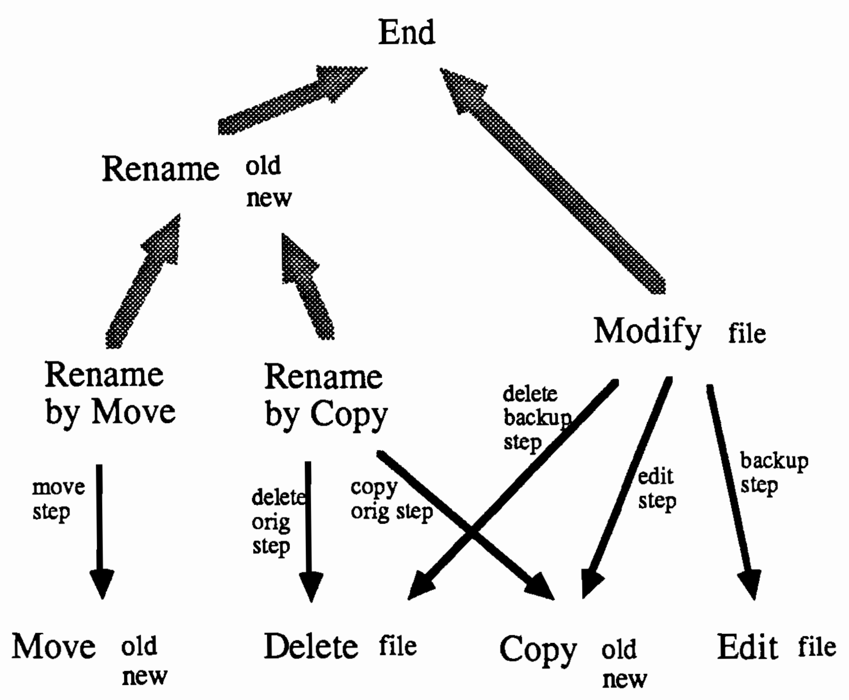


figure 6.2: Operating System Hierarchy

Consider the following hierarchy. There are two End plans: to Rename a file, and to Modify a file.

$\forall x . \text{Rename}(x) \supset \text{End}(x)$

$\forall x . \text{Modify}(x) \supset \text{End}(x)$

There are two ways to specialize the Rename event. A RenameByCopy involves Copying a file, and then Deleting the original version of the file, without making any changes in the original file.

$\forall x . \text{RenameByCopy}(x) \supset \text{Rename}(x)$

$$\begin{aligned}
&\forall x \,.\, \text{RenameByCopy}(x) \supset \\
&\quad \text{Copy}(s1(x)) \wedge \\
&\quad \text{Delete}(s2(x)) \wedge \\
&\quad \text{old}(s1(x)) = \text{old}(x) \wedge \\
&\quad \text{new}(s1(x)) = \text{new}(x) \wedge \\
&\quad \text{file}(s2(x)) = \text{old}(x) \wedge \\
&\quad \text{BeforeMeet}(\text{time}(s1(x)), \text{time}(s2(x))) \wedge \\
&\quad \text{Never}(\text{modified}(\text{old}(x)), \text{time}(x)) \wedge \\
&\quad \text{Starts}(\text{time}(s1(x)), \text{time}(x)) \wedge \\
&\quad \text{Finishes}(\text{time}(s2(x)), \text{time}(x))
\end{aligned}$$

A better way to rename a file is to RenameByMove, which simply uses the Move command. A helpful system might suggest that a user try the Move command if it recognizes many instances of RenameByCopy.

$$\forall x \,.\, \text{RenameByMove}(x) \supset \text{Rename}(x)$$

$$\begin{aligned}
&\forall x \,.\, \text{RenameByMove}(x) \supset \\
&\quad \text{Move}(s1(x)) \wedge \\
&\quad \text{old}(s1(x)) = \text{old}(x) \wedge \\
&\quad \text{new}(s1(x)) = \text{new}(x)
\end{aligned}$$

The Modify command has three steps. In the first, the original file is backed up by Copying. Then the original file is Edited. Finally, the backup copy is deleted.

$$\begin{aligned}
&\forall x \,.\, \text{Modify}(x) \supset \\
&\quad \text{Copy}(s1(x)) \wedge \\
&\quad \text{Edit}(s2(x)) \wedge \\
&\quad \text{Delete}(s3(x)) \wedge \\
&\quad \text{old}(s1(x)) = \text{file}(x) \wedge \\
&\quad \text{file}(s2(x)) = \text{file}(x) \wedge \\
&\quad \text{file}(s3(x)) = \text{new}(s1(x)) \wedge \\
&\quad \text{BeforeMeet}(\text{time}(s1(x)), \text{time}(s2(x))\,) \wedge \\
&\quad \text{BeforeMeet}(\text{time}(s2(x)), \text{time}(s3(x)))
\end{aligned}$$

6.3.2. Assumptions

Following are some of the statements obtained by minimizing the hierarchy. The Component/Use assumptions include the statement that every Copy action is either part of a RenameByCopy or of a Modify.

$$\begin{aligned} &\forall x . \text{Copy}(x) \supset \\ &\qquad (\exists y . \text{RenameByCopy}(y) \wedge x{=}s1(y)) \vee \\ &\qquad (\exists y . \text{Modify}(y) \wedge x{=}s1(y)) \end{aligned}$$

Every Delete event is either the second step of a RenameByCopy, or the third step of a Modify, in any covering model.

$$\begin{aligned} &\forall x . \text{Delete}(x) \supset \\ &\qquad (\exists y . \text{RenameByCopy}(y) \wedge x{=}s2(y)) \vee \\ &\qquad (\exists y . \text{Modify}(y) \wedge x{=}s3(y)) \end{aligned}$$

6.3.3. The Problem

Suppose the plan recognition system observes each action the user performs. Whenever a new file name is typed, the system generates a constant with the same name, and asserts that that constant is not equal to any other file name constant. (We don't allow UNIX™ style "links"!) During a session the user types the following commands.

```
(1)   % copy foo bar

(2)   % copy jack sprat

(3)   % delete foo
```

The system should recognize two concurrent plans. The first is to rename the file "foo" to "bar". The second is to either rename or modify the file "jack". Let's examine how these inferences could be made.
Statement (1) is encoded:

$$\text{Copy}(C1) \wedge \text{old}(C1){=}\text{foo} \wedge \text{new}(C1){=}\text{bar}$$

The decomposition completeness axiom for Copy lets the system infer that C1 is either part of a RenameByCopy or Modify. A new name *I1 is generated (by existential instantiation) for the disjunctively-described event.

$$\begin{array}{l} End(*I1) \land \\ (\quad (RenameByCopy(*I1) \land C1=s1(*I1)) \\ \quad \lor \\ \quad (Modify(*I1) \land C1=s1(*I1)) \\ ) \end{array}$$

Statement (2) is encoded:

$$Copy(C2) \land old(C2)=jack \land new(C2)=sprat \land Before(time(C1), time(C2))$$

Again the system creates a disjunctive description for the event *I2, which has C2 as a component.

$$\begin{array}{l} End(*I2) \land \\ (\quad (RenameByCopy(*I2) \land C2=s1(*I2)) \\ \quad \lor \\ \quad (Modify(*I2) \land C2=s1(*I2)) \\ ) \end{array}$$

The next step is to minimize the number of End events. The system might attempt to apply the strongest minimization default, that

$$\forall x,y \,.\, End(x) \land End(y) \supset x=y$$

However, doing so would lead to a contradiction. Because the types RenameByCopy and Modify are disjoint, *I1=*I2 would imply that C1=C2; however, the system knows that C1 and C2 are distinct -- among other reasons, their times are known to be not equal. The next strongest minimality default, that there are two End events, cannot lead to any new conclusions.

Statement (3), the act of deleting "foo", is encoded:

$$Delete(C3) \land file(C3)=foo \land Before(time(C2), time(C3))$$

The system infers by the decomposition completeness assumption for Delete that the user is performing a RenameByCopy or a Modify. The name *I3 is assigned to the inferred event.

$$
\begin{array}{l}
End(*I3) \wedge \\
(\quad (RenameByCopy(*I3) \wedge C3{=}s2(*I3)) \\
\quad \vee \\
\quad (Modify(*I3) \wedge C3{=}s3(*I3)) \\
)
\end{array}
$$

Again the system tries to minimize the number of End events. The second strongest minimality default says that there are no more than two End events.

$$
\begin{array}{l}
\forall x,y,z \,.\, End(x) \wedge End(y) \wedge End(z) \supset \\
\qquad x{=}y \vee x{=}z \vee y{=}z
\end{array}
$$

In this case the formula is instantiated as follows.

$$*I1{=}*I2 \vee *I1{=}*I3 \vee *I2{=}*I3$$

We've already explained why the first alternative is impossible. Thus the system knows

$$*I1{=}*I3 \vee *I2{=}*I3$$

The system then reduces this disjunction by reasoning by cases. Suppose that *I2=*I3. This would mean that the sequence

```
(2)   % copy jack sprat

(3)   % delete foo
```

is either part of a RenameByCopy or of a Modify, described as follows.

$$
\begin{array}{l}
End(*I2) \wedge \\
(\quad (RenameByCopy(*I2) \wedge C2{=}s1(*I2) \wedge C3{=}s2(*I2) \wedge \\
\text{✘} \quad old(*I2) = jack \wedge old(*I2) = foo) \\
\vee \\
\quad (Modify(*I2) \wedge C2{=}s1(*I2) \wedge C3{=}s3(*I2) \wedge \\
\quad file(s3(*I2)) = foo \wedge \\
\quad new(s1(*I2)) = sprat \wedge \\
\text{✘} \quad file(s3(*I2)) = new(s1(*I2)))) \\
)
\end{array}
$$

But both disjuncts are impossible, since the files which appear as roles of each event do not match up (as marked with ✘'s). Therefore, if the minimality default holds, it must be the case that

*I1=*I3

This means that the observations

```
(1)   % copy foo bar

(3)   % delete foo
```

should be grouped together, as part of a Rename or Modify. This assumption leads the system to conclude the disjunction:

```
End(*I1) ∧
(       (RenameByCopy(*I1) ∧ C1=s1(*I1) ∧ C3=s2(*I1) ∧
        old(*I1) = foo ∧ new(*I1) = bar )
∨
        (Modify(*I1) ∧ C1=s1(*I1) ∧ C3=s3(*I1) ∧
        file(s3(*I1)) = foo ∧
        new(s1(*I1)) = bar ∧
  ✘     file(s3(*I1)) = new(s1(*I1))) )
)
```

The second alternative is ruled out, since the actions cannot be part of the same Modify. The system concludes observations (1) and (3) make up a RenameByCopy act, and observation (2) is part of some unrelated End action.

```
End(*I1) ∧
RenameByCopy(*I1) ∧
old(*I1) = foo ∧ new(*I1) = bar ∧
End(*I2) ∧
(       (RenameByCopy(*I2) ∧ C2=s1(*I2) )
  ∨
        (Modify(*I2) ∧ C2=s1(*I2) )
)
```

Further checking of constraints may be performed, but no inconsistency will arise. At this point the plan recognizer may trigger the "advice giver" to tell the user:

```
*** You can rename a file by typing
*** % move oldname newname
```

6.4. Medical Diagnosis

As discussed in Section 1.4, there are close links between the kinds of reasoning involved in plan recognition, and that employed in medical diagnosis. The following example is drawn from [Pople 82], and was handled by CADUCEUS, an expert system that deals with internal medicine. We have made a number of simplifications, but the key points of the example remain. These include the need to consider specializations of an pathological state (abstract event type) in order to explain a symptom, finding, or state, and the process of combining or unifying the tasks invoked by each finding. This combination process corresponds to the minimization of End events in the plan recognition framework.

6.4.1. Representation

The figure below illustrates a small part of CADUCEUS's knowledge base. The thin "component" arcs are here understood as meaning "can cause". We've added the type End as an abstraction of all pathological states which are not caused by other pathological states. The *basic* specializations of End are called *specific disease entities*. We've simplified the hierarchy by making the specializations of **anemia** and **shock** specific disease entities; in the actual knowledge base, anemia and shock are caused by other conditions.

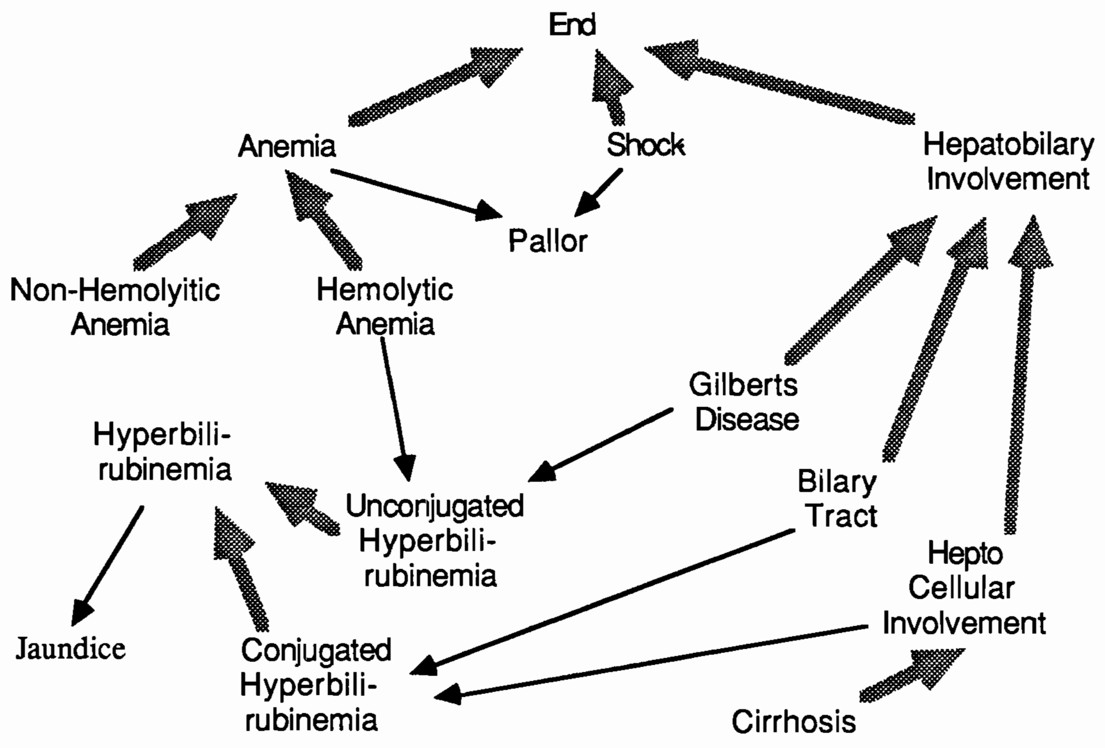


figure 6.3: Medical Hierarchy

The logical encoding of this network is as expected. The symptoms caused by a disease appear in the decomposition axiom for that disease. This is a considerable simplification over the original CADUCEUS model, in which the causal connections need only be *probable*. Symptoms of high probability, however, are taken by CADUCEUS as *necessary* manifestations, and CADUCEUS will *rule out* a disease on the basis of the *absence* of such symptoms. The *constraints* that appear at the end of a decomposition axiom would include conditions of the patient that are (very nearly) necessary for the occurrence of the disease, but are not themselves pathological. These could include the age and weight of the patient, his immunization history, and so on. Thus a constraint on Alzheimer's disease would include the fact that the patient is over 40.

A few of the axioms and assumptions follow. All kinds of **hyperbilirubinemia** cause **Jaundice** , and both **anemia** and **shock** cause **pallor** .

$$\forall y . \text{hyperbilirubinemia}(y) \supset \text{jaundice}(j(y))$$

$$\forall y . \text{anemia}(y) \supset \text{pallor}(p(y))$$

$$\forall y \,.\, \text{shock}(y) \supset \text{pallor}(p(y))$$

Unconjugated hyperbilirubinemia is a kind of **hyperbilirubinemia**, which can be caused by **hemolytic anemia**, a kind of **anemia.**

$$\forall x \,.\, \text{unconjugated-hyperbilirubinemia}(x) \supset \text{hyperbilirubinemia}(x)$$

$$\forall y \,.\, \text{hemolytic-anemia}(y) \supset \text{hyperbilirubinemia}(h(y))$$

$$\forall x \,.\, \text{hemolytic-anemia}(x) \supset \text{anemia}(x)$$

It may seem a bit odd that we need to use a first-order language, when the problem would seem to be expressible in purely propositional terms. The problem with using propositional logic arises from the abstract pathological states. A realistic medical knowledge base incorporates several methods of classifying diseases, leading to a complex and intertwined abstraction hierarchy. It is very likely that any patient will manifest at least two distinct pathological states (perhaps causally related) that specialize the same state. In a purely propositional system such a pair would appear to be competitors, as in a differential diagnosis set. But this would plainly be incorrect, if the two states were causally related.

6.4.2. Assumptions

Now we restrict our attention to the covering models of this hierarchy. The *exhaustiveness assumptions* include the fact that every case of **hyperbilirubinemia** is either **conjugated** or **unconjugated**.

$$\begin{aligned}\forall x \,.\, \text{unconjugated-hyperbilirubinemia}(x) \supset \\ &\text{conjugated-hyperbilirubinemia}(x) \vee \\ &\text{unconjugated-hyperbilirubinemia}(x)\end{aligned}$$

Disjointedness assumptions include the fact that the pathological states of **anemia, shock,** and **hepatobilary involvement** are distinct. It is important to note that this does *not* mean that the states cannot occur simultaneously; rather, that none of these states abstract each other.

$$\forall x \,.\, \neg\text{anemia}(x) \vee \neg\text{shock}(x)$$

$$\forall x . \neg \text{anemia}(x) \vee \neg \text{hepatobilary-involvement}(x)$$

$$\forall x . \neg \text{shock}(x) \vee \neg \text{hepatobilary-involvement}(x)$$

Finally, the *component/use assumptions*, better called the *manifestation/cause assumptions*, allow one to conclude the disjunction of causes of a pathological state, thus creating a *differential diagnosis* set. An important special case occurs when there is only one cause, usually at a fairly high level of abstraction, for a state. An example of this is the association of **jaundice** with **hyperbilirubinemia**. (Pople calls this case a *constrictor relationship* between the manifestation and cause, and argues that such cases play a critical role in reducing search in diagnostic problem solving.) A less conclusive assumption says that **pallor** indicates **anemia** or **shock**.

$$\forall x . \text{jaundice}(x) \supset \exists y . \text{hyperbilirubinemia}(y) \wedge x=j(y)$$

$$\begin{aligned} \forall x . \text{pallor}(x) \supset & \\ & (\exists y . \text{anemia}(y) \wedge x=p(y)) \vee \\ & (\exists y . \text{shock}(y) \wedge x=p(y)) \end{aligned}$$

6.4.3. The Problem

We'll sketch the kind of reasoning that goes on when the diagnostician is confronted with two symptoms, **jaundice** and **pallor**. From **jaundice** the diagnostician concludes **hyperbilirubinemia**. This leads (by exhaustion) to either the **conjugated** or **unconjugated** varieties. Now CADUCEUS (and perhaps a human physician?) may try to perform tests to decide between these alternatives at this point. The framework we've developed, however, allows us continue inference in each alternative. The first leads to **hemolyic anemia** and then **anemia**; the second to three different kinds of **hepatobilary involvement**. The following graph shows the final conclusion.

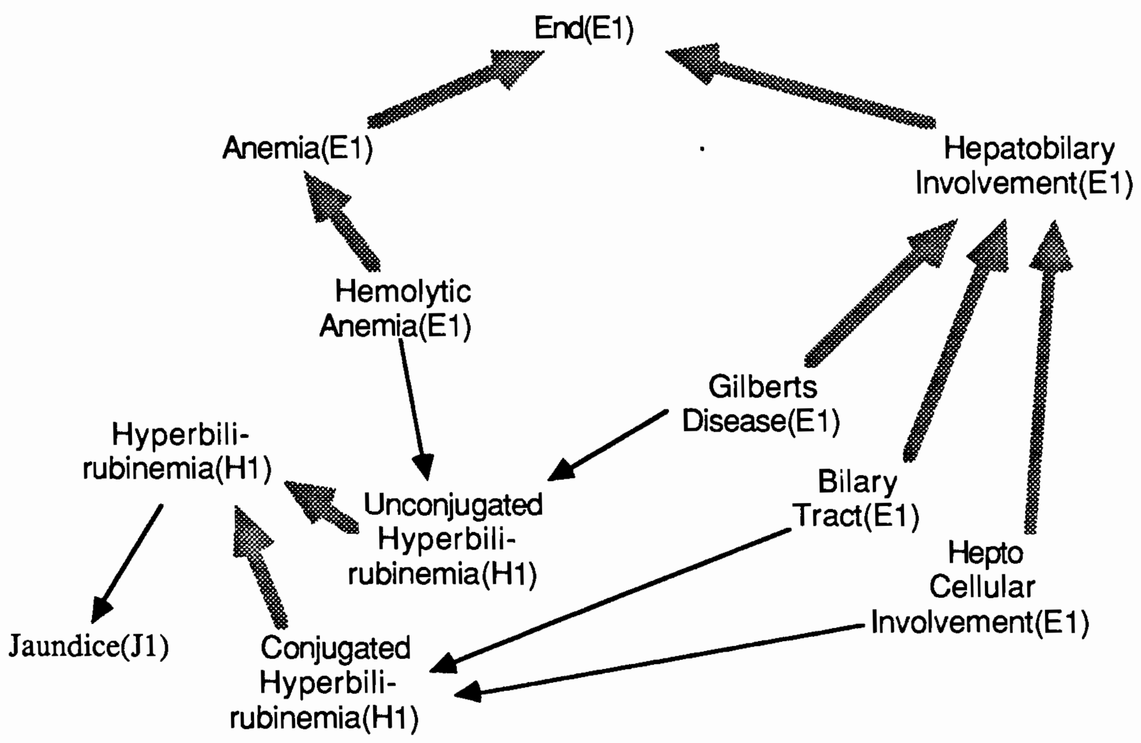

End(E1)
Anemia(E1)
Hepatobilary Involvement(E1)
Hemolytic Anemia(E1)
Gilberts Disease(E1)
Hyperbili-rubinemia(H1)
Unconjugated Hyperbili-rubinemia(H1)
Bilary Tract(E1)
Hepto Cellular Involvement(E1)
Jaundice(J1)
Conjugated Hyperbili-rubinemia(H1)

figure 6.4: Conclusions from Jaundice

The graph represents the following logical sentence. (Further steps in this example with be illustrated only in the graphical form.)

```
jaundice(*J1) ∧ hyperbilirubinemia(*H1) ∧
(       (       unconjugated-hyperbilirubinemia(*H1) ∧
                hemolytic-anemia(*E1) ∧
                anemia(*E1)
        )
        ∨
        (       conjugated-hyperbilirubinemia(*H1) ∧
                (       Gilberts-disease(*E1) ∨
                        bilary-tract(*E1) ∨
                        heptocellular-involvement(*E1)
                ) ∧
                hepatobilary-involvement(*E1)
        )
) ∧
End(*E1)
```

Next the diagnostician considers **pallor**. This leads to a simple disjunction.

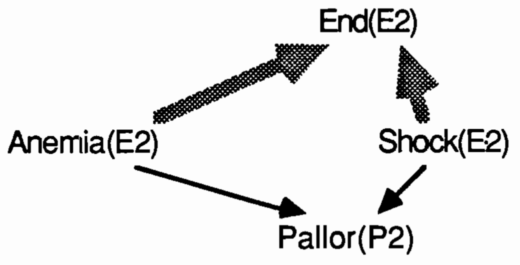


figure 6.5: Conclusions from Pallor

Finally, the diagnostician applies Occam's razor, by making the assumption that the symptoms are caused by the same disease. This corresponds to equating the specific disease entities at the highest level of abstraction (End) in each of the previous conclusions. In other words, we apply the strongest minimum cardinality default. This allows the diagnostician to conclude that the patient is suffering from **hemolytic anemia**, which has led to **unconjugated hyperbilirubinemia**.

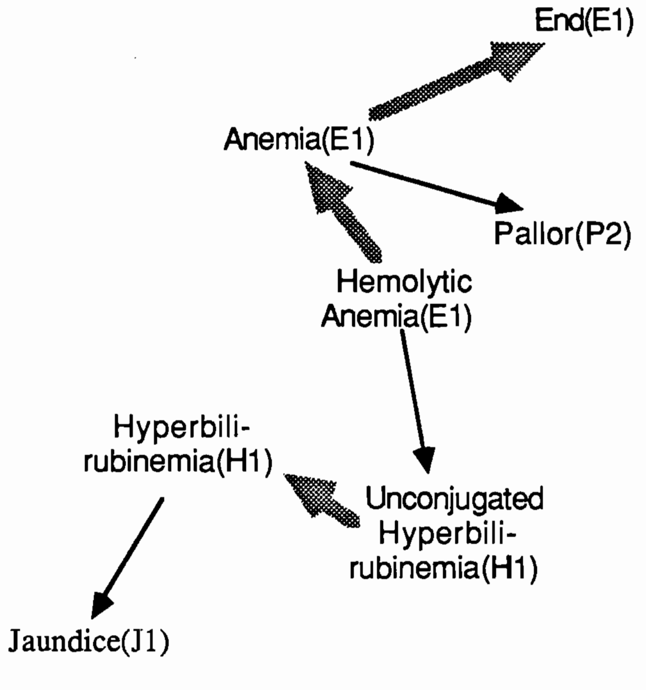


figure 6.6: Conclusions from Jaundice and Pallor

A practical medical expert system must deal with a great many problems not illustrated by this simple example. We have not dealt with the whole problem of generating appropriate *tests* for information; we cannot assume that the expert system is passively soaking in information, like a person reading a book. The full CADUCEUS knowledge base is enormous, and it is not clear whether the complete algorithms discussed in the next chapter could deal with it. It does seem plain, however, that our framework for plan recognition does formalize some key parts of diagnostic reasoning. The rich and highly structured knowledge base required for medical diagnosis makes it a very good domain for exercising any formal theory of non-deductive inference.

Chapter 7
Algorithms for Plan Recognition

7.1. Directing Inference

Two related problems arise in implementing our theory of plan recognition: inference must be *directed* toward some particular goal, and must be *limited* in some manner to insure that our programs don't run forever. The pattern of inference apparent in the previous examples suggests some answers: from each observation, apply Component/Use assumptions until an instance of type End is reached. That is, create a *proof* that some instance of End occurs. Reduce the number of alternatives (or *cases* in the proof) by checking constraints *locally*. In order to combine information from two observations, *equate* the instances of End inferred from each and propagate the equality, further reducing disjunctions. If all alternatives are eliminated, then conclude that the observations belong to distinct End events. Multiple simultaneous End events can be recognized by considering all ways of *grouping* the observations.

These operations suggest a *graph* based implementation, rather than one which stores sentences in clausal form. A graph is built bottom-up, from a node which represents an observed event, to a node which represents an End event. When we need to infer up the *abstraction* hierarchy, we create a new node of the more abstract kind, and record the the more specialized node as an *alternative for* the abstract node. If this is done properly, the result can be viewed as an and/or graph, *rooted at the End node.* The alternative (OR) arcs are marked with an equality sign (=), and the component (AND) arcs are labeled with the role function name. The figure below shows the graph that is generated an observation of MakeMarinara.

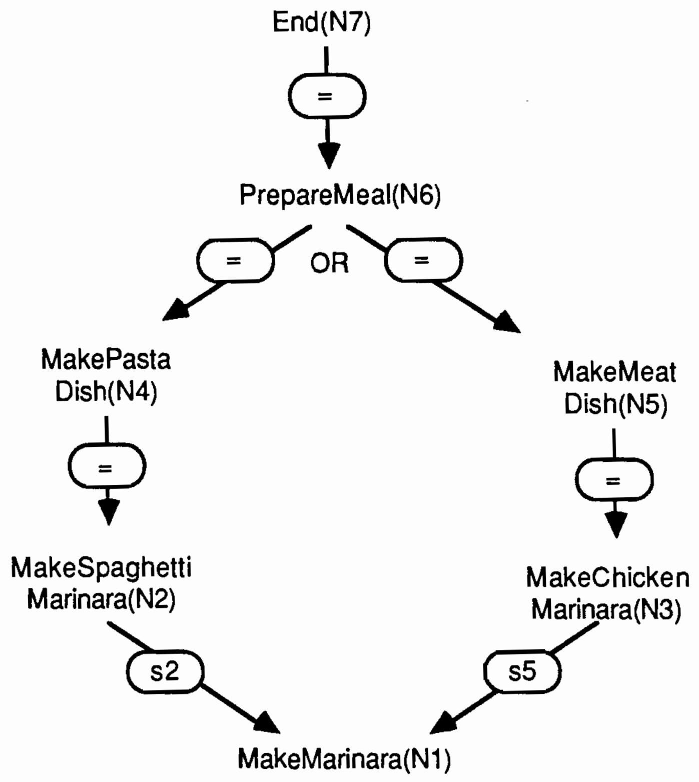


figure 7.1: E-Graph for MakeMarinara

These structures are called *explanation graphs,* or *e-graphs.* The graph has *two* complementary interpretations. It can be viewed as a *proof* that an instance of End occurs, given instances of the events in its leaves. It can also be interpreted as a *sentence* of FOL, which states that an instance of End occurs, that this instance is equal to one of a number of other tokens of more specific type, and so on. The graph above can be interpreted as the sentence

$$
\begin{aligned}
&\exists\, n_7,n_6,n_5,n_4,n_3,n_2,n_1 \;.\; \mathrm{End}(n_7) \wedge n_7=n_6 \wedge \mathrm{PrepareMeal}(n_6) \wedge \\
&\quad (\; (\; n_6=n_4 \wedge \mathrm{MakePastaDish}(n_4) \wedge \\
&\qquad n_4=n_2 \wedge \mathrm{MakeSpaghettiMarinara}(n_2) \wedge \\
&\qquad \mathrm{step2}(n_2)=n_1 \wedge \mathrm{MakeMarinara}(n_1)\;) \\
&\quad \vee (\; n_6=n_5 \wedge \mathrm{MakeMeatDish}(n_5) \wedge \\
&\qquad n_5=n_3 \wedge \mathrm{MakeChickenMarinara}(n_3) \wedge \\
&\qquad \mathrm{step5}(n_3)=n_1 \wedge \mathrm{MakeMarinara}(n_1)\;)\;)
\end{aligned}
$$

Section 7.2.3 below states the rules for translating an explanation graph into an FOL sentence. The algorithms described in this chapter provide the specialized and *deter-*

ministic proof rules for constructing explanation graphs, given some initial descriptions of event instances.

Each e-graph describes *one* End event. The occurrence of *several* End events is represented by a conjunctively-joined *set* of e-graphs, called an *hypothesis*. To account for all possible ways of *grouping* the observed events, it may be necessary to consider a *disjunctively* joined set of hypotheses. The interpretation of such a set of hypotheses is a sentence true in all minimum covering models of the observations.

7.2. Explanation Graphs

7.2.1. Basic Elements of an E-Graph

An e-graph is made up of various relations over a set of values. A value can be a **rigid designator**, a **fuzzy temporal constraint**, or a **node**.

• Rigid designators are unique names for objects; distinct rigid designators refer to distinct objects. It is a convenient philosophical fiction that proper names like "John Smith" are rigid designators. The rigid designators can be identified with a distinguished subset of FOL constants.

• A time interval can be thought of as a pair of real numbers, representing its beginning and ending instance according to some absolute scale. Yet we can rarely specify a particular time interval in so precise a fashion. Instead, we write expressions which place some relative *constraint* on the interval. Various schemes have been proposed for writing down these constraints, such as Allen's interval operators, which have been used in this thesis up to this point. A special but very useful kind of constraint can be represented by a list of four numbers. Where I is a time interval, and start-min, start-max, end-min, and end-max are numbers on some absolute temporal scale,

$$I \in (\text{start-min, start-max, end-min, end-max})$$

is true when the first instance of I falls between start-min and start-max inclusive, and the final instance of I falls between end-min and end-max inclusion. A large fragment of Allen's interval logic can be translated into an algebra over these "fuzzy" temporal constraints. Appendix D shows how this may be done. The main advantage of this approach is that intervals are all related to a single universal scale, so we do not need to maintain a table relating every interval to every other interval (as in [Allen 1983b]).

Furthermore, disjunctions in the e-graph can lead to disjunctive temporal constraints between *different sets* of intervals, and such disjunctive constraints cannot be handled by a straightforward implementation of Allen's logic.

• A node represents any object which is not given a rigid designator, and in particular, all event instances. Nodes are a distinguished subset of the constants of FOL. Every e-graph includes a function **type** which maps every node to a unique unary predicate. In particular, every event node is mapped to a member of of H_E. An e-graph has exactly *one* node of type End.

7.2.2. Roles of Event Types and of Nodes

The formulation of events described in Chapter 2 treats the roles of an event instance as the unary functions which apply to the instance. Because an event token may be of several types, it is useful to define the subset of roles of an instance which are relevant to each *particular* type of the token. For example, a token :C may be of types Change-Location and Ride-Train. The roles of :C include destination and train-number. While both roles are relevant when one thinks of :C *as* a Ride-Train, only the former makes sense when one thinks of :C *as* a Change-Location.

The association of roles with types is an important principle for organizing the construction of an e-graph. Events are abstracted in order to collapse the search space. Some of the roles of the more specialized types of the event are ignored in the process. Unless one "abstracts away" the roles in this manner, it is not possible to have different alternatives for a single abstract event. Consider the example above. The step2 role of N_2 is filled by N_1. When N_2 is viewed abstractly as N_6, the step2 role is eliminated. This allows N_6 to also abstract N_3, which uses N_1 to fill its step5 role. The exact relationship between N_6 and N_1 is left open.

Not all roles of an event are components, of course. Others include the agent of the event, its time, the objects involved, and so on. Any of these roles are also candidates for being abstracted away.

Definitions. The **roles of an event type** E are the unary functions which appear in the decomposition axiom for any type which abstracts* E, and apply to the variable which appears in the main antecedent of the axiom. The **component roles of an event type** are the roles of the event type which yield an event token when applied to an event token. The **parameters of an event type** are all the roles of the event type which are *not* component roles.

In an explanation graph, every node corresponding to an event token has a unique associated type. Therefore we can talk of the pairs of role functions and role values of a node as follows.

Definitions. A **roleval** of a node N is a pair (f_r,v) such that

1. f_r is a role of type(N).

2. v is a value, as defined above.

3. v=f_r(N).

The **known rolevals of N** are simply the rolevals of N for which we can compute the value v. Sometimes we speak of a roleval without reference to a particular node, meaning simply a (f_r,v) pair. The **component** and **parameter rolevals** of N are defined in the obvious way.

7.2.3. Definition of an Explanation Graph

An **explanation graph** G = (Vr, Vt, Vn, Type, R, Rolevals, Alt), where:

Vr is the set of rigid designators.

Vt is the set of fuzzy temporal constraints.

Vn is the set of nodes.

Type is a function from nodes to event types.
$\text{Type: Vn} \Rightarrow H_E$

R is the set of roles.

Rolevals is the known roleval relation, over node, roles, and values.
$\text{Rolevals} \subset \text{Vn x R x } \{\text{Vr} \cup \text{Vt} \cup \text{Vn}\}$

Alt is the alternatives-for function, from nodes to sets of nodes.
$\text{Alt : Vn} \Rightarrow 2^{\text{Vn}}$

An interpretation of a node n in G is the following sentence:

$$\begin{aligned}
&\text{Interpretation}(n) = \\
&\qquad \{E_i(n) \mid \text{type}(n)=E_i\} \wedge \\
&\qquad \bigwedge\{v=f_r(n) \wedge S \mid (n,f_r,v) \in \text{Rolevals} \wedge v \in Vn \wedge \\
&\qquad\qquad\qquad S = \text{Interpretation}(v)\} \wedge \\
&\qquad \bigwedge\{v=f_r(n) \mid (n,f_r,v) \in \text{Rolevals} \wedge v \in Vr\} \wedge \\
&\qquad \bigwedge\{f_r(n) \in v \mid (n,f_r,v) \in \text{Rolevals} \wedge v \in Vt\} \wedge \\
&\qquad \bigvee\{n=m \wedge S \mid m \in \text{Alt}(n) \wedge S = \text{Interpretation}(m)\}
\end{aligned}$$

The sentence states that n is of the specified type; that it has the specified values as roles; and that it must be equal to one of a set of other nodes.The interpretation of an e-graph is the interpretation of its end node, with nodes replaced by existentially-quantified variables.

$$\begin{aligned}
&\text{Interpretation}(G) = \\
&\qquad \exists\, x_1, \dots x_k \,.\, \text{Interpretation}(h)\ \theta \\
&\text{where} \\
&\qquad h \in Vn \wedge \text{type}(h)=\text{End} \\
&\qquad \theta = (n_1/x_1, \dots n_k/x_k) \\
&\qquad \{n_1, \dots n_k\} = Vn
\end{aligned}$$

It is important to understand that an e-graph, unlike a lexical hierarchy, has existential import. It is used to represent the fact that some particular tokens are of some types, and form components of other tokens. A lexical hierarchy has no existential import. It only asserts that if there are any tokens of some type, then they all must have certain properties.

The treatment of the alternative relation distinguishes e-graphs from previous work that uses "semantic nets" to represent sentences. Systems such as KL-1 [Brachman 85] can only represent conjunctions of atomic assertions. There is no way to assert that a particular entity is *either* of type E_1 *or* of type E_2. More recent work on semantic nets, such as KRYPTON [Brachman, Fikes, & Levesque 85], has gained the ability to make disjunctive assertions by adding a non-network component which holds sentences containing arbitrary disjunctions and conjunctions. Such systems use a general-purpose theorem prover to make inferences from these assertions. General theorem proving is both theoretically and practically intractable, and little is known about controlling and focusing their power. The special form of an e-graph, however, lets us closely control specific kinds of reasoning with disjunctions.

7.3. Computing the Uses of an Event Type

As we have hinted above, the first stage of the recognition algorithm is to prove that an observed event token is a component of an End event, by considering all the possible *uses* of the observed event. The Uses relation is roughly – but only roughly – the inverse of the direct component relation. Starting from an event of a particular type – say, GetNewspaper(C_1) – we consider all the uses of C_1 *as* a GetNewspaper – such as a step of BecomeInformed. Then we consider the uses of C_1 for types which abstract and specialize GetNewspaper. So if GetPaper abstracts GetNewspaper, we then consider the uses of C_1 *as* a GetPaper, which *differ* from the uses already considered – such as a step of BuildFire. (As will be seen below, we'd actually use different nodes for each abstraction and specialization of C_1.) Finally, it may be necessary to consider uses of types which specialize the type of the original observation. To allow for the *possibility* that C_1 could be an instance of GetSleazyTabloid, we consider the use of the observed event as a step of EnjoyVicariousTitillation.

7.3.1. Use Abstraction and Specialization

Just as one can talk of one type abstracting another, one can talk of abstraction of the component relationship between types. In the cooking world, the relationship between MakeSauce and MakePastaDish abstracts the relationship between MakeMarinara and MakeSpaghettiMarinara. In order to explain an instance of MakeMarinara, one considers MakeSpaghettiMarinara and MakeChickenMarinara. It is not necessary to consider the path from MakeMarinara to MakeSauce to MakePastaDish, because that way of using the instance abstracts a more specific use of the instance.

A *use* is a triple, (E_c, f_r, E_u), and stands for the possibility that (some instance) of E_c could fill the f_r role of some instance of E_u. For example, when trying to explain event instance C_1, this use would mean

$$\exists\, y \;.\; E_u(y) \wedge C_1 = f_r(y) \wedge E_c(C_1)$$

The abstraction relation between uses is defined as follows.

Definition. (E_{ac}, f_r, E_{au}) **abstracts** (E_c, f_r, E_u) exactly when E_{ac} abstracts E_c, and E_{au} abstracts E_u. When U_1 is either the same as or abstracts U_2, we say U_1 abstracts* U_2. Specializes and specializes* are the inverse of abstracts and abstracts* respectively.

7.3.2. The Uses and Direct Component Relations

It is sometimes necessary to consider uses of a type of which that type is *not* a direct component. Consider the following extension to the cooking hierarchy.

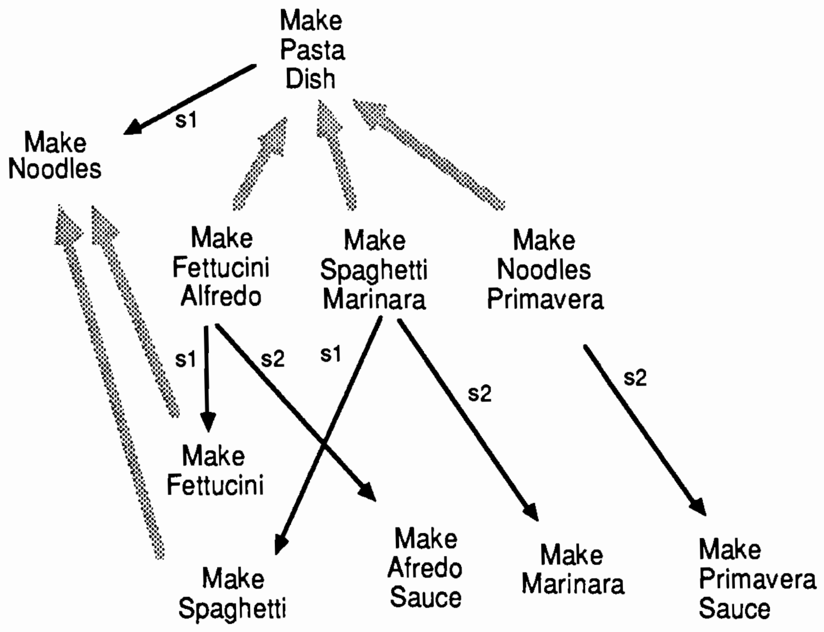


figure 7.2: Extended Cooking Hierarchy Detail

To explain an instance of MakeSpaghetti, one considers the use (MakeSpaghetti, s_1, MakeSpaghettiMarinara). The use (MakeNoodles, s_1, MakePastaDish) is not considered, because it abstracts the former use. But then a possibility is unfairly ignored! An event of type MakeSpaghetti could also be a component of an event of type MakeNoodlesPrimavera, because that type does not further specify the type of its s_1 role. Therefore the set of Uses to consider should include (MakeSpaghetti, s_1, MakeNoodlesPrimavera).

The next section precisely specifies how Uses augments the set of direct components, in order to guarantee that the recognition algorithms are correct. The set Uses needs only be computed once and for all, *before* any observations are input. Therefore we need not be concerned with finding an efficient algorithm to compute Uses; practically any exhaustive, off-line method is sufficient.

7.3.3. Definition of Uses

Given a component event type, Uses yields a very minimal set of use types which describes those types whose instances could have a component of the given type, but do not *require* the component to be of a *particular* subtype of the given type.

A set of types *describes* another set of types if every member of the latter set specializes* or abstracts* a member of the former set. A set of types is *minimal* if no member abstracts another member, and is *very minimal* if it also does not contain a subset $\{E_1, \ldots, E_n\}$ which forms an exhaustive set of specializations for some type E_0. (A very minimal set should contain E_0 instead.) We define Uses in four steps, as follows.

$$\begin{aligned}U_\alpha = \{ (E_c, f_r, E_u) \mid \exists\, E_{du}, E_{ac}, E_{au}\,.\ & E_c \text{ is the } f_r \text{ direct component of } E_{du} \wedge \\ & E_{ac} \text{ abstracts* } E_c \wedge \\ & E_{ac} \text{ is the } f_r \text{ direct component of } E_{au} \wedge \\ & E_{au} \text{ abstracts* } E_u \}\end{aligned}$$

$$\begin{aligned}U_\beta = \{(E_c, f_r, E_u) \mid\ & (E_c,f_r,E_u) \in U_\alpha \wedge \\ & H \cup cl(H) \vdash \forall x\ .\ E_u(x) \supset \neg E_c(f_r(x))\}\end{aligned}$$

$$\begin{aligned}U_\chi = \{(E_c, f_r, E_u) \mid\ & (E_c,f_r,E_u) \in U_\beta \wedge \\ & \forall E_{su}\,.\ E_u \text{ abstracts } E_{su} \supset (E_c, f_r, E_{su}) \in U_\beta \}\end{aligned}$$

$$\begin{aligned}\text{Uses} = \{(E_c, f_r, E_u) \mid\ & (E_c,f_r,E_u) \in U_\chi \wedge \\ & \neg\exists\, E_{au}\,.\ E_{au} \text{ abstracts } E_u \wedge (E_c, f_r, E_{au}) \in U_\chi \}\end{aligned}$$

This rather complicated construction can be understood as follows. We start by considering E_c which have some explicit direct uses. Then we add uses from E_c to all the types which directly use (in the same role) abstractions of E_c. Next we add uses from E_c to all types which specialize any of the previous uses. (This is to catch uses like (MakeSpaghetti, s_1, MakeNoodlesPrimavera) above.) This large set is U_α. Next we throw out those uses which are not in fact possible, but arose in the specialization step. This yields U_β. The next two steps redescribe U_β in simpler terms. In U_χ we throw out all the uses where the user type has *some* specializations which *cannot* have components of type E_c. Then in Uses we eliminate a set of uses if they exhaustively specialize a use already under consideration. This corresponds to making uses *very minimal*, as described above.

7.3.4. Example

Following are the uses calculated at each stage for MakeNoodles and MakeSpaghetti, using the extended hierarchy above.

U_α = { (MakeNoodles, s1, MakePastaDish)
(MakeNoodles, s1, MakeFettuciniAlfredo)
(MakeNoodles, s1, MakeSpaghettiMarinara)
(MakeNoodles, s1, MakeNoodlesPrimavera)
(MakeSpaghetti, s1, MakeSpaghettiMarinara)
(MakeSpaghetti, s1, MakePastaDish)
(MakeSpaghetti, s1, MakeFettuciniAlfredo)
(MakeSpaghetti, s1, MakeNoodlesPrimavera) }

Next, eliminate the impossible uses.

U_β = { (MakeNoodles, s1, MakePastaDish)
(MakeNoodles, s1, MakeFettuciniAlfredo)
(MakeNoodles, s1, MakeSpaghettiMarinara)
(MakeNoodles, s1, MakeNoodlesPrimavera)
(MakeSpaghetti, s1, MakeSpaghettiMarinara)
(MakeSpaghetti, s1, MakePastaDish)
(MakeSpaghetti, s1, MakeNoodlesPrimavera) }

Throw out uses where some specializations of the user type are not uses.

U_χ = { (MakeNoodles, s1, MakePastaDish)
(MakeNoodles, s1, MakeFettuciniAlfredo)
(MakeNoodles, s1, MakeSpaghettiMarinara)
(MakeNoodles, s1, MakeNoodlesPrimavera)
(MakeSpaghetti, s1, MakeSpaghettiMarinara)
(MakeSpaghetti, s1, MakeNoodlesPrimavera) }

Simplify this set by eliminating sets of uses which exhaustively specialize another use.

Uses = { (MakeNoodles, s1, MakePastaDish)
(MakeSpaghetti, s1, MakeSpaghettiMarinara)
(MakeSpaghetti, s1, MakeNoodlesPrimavera) }

Thus from MakeNoodles one should consider the use MakePastaDish, while two search paths start from MakeSpaghetti, one through MakeSpaghettiMarinara and the other through MakeNoodlesPrimavera.

7.4. Constraint Checking

The algorithms need to check the various constraints which appear in the decomposition axioms for the event types. It is never necessary to prove that a constraint actually holds; rather, one should *fail* to prove that the constraint is false. As a side effect of constraint checking, the values of parameters are propagated to a node from its components. In Section 2.6.3, constraints were categorized as equality constraints, temporal constraints, and preconditions and effects; the latter two jointly called *fact* constraints. Each kind of constraint is handled in a different manner.

7.4.1. Equality Constraints

An equality constraint usually equates a parameter of a node with a parameter of one of its components. An equality constraint fails if the values of the two items are distinct rigid designators. As a side effect, when a value is known for the parameter of the component, but none for the parameter of the node, the value is assigned to the parameter of the node.

7.4.2. Temporal Constraints

Rather than passing around precise values for time parameters, the implementation passes fuzzy temporal constraints. All unknown times are implicitly constrained by

$$(-\infty \ +\infty \quad -\infty \ +\infty)$$

A temporal constraint fails if a time parameter is assigned an *empty* fuzzy constraint, such as

(5 6 2 3)

There can be no time interval which starts between times 5 and 6, and ends between times 2 and 3. (Time doesn't run backwards!) Corresponding to each binary temporal relation, such as Before or During, there an algebraic operation on fuzzy constraints, with the following property:

Where T_1, T_2 are time intervals, and Z_1,Z_2 are fuzzy constraints:

$$T_1 \in Z_1 \wedge T_2 \in Z_2 \wedge T_1 \text{ binary-op } T_2 \supset$$
$$T_1 \in \text{Ftransitive}(Z_1, \text{binary-op}, Z_2)$$

This algebra is described in Appendix D. These rules are used to update the fuzzy constraints assigned to the time parameters of the node under consideration. The appendix also defines the functions **Fintersection** and **empty**, which are used in the algorithms.

7.4.3. Fact Constraints

Fact constraints are checked only if values are known for all the arguments of the predicates in the constraint. A limited theorem prover attempts to prove the negation of the fact. The constraint fails if the proof succeeds. In order to check a fact of the form

$$\text{Never}(T_1, \text{property})$$

the theorem prover attempts to show that there exists a time T_2 over which the property holds, and T_1 *must* intersect T_2, in order to satisfy their respective fuzzy temporal constraints.

The implementation assumes that the general domain axioms H_G are such as to guarantee finite failure.

7.5. Overview of the Algorithms

The algorithms are presented in a structured pseudo-code, freely mixing English and programming constructs in order to maximize clarity. All functional parameters are passed by value. There are three basic algorithms: **explain-observation, match-graphs,** and **group-observations**. We present an overview of the algorithms, then the pseudo-code, and finally a more detailed description of their operation.

7.5.1. Explain

The function call **explain-observation(etype,edescr)** builds an e-graph on the basis of a single observation of an event of etype, which satisfies the role/value pairs in edescr. Two versions of the subroutine which checks for redundant search paths are presented. The first assumes that the abstraction hierarchy forms a tree. The more general version contains a slightly more complex test to avoid finding redundant paths from a node to End.

Following is an example of the "multi-path problem" with multiple inheritance. Consider the following fragment of an event hierarchy.

$$\begin{tikzcd}
 & & & L \arrow[dl, "s1"] & \\
M \arrow[urrr] \arrow[dd, "s1"] & & D & & K \arrow[ul] \arrow[dl, "s1"] \\
 & B \arrow[ur] & & C \arrow[ul] & \\
N \arrow[ur] & & A \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

figure 7.3: Multiple Inheritance Hierarchy

Suppose the *primary* event to be explained is of type A. Then one should consider the use (C, s_1, K) but *not* the more general use (D, s_1, L). (Why? Because of the exhaustiveness assumption, every L is either an M or a K. M's must have components of type N, and not of type A. Every C is an A. Therefore, from C it is safe to infer K. The weaker conclusion of L is not an *alternative* conclusion.) On the other hand, if the primary node is of type B, then one should consider (D, s_1, L) but not the more specific (C, s_1, K). (Why? Because the B could be an N or an A, so the best one can do is conclude L.) The potentially expensive test in the more general version of **redundant** "looks back" at the primary node to see whether or not a different abstraction path from the primary node could reach a more specific use than the one at the current node.

An important feature of the algorithm is the test involving the **consider-spec** flag. In order to explain an event of etype, one must consider event types that (1) abstract etype; (2) specialize etype; and (3) abstract types which specialize etype. We must *not* consider types that specialize types that abstract etype – because this includes *all* of H_E.

7.5.2. Match

The function call **match-graphs(graph1,graph2)** builds a new graph which equates the End nodes of graph1 and graph2. The match algorithm assumes that there exists either no or exactly one type which is the "greatest lower bound" of any pair of elements of H_E. Formally:

E_1 and E_2 are compatible $\supset$

$$\exists E_3 \in H_E . E_1 \text{ abstracts* } E_3 \wedge E_2 \text{ abstracts* } E_3 \wedge$$
$$\forall E_4 . E_1 \text{ abstracts* } E_4 \wedge E_1 \text{ abstracts* } E_4 \supset$$
$$E_3 \text{ abstracts* } E_4$$

A technical term for this is that H_E is a lower semi-lattice. The glb of non-compatible types is by definition $\varnothing$.

The key feature of match-graphs is the use of a cache to store matches between nodes. Because e-graphs are digraphs, rather than trees, it is possible for the same pair of nodes to be visited more than once in the matching process. The node created the first time the pair is visited is used as the result of any subsequent match of the pair. Without this feature, match-graphs would multiply the input digraphs into a tree, possibly causing an exponential increase in the size of the result over the size of the inputs.

7.5.3. Group

The function **group-observations** continually inputs observations and groups them into sets to be accounted for by particular e-graphs. The function **minimum-Hypoths** is called to retrieve a disjunctive set of current hypotheses, each of which is a conjunctive set of e-graphs which accounts for all of the observations, using as few End events as possible.

There are three versions of this algorithm. The **Non-Dichronic** algorithm returns the same results regardless of the order in which the observations are input, and computes conclusions *mc-entailed* by the inputs. The **Incremental Minimization** and **Sticky** versions correspond to the *incremental* theories of recognition discussed in Chapter 5. The **Incremental Minimization** algorithm only increases the number of different End events under consideration when it has to, and retains all previous conclusions. It implements the *imc-entailment* operator. The **Sticky** algorithm tries to explain each observation by making it part of the most recently invoked End event.

7.6. Pseudo-Code

7.6.1. Utility Functions

```
                    Utility Functions
            on Nodes of Explanation Graphs

function type(node)
        /* most specific type assigned to the node */

function roleval(node, role)
        /* the value of role(node) */

function alternatives-for(node)
        /* a set {$n_1$, $n_2$, ..., $n_j$} such that
        $node=n_1 \vee node=n_2 \vee ... \vee node=n_j$ */

function status(node)
        /* initially True for all nodes, set to False when the alternative con-
        taining the node is ruled out */
```

7.6.2. Explain-Observation

```
                    Explain-Observation

global egraph /* stores the explanation graph built by explain */

/* explain-observation
        etype : type of observed event
        edescr : role/value pairs of observed event
returns
        egraph : explanation graph rooted at End-node-of(graph)
*/
function explain-observation(etype, edescr) is
        initialize egraph
        explain(etype, edescr, {etype}, True, etype)
        return egraph
end build-explanation-graph
```

```
/* explain
        etype : type of event to be explained
        edescr : list of (role value) pairs which describe the event
        visited : set of event types considered so far
        consider-spec : if true, then consider specializations of etype
        primary : the type at which search to explain this event
                instance began
returns
        node : represents event token of etype
        pathToEndFound : true if successfully explained event
        allVisited : visited augmented with types considered which
                abstract etype
*/
function explain(etype, edescr, visited, consider-spec, primary) is
        explained := False /* need to find a path to End */
        visited := visited ∪ {etype}
        /* Merge search paths if possible */
        if egraph has a node n of etype which exactly matches edescr then
                return (n, status(n), visited) endif
        /* Can't merge, must consider a new node */
        create a new node n of etype which satisfies edescr in egraph
        if etype=End then return (n, True, visited) endif
        propagate equality, temporal, and fact constraints at n
        if constraints violated then
                status(n) := False
                return (n, False, visited) endif
        /* Consider direct Uses of etype */
        forall (etype, r, utype) ∈ Uses
                /* eliminate unnecessary Uses */
                if ¬redundant(etype, r, utype, visited, primary) then
                        (–,foundpath,–) := explain(utype, {(r,n)}, ∅, True, utype)
                                /* the r role of the event must be n */
                        explained := foundpath or explained
                endif
        endfor
        /* Consider abstractions of etype */
        forall atype ∈ direct-abstractions(etype)
                if atype ∉ visited then
                                /* eliminate unnecessary abstractions */
                        let adescr be the role/value pairs computed for n,
                                limited to the non-step roles defined for atype
                        (an,foundpath,visited) :=
                                explain(atype, adescr, visited, False, primary)
                        add n as an alternative for an
                        explained := foundpath or explained
```

```
            endif
        endif
        /* Consider specializations of etype */
        if consider-spec then
                        /* don't abstract up and then specialize back down */
            forall stype ∈ direct-specializations(etype)
                (–,foundpath,–) :=
                    explain(stype, edescr, visited, True, primary)
                explained := foundpath or explained
            endfor
        endif
        status(n) := explained
        return (n, explained, visited)
end explain
```

Check for Redundant Paths
(called by Explain)

```
/* Tree Version */
function redundant(etype, r, utype, visited, primary) is
        return (etype, r, utype) abstracts or specializes a use
                for some member of visited
end redundant

/* Multiple-Inheritance Version */
function redundant(etype, r, utype, visited, primary) is
        return
            ((etype, r, utype) abstracts a use for a type which
                abstracts*  primary) ∨
            ((etype, r, utype) specializes a use for a type which
                specializes*  primary) ∨
            (etype does not abstract* or specialize* primary ∧
                (etype, r, utype) specializes a use for a type which
                    abstracts  primary) ∨
            (etype does not abstract* or specialize* primary ∧
                (etype, r, utype) abstracts a use for a member of
                    visited ∧
                (etype, r, utype) does not specialize a use for type
                    which abstracts primary)
end redundant
```

7.6.3. Match-Graphs

Match-Graphs

```
global mgraph /* stores the graph created by match */

/* match-graphs
        graph1, graph2 : graphs to be matched
returns
        graph3 :  represents equating End nodes of graphs 1 and 2
                and propagating equality
        succeeded : True if match succeeded, false otherwise
*/

function match-graphs(graph1, graph2) is
        initialize mgraph
        (–,succeeded ) := match(End-node-of(graph1), End-node-of(graph2))
        return (mgraph, succeeded )
end match-graphs

hash-table previous-match /* cache for results of each call to match */

/* match
        n1 , n2 : values to be matched (either  nodes, time intervals, or
                fuzzy temporal constraints)
returns
        n3 : value which represents the result of the match
        succeeded : if true, then n1 and n2 could be equal
*/
function match(n1, n2) is
        if n1 and n2 are rigid designators then
                return (n1, n1=n2)
        elseif n1 and n2 are fuzzy temporal constraints then
                return ( Fintersection(n1,n2), ¬empty(Fintersection(n1,n2)) )
        else /* n1 and n2 are nodes */
                /* if n1 and n2 have already been matched, reuse that result */
                n3 := previous-match(n1,n2)
                if n3 ≠ ∅ then return (n3, status(n3)) endif
                /* check that types of n1 and n2 are compatible */
                n3type := greatest lower bound of {type(n1), type(n2)}
                if n3type = ∅  then return (–,False) endif
                create new node n3 of n3type in mgraph
                /* add all known role/values of n1 and n2 to n3, matching
                        identical roles */
                forall roles r defined for n3type
                        v1 := roleval(n1,r)
                        v2 := roleval(n2,r)
                        if either v1 or v2 is defined then
                                if v1 is defined but not v2 then
                                        (v3, okay) := match(v1,v1)
                                elseif v2 is defined but not v1 then
                                        (v3, okay) := match(v2,v2)
                                elseif
                                        (v3, okay) := match(v1,v2)
                                endif
                                if ¬okay then return (n3, False) endif
                                add v3 as the r role of n3
                        endif
                endfor
                check equality, temporal, and fact constraints on n3
                if constraints violated then
                        status(n3) := False
                        return (n3, False) endif
                /* match alternatives for each node */
                if either n1 or n2 has an alternative then
                        for a1 ∈ alternatives-for(n1)
                                        (or a1=n1 if no alternatives)
                                for a2 ∈ alternatives-for(n2)
                                                (or a2=n2 if no alternatives)
                                        (a3, okay) := match(a1, a2)
                                        if okay then
                                                add match(a1,a2) as a alternative
                                                for n3
                                        endif
                                endfor
                        endfor
                endif
                if there were alternatives of n1 or n2, but none matched then
                        return (n3,False)
                else return (n3,True) endif
        endif
end match.
```

7.6.4. Group Observations

7.6.4.1. Non-Dichronic Version

Group-Observations
Non-Dichronic Version

```
global Hypoths
        /* A set (disjunction) of hypotheses, each a set (conjunction) of ex-
        planation graphs.  Each hypothesis corresponds to one way of
        grouping the observations.  Different hypotheses may have different
        cardinalities. */

function minimum-Hypoths is
        smallest := min { card(hypo) | hypo ∈ Hypoths}
        return { hypo | hypo ∈ Hypoths ∧ card(hypo)=smallest }
end minimum-Hypoths

procedure group-observations is
        Hypoths := {∅}
        while more observations
                obtain observation (etype, edescr)
                obs-graph := explain-observation(etype, edescr)
                forall hypo ∈ Hypoths
                        remove hypo from Hypoths
                        add hypo ∪ obs-graph to Hypoths
                        forall g ∈ hypo
                                (new-g, okay) := match-graphs(obs-graph, g)
                                if okay then
                                        add (hypo – {g}) ∪ new-g to Hypoths
                                endif
                        endfor
                endfor
        endwhile
end group-observations
```

7.6.4.2. Incremental Minimization Version

Group-Observations
Incremental Minimization Version

```
global Hypoths
    /* A set (disjunction) of hypotheses, each a set (conjunction) of ex-
    planation graphs. Each hypothesis corresponds to one way of
    grouping the observations. All hypotheses have the same cardinali-
    ty, corresponding to the minimum number of End events */

function minimum-Hypoths is
    return Hypoths
end minimum-Hypoths

procedure group-observations is
    Hypoths := {∅}
    while more observations
        obtain observation (etype, edescr)
        obs-graph := explain-observation(etype, edescr)
        old-hypoths := Hypoths
        forall hypo ∈ Hypoths
            remove hypo from Hypoths
            forall g ∈ hypo
                (new-g, okay) := match-graphs(obs-graph, g)
                if okay then
                    add (hypo – {g}) ∪ new-g to Hypoths
                endif
            endfor
        endfor
        if Hypoths=∅ then
            forall hypo ∈ Hypoths
                add obs-graph to hypo
            endfor
        endif
    endwhile
end group-observations
```

7.6.4.3. Sticky Version

Group-Observations
Sticky Version

```
global hypo
    /* An ordered list of e-graphs, from most recent to least recent, repre-
    senting a single hypothesis */
```

```
function minimum-Hypoths is
    return {hypo}
end minimum-Hypoths

procedure group-observations is
    hypo := NIL
    while more observations
        obtain observation (etype, edescr)
        obs-graph := explain-observation(etype, edescr)
        block update
            foreach g ∈ hypo in order
                (new-g, okay) := match-graphs(obs-graph, g)
                if okay then
                    hypo := cons(new-g,  remove(g, hypo))
                    exit block update
                endif
            endfor
            hypo := cons(obs-graph, hypo)
        endblock update
    endwhile
end group-observations
```

7.7. Description of Operation

The following section steps through and explains the operation of the algorithms. The transcripts in Appendix E contain detailed traces of **Explain** and **Match** on many of the specific examples discussed in this thesis.

7.7.1. Explain

• Check whether the graph under construction already contains a node of **etype** which exactly matches **edescr**, and contains no additional parameters. If this is the case, then the graph merges at this point. Consider the e-graph at the beginning of this chapter. Suppose the left-hand side of the graph has been constructed (nodes N1, N2, N4, N6, and N7). Search is proceeding along the right-hand part of the graph, through N3. The invocation of **explain** which created MakeMeatDish(N5) is considering abstractions of MakeMeatDish (see below), and recursively calls **explain** with type Prepare-Meal. The specific call would be:

explain(PrepareMeal, {...}, {MakeMeatDish}, False, MakeMeatDish)

This description exactly matches previously-created node N6, which is returned. Then N5 is made an alternative for N6. Thus the left path through N2 and N4 merges with the right path through N3 and N5. This kind of merging can prevent combinatorial growth in the size of the graph.

• Create a new node of type **etype**, and link all the role/value pairs in **edescr** to it.

• Check whether the type of the newly-created node is End. If so, then no further explanation is possible, and so return.

• Propagate and check constraints. Suppose this is the invocation of **explain** which created MakeSpaghettiMarinara(N2). **Edescr** is {(step2 N1)}, meaning that component step2 of N2 is N1. The equality constraints say that the agent of any MakeSpaghettiMarinara must equal the agent of its MakeMarinara step. (This constraint is inherited from the more general one given for MakePastaDish.) If initially (N1 agent Joe) appears in the graph, after this step (N2 agent Joe) also appears.

Fuzzy time constraints are also propagated. N2 is constrained to occur over an interval which contains the time of N1. Suppose the graph initially contains (N1 time (4 5 6 7)). After this step, it also contains (N2 time ($-\infty$ 5 6 $+\infty$)).

This step can also eliminate nodes, marking them as failed. The agent of every specialization of MakePastaDish is constrained to be dexterous. Suppose the general world knowledge base contains the assertion $\neg$Dexterous(Joe). Then this step fails for this invocation, and **explain** sets the status of N2 to false and returns.

• Consider Uses of **etype**. The calculation of the set of Uses is described in Section 7.3.3. If **etype** is MakeMarinara, then **explain** is recursively invoked with etypes of MakeSpaghettiMarinara and MakeChickenMarinara. The call to **redundant** eliminates uses which have already been considered by alternative paths. Examples follow.

• Explain **etype** by considering its abstractions. The node becomes an *alternative-for* its abstractions. Suppose the current invocation has created MakePastaDish(N4). This step calls

```
explain( PrepareMeal, {(agent Joe) (time (-∞ 5 6 +∞))},
                {MakePastaDish}, False, MakePastaDish )
```

which returns N6.

Not all abstractions lead to End; some are pruned, and don't appear in the final graph. Consider the invocation which created MakeMarinara(N1). It calls **explain** for MakeSauce. The only Use for MakeSauce, however, is (MakeSauce, step2, MakePastaDish); but that use is eliminated by the **redundant** test. Therefore no node of type MakeSauce appears in the final graph.

• Try to explain **etype** by considering specializations of **etype**. This step is not performed if **etype** was reached by abstracting some other type, rather than as a Use. This always occurs in the example. But suppose **explain** were initially invoked with **etype** equal to MakeSauce. Then the specialization MakeMarinara is considered. In the recursive invocation of **explain**, the Uses of MakeMarinara are examined. The use (MakeMarinara, step2, MakeSpaghettiMarinara) is eliminated by **redundant**, because it specializes (MakeSauce, step2, MakePastaDish). The use (MakeMarinara, step5, MakeChickenMarinara), however, *does* lead to a path to End.

• If any path to End was found, the **status** of the node is set to True, and otherwise to False. Finally the name of the node and its status are returned.

7.7.2. Match

The following diagram shows two e-graphs, the first built from an observation of MakeMarinara, and the second from an observation of MakeNoodles. **Match** is initially invoked on the End nodes of the two graphs:

match(N7, N11)

and returns N12, the End node of the combined graph. The following section steps through the operation of **Match**.

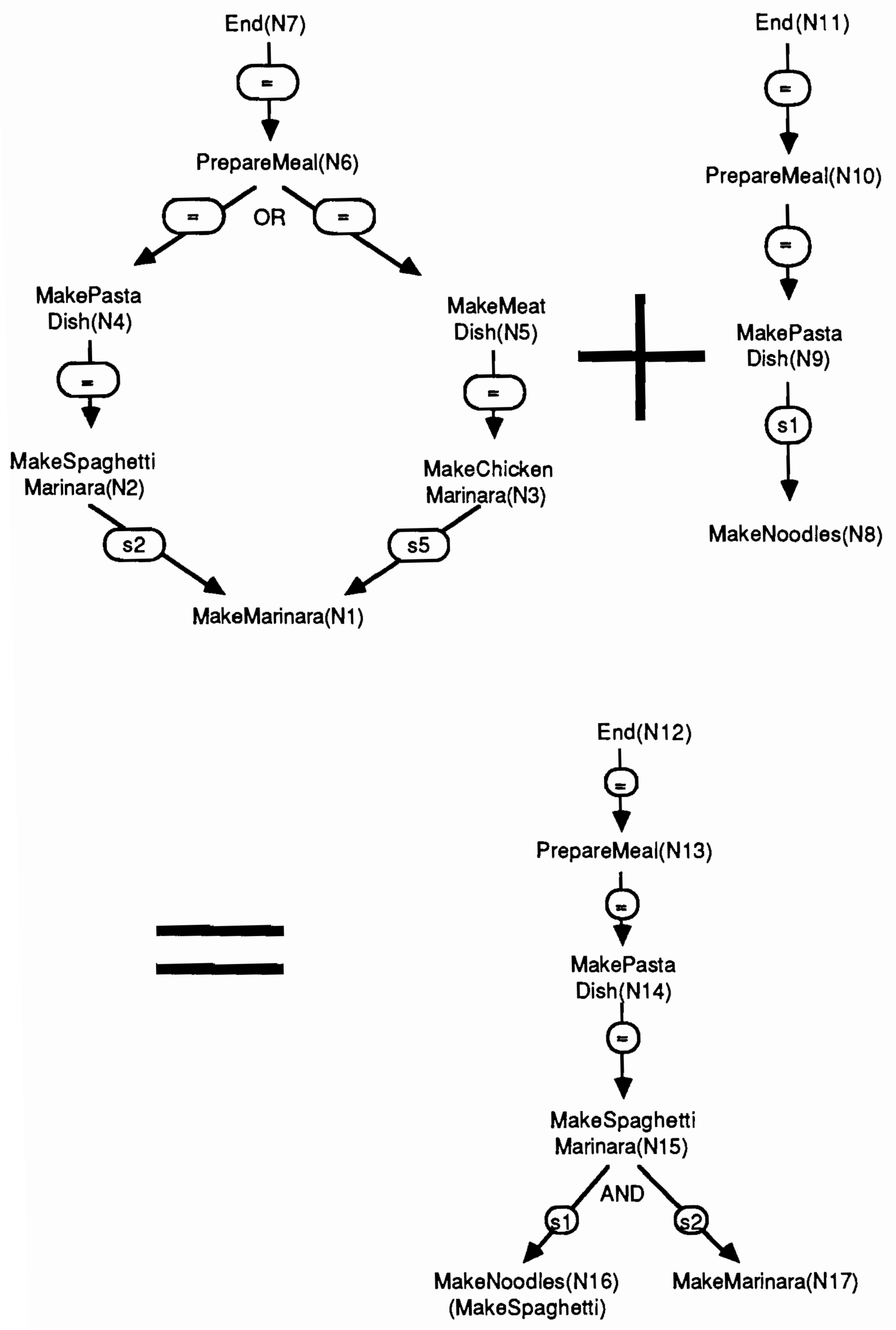


figure 7.4: E-Graphs for MakeMarinara, MakeNoodles, and their Match

• If the objects to be matched are rigid designators, they must be lexically identical.

• If the objects are fuzzy temporal constraints, then take their intersection. Suppose **n1** is (–∞ 5 6 +∞), and **n2** is (–∞ 8 7 +∞). Then **match** returns (–∞ 5 7 +∞).

• Check whether **n1** and **n2** have already been matched, and if so, reuse that value. Suppose that the first e-graph in the diagram above were matched against an e-graph of identical shape; for example, there were two observations of MakeMarinara, and they *may* have been identical. During the match down the left hand side of the graphs, through N4 and N2, MakeMarinara(N1) would match against the MakeMarinara node in the second graph (say, N1'), resulting in some final node, say N1". Then the right-hand side of the graphs would match, through N5 and N3. N1 would match against N1' a second time, and the value N1" would be used again. This would retain the shape of the digraph, and prevent it from being multiplied out into a tree.

• Add a new node to the graph, **n3**, to represent the result of the match, which is of the greatest lower bound type of **n1** and **n2**. Matching MakeSpaghettiMarinara(N2) against MakePastaDish(N9) results in MakeSpaghettiMarinara(N15). The match fails if the types are not compatible. Thus MakeMeatDish(N5) fails to match against MakePastaDish(N9)

• Match the roles of **n1** and **n2**. If a role value is defined for one node but not the other, match the role against itself in order to simply copy the structure into the resulting graph. This what happens when N2 matches N9, yielding N15. The new node gets both the **step2** role value from N2 (a copy of N1, which is N17) and the **step1** role value from N9 (a copy of N8, which is N16). If N2 had the role **step1** defined, that value would have had to match against N8.

Suppose both End nodes, N7 and N11, have the **agent** roles defined. This step checks that the agents are the same.

• Check and propagate the constraints on **n3**. New constraint violations may be detected at this point because more of the roles of the nodes are filled in. Violations of temporal orderings between steps are detected at this point. The transcripts contain an example where an e-graph based on Boil fails to match against one based on MakeNoodles, because the Boil occurred *before* the MakeNoodles.

• Try matching every alternative for **n1** against every alternative for **n2**. The successful matches are alternatives for **n3**. If one of the nodes has some alternatives, but the other does not, then match the alternatives for the former against the latter directly.

This occurs in the example above. MakePastaDish(N4) matches against MakePastaDish(N9). N4 has the alternative N2, but N9 has none. Therefore MakeSpaghetti-Marinara(N2) matches against MakePastaDish(N9) as well.

If there were some alternatives but all matches failed, then **n3** fails as well.

- Return **n3** and whether or not the match succeeded.

7.7.3. Group

The various versions of **group-observations** are considerably simpler than the previous algorithms.

The **Non-Dichronic Version** maintains a set of all possible groupings of the observations. It functions as follows:

- Observe an event of **etype** satisfying **edescr**.
- Generate **obs-graph** by **explain**.
- *Conjoin* **obs-graph** with each hypothesis consisting of a set of e-graphs. Thus **obs-graph** is considered to be *unrelated* to the previous observations.
- Try to **match** **obs-graph** with each e-graph in each hypothesis. Thus **obs-graph** is considered to be *related* to a previous observation.
- The current (mc-entailed) conclusion corresponds to the *disjunction* of all hypotheses of minimum size.

The **Incremental Minimization Version** works as follows:

- Observe an event of **etype** satisfying **edescr**.
- Generate **obs-graph** by **explain**.
- Try to **match** **obs-graph** with each e-graph in each hypothesis. Thus **obs-graph** is considered to be *related* to a previous observation. Each successful match leads to a new hypothesis.

- If **obs-graph** *cannot* be matched with previous e-graphs, then *conjoin* **obs-graph** with each hypothesis.

- The current (imc-entailed) conclusion corresponds to the *disjunction* of all hypotheses, all of which are of same size.

Finally, the **Sticky Version** works as follows:

- Observe an event of **etype** satisfying **edescr**.

- Generate **obs-graph** by **explain**.

- Try to **match obs-graph** with each e-graph in the (single) hypothesis, in chronological order, from most to least recent. As soon as a match succeeds, update the hypothesis, removing the old matching e-graph, and adding the result of the match as the most recent e-graph.

- If **obs-graph** *cannot* be matched with any previous e-graphs, then *conjoin* **obs-graph** with the hypothesis, as the most recent e-graph.

- The current conclusion corresponds to the single hypotheses, a conjunction of e-graphs.

7.8. Completeness & Correctness

The necessarily limited nature of the constraint checking performed by the algorithms, together with the open-ended nature of the plan recognition problem, makes it impossible to guarantee that the algorithms will arrive at the *strongest* conclusions justified by the theory of mc-entailment. The algorithms are, however, *correct:* the interpretation of their outputs holds in all minimum-covering models. This most easily seen by noting that the algorithms implement the proof rules discussed in Chapters 3 and 4.

The key to the correctness of **explain** is the fact that the algorithm searches from the input event type to every possible use of events of that type, as specified by Theorem 3.11. (A precise statement and proof of this fact, which demonstrates that the set Uses and the **redundant** check are properly formulated, appears in Appendix F.) Each path leads to a different disjunctive conclusion, which may be collapsed with others via the **alternative-for** arcs.

The **match** algorithm simply propagates an equality assertion between terms. If two terms n1 and n2 are equal, then the value of a function applied to the first is the same as the value of that function applied to the second. If n1 is equal to one of a set of alternatives, and n2 is equal to one of another set of alternatives, then certainly at least one alternative from the first set is equal to some alternative from the second. The Disjointedness Assumptions justify concluding that n1 and n2 are not equal if they are not of compatible types.

7.9. Complexity

One straightforward measure of the complexity of the algorithms is the size of largest data structure possibly produced. A more detailed analysis would measure time complexity as well; but since the algorithms visit each node a fixed number of times, we can take space complexity as a rough measure of time complexity as well. (A possible exception is the test to handle the multi-path problem in the multiple-inheritance version of explain-observation. The cost of the best algorithm to perform this test is not obvious.)

The *size* of an e-graph, |G|, is the number of nodes in G. The expression "Algorithm X is of O(f(n))" in the following sections means that the largest data structure constructed by X on input n is no larger than c*f(n), for some fixed constant c. The *actual* worst-case complexity of X may, of course, be smaller than f(n). The expression "Algorithm X is no better than O(f(n))" means that there is no g such that X is of O(g(n)) and g is better than f by more than a constant factor; that is, it is not the case that

$$\forall k > 0 \,.\, \exists m \,.\, \forall n \,.\, n > m \supset g(n) < k{*}f(n)$$

7.9.1. Explain

Suppose that the events described by H contain no parameters. Then explain-observation is of $O(|H_E|)$, since the worst case would lead to an exhaustive search of all of H. But suppose explain-observation did not merge search paths at abstraction points. Then the algorithm would be no better than $O(2^{|H_E|})$, because an event can have several different components of the same type. The figure below is a case where such combinatorial blowup could occur.

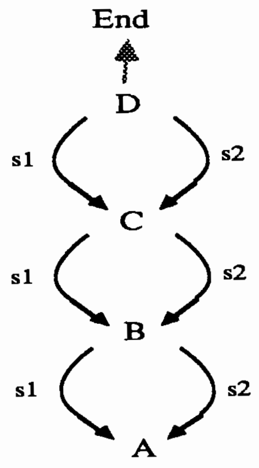


figure 7.5: Combinatorially Explosive Hierarchy

This example shows the importance of maintaining disjunctions within each e-graph, using the alternative-for relation, rather than creating separate databases for each alternative.

If parameters (such as agent, time, etc.) do appear, the search space will be larger. During the construction of the e-graph, however, very few of these roles will have known values. Thus the first step in explain will very frequently find a similar node and cut off search.

7.9.2. Match

Calling match-graphs(G_1,G_2) frequently returns a graph which is smaller than either G_1 or G_2. Unfortunately, sometimes the resulting graph can be larger. This occurs when two nodes are matched which have several alternatives, all of which are mutually compatible. Therefore the worst case complexity of match-graphs(G_1,G_2) is $O(|G_1|*|G_2|)$.

There cannot be more than $|G_1|*|G_2| + |G_1| + |G_2|$ nodes in the result, because every pair of nodes always matches to the same node. The last two addends enter in because a node may also match against itself. Without the use of the cache of previous results, performance could be much worse. The resulting graph would always be a tree, and the algorithm would be no better than $O(2^{|G_1|*|G_2|})$.

7.9.3. Group

Some of the most intimidating complexity results arise from the group-observations algorithm. The non-dichronic algorithm finds all consistent *partitions* of the set of observations. In the worst case, all partitions are possible – any subset of the observed events could be part of the same End event. Then Hypoths will contain B(n) hypotheses, where n is the number of observations. There is no simple closed form for B(n), but it is easy to show that

$$B(n) < \frac{n^n}{n!} \approx \frac{n^n}{\left(\frac{n}{e}\right)^n \sqrt{2\pi n}} \approx \frac{e^n}{\sqrt{2\pi}}$$

so that group-observations could contain at least 2^n hypotheses.

Together with the worst-case estimates for explain and match, Hypoths could total more than $O(2^n|H_E|^n)$ nodes. Of course, if all the observations could be part of the same End event, then minimum-hypos will return a single hypothesis consisting of a single e-graph which incorporates all the observations.

The basic dichronic algorithm does much better in this case. If all the observed events could be part of the same hypothesis, then Hypoths will be of size 1. This worst-case size of this hypothesis is $O(|H_E|^n)$. This may not seem like much of an improvement, but remember that the $|H_E|^n$ is a very unlikely upper bound – it would only hold if all the observations were not telling us *anything* definite! – while the 2^n factor in the non-dichronic algorithm *would be accurate* in the common case of all the observations being possibly related.

Once several End events are required to account for all the observations, however, the dichronic algorithm could fall prey to the same sort of combinatorial explosion. Thus a truly practical system may need to resort to the sticky dichronic algorithm. Because it develops a single hypothesis, the sticky algorithm has an absolute upper bound of $O(|H_E|^n)$ (and will usually do much better). Sometimes the sticky algorithm will fail to find the correct – or any – interpretation of the data. An endless number of more or less general and more or less efficient algorithms lie between the non-dichronic and sticky versions.

7.10. Transcripts

The **explain-observation** and **match-graphs** algorithms have been implemented in Common Lisp, and appear practical in small domains. The recognition problems discussed in this thesis all run in a few seconds. Appendix E contains transcripts of the programs in operation.

Chapter 8
Conclusions

8.1 The Three Levels

[Marr 82] argues that research in A.I. should analyze each problem at three distinct levels. The *computational theory* states the goal of the computation, why it is appropriate, and provides an abstract mapping from the input information to the output. The level of *representation and algorithms* provides a particular representation for the input and output, and an algorithm for the transformation. The *hardware implementation* shows how the algorithm can be realized physically. Quite often the computational level is overlooked; the result, as Marr demonstrates with many examples from the study of vision, are algorithms which do not actually solve the intended problem, and are based on incorrect assumptions. Neglecting the algorithmic theory is dangerous in another way: one can develop elegant formal theories which would require infinite time and resources to implement.

Our framework includes model, proof, and algorithmic theories. How do these relate to the three levels? The model theory provides a mapping from models of the input to models of the conclusions; thus it is a pure information to information mapping, a computational level theory of the most abstract kind. The proof theory is a computational theory of a more concrete kind. It provides a non-deterministic procedure (namely, the generation of the non-monotonic assumptions, followed by the application of deductive rules) that transforms a first-order representation of the input to that of the output. Thus the proof theory stands between Marr's computational and algorithmic levels. Finally our algorithmic theory fills the algorithmic level, by providing a deterministic procedure which may be directly programmed on a computer. We have nothing to say about the hardware level, how anything like our theory could be realized in a brain. Practically all work that suggests how high-level inference could be performed by neurons is fueled mostly by (often ingenious) speculation; but see [Feldman 81] for some proposals in this direction.

Having completed this thesis, there is no doubt in my mind that the most difficult task was to define the computational theory at the most abstract level. Once the rather amorphous problem of "plan recognition" was tied down, work could proceed in earnest. As the literature review in Chapter 1 revealed, similar proposals and heuristics for plan recognition problems have been hashed over for years, since the early work of

Schmidt and Genesereth. The minimal model construct we've developed suggests what some of these heuristics are heuristics *for*. The lack of a clear problem statement is no doubt one of the reasons for the stagnation in work in story understanding in the past decade.

8.2 Applicability

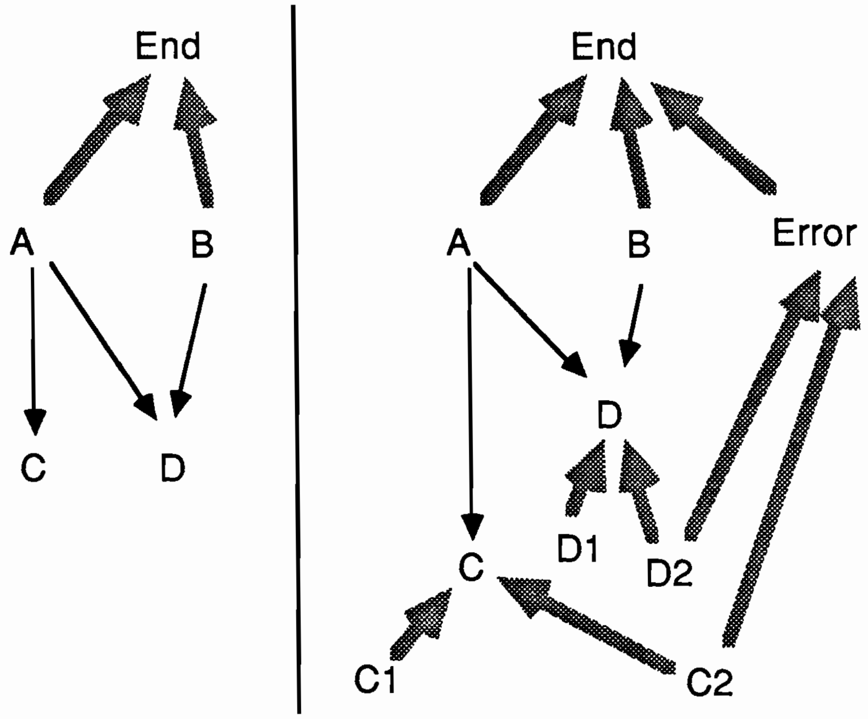
Our framework makes explicit the assumptions of Exhaustiveness, Disjointedness, and Component/Use Completeness which underlie our theory of plan recognition, as well as most previous work. These assumptions simplify the problem, and restrict its applicability. Do these assumptions limit our work to toy micro-worlds?[1]

Every day new kinds of events occur, and yet they do not baffle us. An intelligent agent cannot rely only on a recognition system; it must contain a learning component as well. Earlier we suggested that it might be appropriate to invoke the learning module when recognition failed. The ability of our framework to handle levels of abstraction provides a crucial feature needed for learning. For example, we might be able to conclude that the agent is cooking some kind of pasta dish, even if we can rule out any of the *particular* pasta dishes in our library. It may be possible to modify the minimal model construction so that the Exhaustiveness assumption is weakened, and thus allow the recognition and learning modules to be integrated.

None the less, much of what a plan recognition system must handle is routine. A more serious problem is that of recognizing *erroneous* plans. Our framework assumes all plans are internally consistent, and that all acts are purposeful. Yet real people make frequently make planning errors and change their minds in midcourse.

One class of errors can be handled without significant change to our framework. Suppose *every* basic event type is given two new specializations, one of which also specializes a new type called Error. The diagram below illustrates how a hierarchy would be transformed.

[1] [Dreyfus 85] argues that this criticism applies to work in A.I. in general.



Then each observed event could either be an End event on its own by being an Error, or it could be part of an End event in a correct plan from the original hierarchy. Minimizing the number of End events prefers interpretations in which Errors do not occur; but observations sequences which *must* be part of erroneous plans can still be recognized.

8.3 Generality & Extensibility

Despite the limitations outlined above, the generality of our framework must be stressed. We do *not* assume that there is a single plan underway, which can be uniquely identified from the first input. We do *not* assume that the sequence of observations is complete.[2] Finally, we do *not* assume that all the steps in a plan are linearly ordered. Indeed, we know of no other implemented plan recognition system which handles arbitrary temporal relations between steps.

The logical basis for the system makes it clear what conclusions should be reached on the basis of quantified and disjunctive information. Section 2.7 showed

[2]Completeness of the observations is a nice property to exploit when it is available. [Segre 86] has built a plan recognition system for use in programming a robot arm. The operator guides the arm manually, and the system recognizes what plans are being performed. No kind of non-deductive inference is needed: because every step of the plan is input, the fact that the plan occurs logically follows.

how plans involving conditional actions could be represented by event hierarchies. Previous plan recognition systems did not handle any of these situations.

The generality of the system suggests that it may be extended in various ways in a straightforward manner. An important one for plan recognition is the ability to deal with plans containing iteration and recursion. For example, a plan to pick up all the blue blocks might be encoded as follows:

$$\forall x\ .\ \text{pickup-blues}(x) \supset \forall y.\ \text{block}(y) \wedge \text{blue}(y) \supset \text{pickup}(s(y,x)) \wedge \text{object}(s(y,x))=y$$

That is, if an instance of picking up blue blocks occurs, then for every block that is blue, an instance of pickup occurs with that block as its object. We have not yet investigated whether this extension to the form of the event hierarchy requires modification in the model theory. Another area for future work is to extend our notion of an event hierarchy to include hierarchies with exceptions [Touretzky 84].

Finally we need to consider principles other than Occam's Razor for determining the class of preferred interpretations. Rather than ordering models during the minimization process by the number of End events, one may wish to include other factors, such as qualitative likelihoods of various event types. It remains to be seen how far the general framework can be pushed, or if other, more quantitative measures belong in a separate theory, which applies to the *conclusions* of our system.

8.4 Two Unresolved Issues

After reviewing the strong and weak points of our work, a couple of unresolved issues remain. The first is the question of scale-up. As discussed in Chapter 7, the worst-case behavior of our recognition algorithms can be bad, although proper structuring of the hierarchy greatly reduces search. How much of a problem will this be in more realistically-sized knowledge bases? Experience in creating and examining the structure of larger domains is necessary.

A more philosophical problem is the whole issue of what serves as primitive input to the recognition system. Throughout this thesis we've assumed that arbitrary high-level descriptions of events are simply presented to the recognizer. This assumption is reasonable in many domains, such as understanding written stories, or observing the words typed by a computer operator at a terminal. But a real plan recognizer – a person – doesn't always get his or her input in this way. How are visual impressions of simple bodily motions – John is moving his hands in such and such a manner – translate into the impression that John is rolling out dough to make pasta? There is a great deal of work in low-level perception, and a great deal in high level recognition – including this thesis. The semantic gap between the output of the low-level processes and the high-level inference engines remains wide, and few have ventured to cross it.

References

Allen, James (1983) Recognizing Intentions From Natural Language Utterances, in *Computational Models of Discourse*, eds. Michael Brady & Robert Berwick, The MIT Press, Cambridge.

Allen, James (1983b) Maintaining Knowledge About Temporal Intervals, *Communications of the ACM*, no. 26, pp. 832-843.

Allen, James (1984) Towards a General Theory of Action and Time, *Artificial Intelligence*, vol. 23, no. 2, pp. 832-843.

Allen, James & Alan Frisch (1982) What's In a Semantic Network?, *Proceedings of the ACL-82*, Toronto.

Allen, James & Johannes Koomen (1983) Planning Using a Temporal World Model, *Proceedings of IJCAI-83*, Karlsruhe, West Germany.

Allen, J.F. & C.R. Perrault (1980) Analyzing Intention in Dialogues, *Artificial Intelligence*, vol. 15, no. 3, pp. 143-178.

Barwise, Jon (1977) editor, *The Handbook of Mathematical Logic.*

Blenko, Tom (1985) Suggestions and Offers, in *Report on Commonsense Summer*, ed. Jerry Hobbs, Center for the Study of Language and Information, Stanford University.

Bossu, G. & P. Siegel (1985) Saturation, Non-Monotonic Reasoning, and the Closed-World Assumption, *Artificial Intelligence,* vol. 25, no. 1, pp. 13-63.

Brachman, Ronald J. (1985) On the Epistemological Status of Semantic Nets, in *Readings in Knowledge Representation,* eds. R.J. Brachman & H.J. Levesque, Morgan Kaufman, Los Altos, CA.

Brachman, Ronald J., Richard E. Fikes & Hector J. Levesque (1985) KRYPTON, A Functional Approach to Knowledge Representation, in *Readings in Knowledge Representation,* eds. R.J. Brachman & H.J. Levesque, Morgan Kaufman, Los Altos, CA.

Bruce, B.C. (1981) Plans and Social Action, in *Theoretical Issues in Reading Comprehension*, eds. R. Spiro, B. Bruce, & W. Brewer, Lawrence Erlbaum, Hillsdale, New Jersey.

Carberry, Sandra (1983) Tracking Goals in an Information Seeking Environment, *Proceedings of 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.

Charniak, Eugene & Drew McDermott (1985) *Introduction to Artificial Intelligence*, Addison Wesley, Reading, MA.

Clark, K.L. (1978) Negation as Failure, in *Logic and Databases*, ed. J. Minker, Plenum Press, New York.

Cohen, P.R. (1978) On Knowing What to Say: Planning Speech Acts, TR 118, Department of Computer Science, University of Toronto, Ontario.

Cohen, Phillip (1984) Referring as Requesting, *Proceedings of COLING-84*, pp. 207, Stanford University.

Cohen, P. & Levesque, H. (1980) Speech Acts and the Recognition of Shared Plans, *Proceedings of the CSCSI-80*, Victoria, British Columbia, pp. 263-271.

Cohen, P. & Levesque, H. (1987) Rational Interaction as the Basis for Communication, *Proceedings of the Symposium on Intentions and Plans in Communication and Discourse*, Center for the Study of Language & Information, Stanford, CA.

Cohen, P., R. 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*, vol. 13, pp. 73-80.

Dreyfus, Hubert L. (1985) From Micro-Worlds to Knowledge Representation: A.I. at an Impasse, in *Readings in Knowledge Representation,* eds. R.J. Brachman & H.J. Levesque, Morgan Kaufman, Los Altos, CA.

Dwyer, M.G. (1982) In Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension, TR 219, Department of Computer Science, Yale University.

Etherington, David (1986) Reasoning with Incomplete Information: Investigations of Non-Monotonic Reasoning, Technical Report 86-14, Department of Computer Science, University of British Columbia, Vancouver, Canada.

Etherington, David (1987) personal communication.

Feldman, Jerome A. (1981) Memory and Change in Connection Networks, TR 96, Department of Computer Science, University of Rochester, Rochester, NY.

Fillmore, Charles (1968) The Case for Case, in *Universals in Linguistic Theory,* Holt, New York.

Genesereth, Michael (1979) The Role of Plans in Automated Consulting, *Proceedings of IJCAI-79*, pp. 119.

Genesereth, Michael (1983) An Overview of Meta-Level Architecture, *Proceedings of AAAI-83*, Washington, D.C.

Goldman, Alvin (1970) *A Theory of Human Action,* Princeton University Press, Princeton.

Green, C. (1969) An Application of Theorem Proving to Problem Solving, *Proceedings of IJCAI-69*, 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-Verlang.

Hacking, Ian (1983) *Representing and Intervening: Introductory Topics in the Philosophy of Natural Science,* Cambridge University Press.

Hayes, P.J. (1985) The Logic of Frames, in *Readings in Knowledge Representation,* eds. R.J. Brachman & H.J. Levesque, Morgan Kaufman, Los Altos, CA.

Hintikka, Jaakko (1962) *Knowledge and Belief,* Cornell University Press, Ithaca, NY.

Huff, Karen & Victor Lesser (1982) KNOWLEDGE-BASED COMMAND UNDERSTANDING: An Example for the Software Development Environment, TR 82-6, Computer and Information Sciences, University of Massachusetts at Amherst.

Kunz, John C. (1983) Analysis of Physiological Behavior Using a Causal Model Based on First Principles, *Proceedings of AAAI-83*, Washington, D.C., pp. 225-228.

Kautz, Henry (1985) Towards a Theory of Plan Recognition, TR 162, Department of Computer Science, University of Rochester.

Kautz, Henry & James Allen (1986) Generalized Plan Recognition, *Proceedings of AAAI-86*, Philadelphia.

Kautz, Henry (1986b) The Logic of Persistence, *Proceedings of AAAI-86*, Philadelphia.

Kautz, Henry (1986c) Poèmes humoristiques sur l'AI, *Canadian Artificial Intelligence,* no. 9, Sept.

Kautz, Henry (1987) *A Formal Theory of Plan Recognition*, PhD Thesis, Department of Computer Science, University of Rochester.

Kyburg, 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.

Marr, David (1982) *Vision,* W.H. Freeman, New York.

McAllester, David (1980) An Outlook on Truth Maintenance, AI Memo 551, M.I.T..

McCarthy, John (1980) Circumscription -- A Form of Non-Monotonic Reasoning, *Artificial Intelligence*, vol. 13, pp. 27-39.

McCarthy, John (1984) Applications of Circumscription to Formalizing Common Sense Knowledge, *Proceedings of the AAAI Workshop on Non-Monotonic Reasoning.*

McCarthy, J. & P. Hayes (1969) Some Philosophical Questions from the Standpoint of Artificial Intelligence, *Machine Intelligence 4*, Edinburg University Press, Edinburg, UK.

McDermott, Drew (1981) A Temporal Logic for Reasoning About Processes and Plans, Research Report #196, Department of Computer Science, Yale University.

Minker, J. & Perlis, D. (1985) Circumscription: Finitary Completeness Results, Technical Report, Computer Science Department, University of Maryland.

Minsky, Marvin (1975) A Framework for Representing Knowledge, in *The Psychology of Computer Vision*, McGraw-Hill, New York.

Nilsson, N.J. (1980) *Principles of Artificial Intelligence*, Tioga Press, Palo Alto.

Patil, Ramesh, Peter Szolovits, & William Schwartz (1982) Modeling Knowledge of the Patient in Acid-Base and Electrolyte Disorders, in *Artificial Intelligence in Medicine,* ed. Peter Szolovits, AAAS Select Symposium 51, Westview Press, Boulder, Colorado.

Peirce, Charles S. (1958) *Collected Papers of Charles Sanders Peirce,* Cambridge, Mass.

Pereira, F. & D. Warren (1982) Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Transition Networks, *Artificial Intelligence*, vol 13, pp 231-278.

Perlis, D. (1980) *Truth, Syntax, and Reason*, PhD Thesis, Computer Science Department, University of Rochester.

Pierrehumber, Janet & Julia Hirschberg (1987) The Meaning of Intonational Contours in the Interpretation of Discourse, in *Readings in Knowledge Representation,* eds. R.J. Brachman & H.J. Levesque, Morgan Kaufman, Los Altos, CA.

Pollack, Martha (1986) A Model of Plan Inference that Distinguishes Between the Beliefs of Actors and Observers, *Proceedings of the ACL-86*, New York.

Pople, Harry (1973) On the Mechanization of Abductive Logic, *Proceedings of IJCAI-73*, Stanford University, CA.

Pople, Harry (1977) The Formation of Compound Hypotheses in Diagnostic Problem Solving: an Exercise in Synthetic Reasoning, *Proceedings of IJCAI-77*, Cambridge, MA.

Pople, Harry (1982) Heuristic Methods for Imposing Structure on Ill-Structured Problems: The Structuring of Medical Diagnostics, in *Artificial Intelligence in Medicine,* ed. Peter Szolovits, AAAS Select Symposium 51, Westview Press, Boulder, Colorado.

Reggia, J., D.S. Nau, & P.Y. Wang (1983) Diagnostic Expert Systems Based on a Set Covering Model, *International Journal of Man-Machine Studies*, vol. 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.

Reiter, Ray (1982) Circumscription Implies Predicate Completion (Sometimes), *Proceedings of AAAI- 82*, William Kaufman, Inc.

Rich, Charles (1981) Inspection Methods in Programming, AI-TR-604, Laboratory for Artificial Intelligence, M.I.T..

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.

Schafer, Glenn (1976) *A Mathematical Theory of Evidence,* Princeton University Press, Princeton, NJ.

Schank, R. (1975) *Conceptual Information Processing*, American Elsevier, New York.

Schank, Roger (1983) The Current State of A.I.: One Man's Opinion, *The A.I. Magazine,* vol. 4, no. 1.

Schmidt, C.F., N.S. Sridharan, & J.L. Goodson (1978) The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence, *Artificial Intelligence*, vol.11, pp. 45-83.

Schmolze, James G. (1986) Physics for Robots, *Proceedings of AAAI-86*, Philadelphia.

Segre, Alberto (1976) Talk given at University of Rochester.

•

Sidner, Candace (1983) Focusing in the Comprehension of Definite Anaphora, in *Computational Models of Discourse*, eds. M. Brady & R. Berwick, M.I.T. Press, Cambridge.

Sidner, Candace (1985) Plan Parsing for Intended Response Recognition in Discourse, *Computational Intelligence*, vol.1, no. 1, pp. 1.

Sidner, Candace & David Israel (1981) Recognizing Intended Meaning and Speakers' Plans, *Proceedings of IJCAI-81*, University of British Columbia, Vancouver, BC, pp. 203-298.

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.

Stefik, Mark (1980) Planning with Constraints, Report STAN-CS-80-784, Department of Computer Science, Stanford University.

Tenenburg, Josh (1986) Reasoning Using Exclusion: An Extension of Clausal Form, TR 147, Department of Computer Science, University of Rochester.

Vilain, Marc & Henry Kautz (1986) Constraint Propagation Algorithms for Temporal Reasoning, *Proceedings of AAAI-86*, Philadelphia.

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.

Appendix A
Chapter 3 Proofs

Theorem 3.1 (Exhaustiveness)

Suppose $\{E_1, E_2, \ldots, E_n\}$ are all the predicates directly abstracted by E_0 in H_A. Then the statement:

$$\forall x . E_0(x) \supset (E_1(x) \vee E_2(x) \vee \ldots \vee E_n(x))$$

is true in all models of H_A which are closed under specialization. The statement is also true in all A-closed models of H.

Proof

Let M_1 be a model of H_A in which the statement does not hold. We prove that M_1 cannot be closed under specialization.

For the statement to be false in M_1, there must be some :C such that

$$E_0(x) \wedge \neg E_1(x) \wedge \ldots \wedge \neg E_n(x)$$

is true in $M_1\{x/{:}C\}$. Define M_2 by

$$\text{Domain}(M_2) = \text{Domain}(M_1)$$
$$M_2[Z] = M_1[Z] \text{ for } Z \neq E_0$$
$$M_2[E_0] = M_1[E_0] - \{{:}C\}$$

That is, M_2 is the same as M_1, except that $:C \notin M_2[E_0]$.

We claim that M_2 is a model of H_A. Every axiom that does not contain E_0 is true in M_1, and therefore receives the same valuation in M_2. So consider axioms that contain E_0. Axioms of the form:

$$\forall x . E_0(x) \supset E_i(x) \qquad i \neq 0$$

are false only if there is a :D such that

$$:D \in M_2[E_0] \wedge :D \notin M_2[E_i]$$

But we know that

$:D \in M_2[E_0] \Rightarrow$

$:D \in M_1[E_0] \Rightarrow$

$:D \in M_1[E_i] \Rightarrow$

$:D \in M_2[E_i]$

which is a contradiction. Therefore there can be no such :D, and the axiom holds in M_2.

The only other axioms containing E_0 are of the form

$\forall x \,.\, E_i(x) \supset E_0(x) \qquad\qquad 1 \leq i \leq n$

Again, this fails in M_2 if there is a :D such that

$:D \in M_2[E_i] \wedge :D \notin M_2[E_0]$

If $:D \neq :C$, then

$:D \in M_2[E_i] \Rightarrow$

$:D \in M_1[E_i] \Rightarrow$

$:D \in M_1[E_0] \Rightarrow$

$:D \in M_2[E_0]$

which is a contradiction. Finally, if $:D = :C$, then, since $:C \notin M_1[E_i]$,

$:D \notin M_1[E_i] \Rightarrow$

$:D \notin M_2[E_i]$

which also is a contradiction. Therefore there can be no such :D. So M_2 is a model of H_A.

Furthermore, M_1 and M_2 agree on all terms and predicates outside of H_E–H_{EB}. (E_0 cannot be a basic event type.) M_2 defeats M_1's candidacy for minimality in H_E–H_{EB}. Therefore the statement in Theorem 3.1 must hold in all models closed under specialization. Since A-closed models are a subset of those closed under specialization, the statement also holds in all A-closed models.

Q.E.D.

Theorem 3.2

Let EXA be the set of all statements which instantiate Theorem 3.1 for a particular H. If M_1 is a model of $H_A \cup$ EXA such that $:C \in M_1[E_0]$, then there is a basic event type $E_b \in H_{EB}$ such that $:C \in M_1[E_b]$ and E_0 abstracts* E_b.

Proof

By induction. Define a partial ordering over H_E by $E_j < E_k$ iff E_k abstracts E_j. Suppose $E_0 \in H_{EB}$. Then E_0 abstracts* E_0. Otherwise, suppose the lemma holds for all $E_i < E_0$. Since

$$\forall x . E_0(x) \supset (E_1(x) \vee E_2(x) \vee \ldots \vee E_n(x))$$

and $:C \in M_1[E_0]$, it must the case that

$$:C \in M_1[E_1] \vee \ldots \vee :C \in M_1[E_n]$$

WLG, suppose $:C \in M_1[E_1]$. Then there is an E_b such that E_1 abstracts* E_b and $:C \in M_1[E_b]$. Since E_0 abstracts E_1, it also abstracts* E_b.

Q.E.D.

Theorem 3.3

Every event in a model closed under specialization is of at least one basic type. That is, if M_1 is a model of H_A closed under specialization such that $:C \in M_1[E_0]$, then there is a basic event type $E_b \in H_{EB}$ such that $:C \in M_1[E_b]$ and E_0 abstracts* E_b.

Proof

By Theorem 3.1, $H_A \cup$ EXA holds in M_1, and by Theorem 3.2 there is such an E_b.

Q.E.D.

Theorem 3.4

M_1 is a model of H_A closed under specialization if and only if M_1 is a model of $H_A \cup EXA$.

Proof

The "only if" half follows from Theorem 3.1. We prove that if M_1 is a model of $H_A \cup EXA$, then M is closed under specialization.

Suppose not. Then there is an M_2 which defeats M_1. M_2 and M_1 agree on H_{EB}, but there is (at least one) $E_0 \in H_E - H_{EB}$ and event :C such that

$$:C \in M_1[E_0] \wedge :C \notin M_2[E_0]$$

By Theorem 3.2, there is an $E_b \in H_{EB}$ such that

$$:C \in M_1[E_b]$$

Since M_1 and M_2 agree on H_{EB},

$$:C \in M_2[E_b]$$

But then because E_0 abstracts E_b

$$:C \in M_2[E_0]$$

which is a contradiction. Therefore there can be no such M_2. Thus M_1 is closed under specialization.

Q.E.D.

Theorem 3.5 (Disjointedness)

If event predicates E_1 and E_2 are not compatible, then the statement:

$$\forall x \,.\, \neg E_1(x) \vee \neg E_2(x)$$

is true in all models of H_A which are closed under abstraction. The statement is also true in all A-closed models of H.

Proof

Suppose M_1 is a model of H_A closed under specialization in which the statement is false. We prove that M_1 is not closed under abstraction.

So let :C be an event such that

$$E_1(x) \wedge E_2(x)$$

is true in M1{x/:C}, where E_1 and E_2 are not compatible. Using Theorem 3.3, let E_b be a basic event type abstracted by E_1 such that $:C \in M_1[E_b]$. Define M_2 as follows.

$$\mathrm{Domain}(M_2) = \mathrm{Domain}(M_1)$$
$$M_2[Z] = M_1[Z] \text{ for } Z \notin H_E$$
$$M_2[E_i] = \begin{array}{ll} M_1[E_i] & \text{if } E_i \text{ abstracts* } E_b \\ M_1[E_i]-\{:C\} & \text{otherwise} \end{array}$$

In particular, note that M_1[AnyEvent]=M_2[AnyEvent], since AnyEvent certainly abstracts E_b. We claim that M_2 is a model of H_A closed under specialization.

(Proof that M_2 is a model of H_A)

Suppose M_2 is not a model of H_A; in particular, suppose the axiom

$$\forall x \,.\, E_j(x) \supset E_i(x)$$

is false in M_2. Since it is true in M_1, and M_2 differs from M_1 only in the absence of :C from the extension of some event predicates, it must be the case that

$$:C \in M_2[E_j] \wedge :C \notin M_2[E_i]$$

while

$$:C \in M_1[E_j] \wedge :C \in M_1[E_i]$$

By the definition of M_2, it must be the case that E_j abstracts* E_b. Since E_i abstracts E_j, then E_i abstracts* E_b as well. But then M_1 and M_2 would have to agree on E_i; that is,

$$:C \in M_2[E_i]$$

which is a contradiction. So M_2 must be a model of H_A after all.

(Proof that M2 is closed under specialization)

By Theorem 3.4, M_2 is closed under specialization if it is a model of EXA. So suppose it is not a model of EXA; in particular, suppose

$$\forall x \,.\, Ej_0(x) \supset (Ej_1(x) \vee Ej_2(x) \vee \ldots \vee Ej_n(x))$$

is false. Then it must be the case that

$$:C \in M_2[Ej_0] \wedge :C \notin M_2[Ej_1] \wedge \ldots \wedge :C \notin M_2[Ej_n]$$

But $:C \in M_2[Ej_0]$ means that Ej_0 abstracts* E_b. Since Ej_0 is not basic, at least one of $Ej_1, \ldots, Ej_n$ abstracts E_b. WLG, suppose it is Ej_1. Then

$$:C \in M_2[E_b] \Rightarrow$$
$$:C \in M_2[Ej_1]$$

which is a contradiction. Therefore all members of EXA hold in M_2, so M_2 is closed under specialization.

(Conclusion of proof)

Furthermore, M_1 and M_2 agree on all terms and predicates outside of H_E–{AnyEvent}. Because the extension of E_2 in M_2 is a proper subset of the extension of E_2 in M_1, M_2 defeats M_1's candidacy for minimality in H_E–{AnyEvent} among models closed under specialization. Thus the statement in Theorem 3.5 holds in all models closed under abstraction, and the subset of A-closed models.

Q.E.D.

Theorem 3.6

Let DJA be the set of all statements which instantiate Theorem 3.5 for a particular H. If M_1 is a model of $H_A \cup$ EXA $\cup$ DJA such that $:C \in M_1[E_0]$, then there is a unique basic event type E_b such that $:C \in M_1[E_b]$. Any event type which holds of :C abstracts* E_b.

Proof

By Theorem 3.2 there must be at least one $E_b \in H_{EB}$ such that $:C \in M_1[E_b]$. By definition any two distinct basic event types are not compatible; thus DJA contains an axiom of the form

$\forall x . \neg E_b(x) \vee \neg E_{b'}$
for all other basic event types $E_{b'}$. Therefore E_b is unique. By Theorem 3.2 any event type which holds of :C must abstract* E_b.

Q.E.D.

Theorem 3.7 (Unique Basic Types)

If M_1 is a model of H_A closed under abstraction containing event token $:C_1$, then there is a unique basic event type E_b such that $:C \in M_1[E_b]$. Any event type which holds of :C abstracts* E_b.

Proof

By Theorems 3.1 and 3.5, M1 is a model of $H_A \cup EXA \cup DJA$. By Theorem 3.6 there is such an E_b.

Q.E.D.

Theorem 3.8

If M_1 is an model of $H_A \cup EXA \cup DJA$, then M_1 is a model of H_A closed under abstraction.

Proof

Suppose not; then there is an M_2 closed under specialization which defeats M_1 for minimality in H_E–{AnyEvent}. M_1 and M_2 agree on AnyEvent, but there exists (at least one) $E_0 \in H_E$–{AnyEvent} and event :C such that

$:C \in M_1[E_0] \wedge :C \notin M_2[E_0]$

Now

$:C \in M_1[E_0] \Rightarrow$

$:C \in M_1[\text{AnyEvent}] \Rightarrow$

$:C \in M_2[\text{AnyEvent}]$

By Theorem 3.3 there is some $E_b \in H_{EB}$ such that

$:C \in M_2[E_b]$

Since M_2 defeats M_1,

$:C \in M_1[E_b]$

By Theorem 3.6, E_b is the unique basic type of :C in M_1, and E_0 abstracts* E_b. But E_0 abstracts* E_b means that

$:C \in M_2[E_b] \Rightarrow$

$:C \in M_2[E_0]$

which is a contradiction. Therefore there can be no such M_2, and M_1 must be closed under abstraction.

Q.E.D.

Theorem 3.9 (Abstraction Completeness)

M_1 is an A-closed model of H if and only if M_1 is a model of $H \cup EXA \cup DJA$.

Proof

If M_1 is an A-closed model of H, then it is also a model of H_A closed under abstraction, so by Theorems 3.1 and 3.5 it is a model of $H \cup EXA \cup DJA$. If M_1 is a model of $H \cup EXA \cup DJA$, then it is also a model of $H_A \cup EXA \cup DJA$, so by Theorem 3.8 it is a model of H_A closed under abstraction. Since it is also a model of H, it is an A-closed model of H.

Q.E.D.

Theorem 3.10 (No Useless Events)

Let M_1 be a covering model of H, containing event token $:C_1$. Then either $:C_1 \in M_1[\text{End}]$ is true, or there exists some event token $:C_2$ such that $:C_1$ is a direct component of $:C_2$.

Proof

Suppose the lemma is false: M is a covering model,

$:C_1 \in M_1[E_1]$

$:C_1 \notin M_1[\text{End}]$

and there does not exist event token which has $:C_1$ as a direct component. Define M_2 as follows.

$M_2[Z] = M_1[Z]$ for $Z \notin H_E$

$M_2[E] = M_1[E] - \{:C1\}$ for $E \in H_E$

Note that M_1 and M_2 agree on End. If M_2 is an A-closed model of H_A, then M_2 defeats M_1 for minimality in $H_E - \{\text{End}\}$. We will prove this by showing that $H \cup \text{EXA} \cup \text{DJA}$ holds in M_2. We consider each of the types of axioms in turn.

(Case 1) Axioms in H_G must hold, because they receive the same valuation in M_1 and M_2.

(Case 2) Axioms in H_A are of the form:

$\forall x \,.\, E_j(x) \supset E_i(x)$

Suppose one is false; then for some :D,

$:D \in M_2[E_j] \wedge :D \notin M_2[E_i]$

But this is impossible, because M_1 and M_2 must agree when $:D \neq :C_1$, as must be case, because $:C_1$ does not appear in the extension of any event type in M_2.

(Case 3) Axioms in EXA must hold by the same argument.

(Case 4) Axioms in DJA must hold because they contain no positive uses of H_E.

(Case 5) The j-th axiom in H_D is of the form:

$$\forall x \,.\, E_{j0}(x) \supset E_{j1}(f_{j1}(x)) \wedge E_{j2}(f_{j2}(x)) \wedge \ldots \wedge E_{jn}(f_{jn}(x)) \wedge \kappa$$

Suppose it does not hold in M_2. Then there must be some $:C_2$ such that

$$\begin{aligned} E_{j0}(x) \wedge \{ \quad & \neg E_{j1}(f_{j1}(x)) \vee \\ & \neg E_{j2}(f_{j2}(x)) \vee \ldots \vee \\ & \neg E_{jn}(f_{jn}(x)) \vee \\ & \neg \kappa \, \} \end{aligned}$$

is true in $M_2\{x/{:}C2\}$. M_1 and M_2 agree on κ, so it must the case that for some ji,

$$M_2[f_{ji}](:C_2) \notin M_2[E_{ji}]$$

while

$$M_1[f_{ji}](:C_2) \in M_1[E_{ji}]$$

Because M_1 and M_2 differ on Eji only at $:C_1$, it must be the case that

$$M_1[f_{ji}](:C_2) = :C_1$$

But then $:C_1$ <u>is</u> a component of $:C_2$ in M_1, contrary to our original assumption.

(End of Cases)

By Theorem 3.9 M_2 is A-closed, and so it defeats M_1. This contradiction proves the theorem.

Q.E.D.

Theorem 3.11 (Component/Use)

Let $E \in H_E$, and Com(E) be the set of event predicates with which E is compatible. Consider all the decomposition axioms in which any element of Com(E) appears on the right-hand side. The j-th such decomposition axiom has the following form, where E_{ji} is the element of Com(E):

$$\forall x \,.\, E_{j0}(x) \supset E_{j1}(f_{j1}(x)) \wedge \ldots \wedge E_{ji}(f_{ji}(x)) \wedge \ldots \wedge E_{jn}(f_{jn}(x)) \wedge \kappa$$

Suppose that the series of these axioms, where an axiom is repeated as many times as there are members of Com(E) in its right-hand side, is of length $m > 0$. Then the following statement is c–valid:

$$\begin{aligned}\forall x \,.\, E(x) \supset\ & End(x) \vee \\ & (\exists y \,.\, E_{1,0}(y) \wedge f_{1i}(y)=x) \vee \\ & (\exists y \,.\, E_{2,0}(y) \wedge f_{2i}(y)=x) \vee \\ & \ldots \vee \\ & (\exists y \,.\, E_{m,0}(y) \wedge f_{mi}(y)=x)\end{aligned}$$

Proof

Let M be a covering model such that :C $\in$ M[E] and :C $\notin$ M[End]. By Theorem 3.10 there is an E_{j0}, E_{ji}, and :D such that

:D $\in$ M[E_{j0}]

:C $\in$ M[E_{ji}]

:C = M[f_{ji}](:D)

where f_{ji} is a role function in a decomposition axiom for E_{j0}. By Theorem 3.5, E and E_{ji} are compatible. By inspection we see that the second half of the formula above is true in M when x is bound to :C, because the disjunct containing E_{ji} is true when the variable y is bound to :D. Since the choice of :C was arbitrary, the entire formula is true in M. Finally, since M could be any covering model, the formula is c-valid.

Q.E.D.

Theorem 3.12

Let CUA be the set of all formulas which instantiate Theorem 3.11 for a particular H. If M_1 is a model of H $\cup$ EXA $\cup$ DJA $\cup$ CUA such that :C $\in$ M_1[E], then there is a :C_n such that :C_n $\in$ M_1[End] and :C is a component* of :Cn.

Proof

If :C_1 $\in$ M1[End], then :C_1 is a component* of :C_n=:C_1. So suppose :C_1 $\notin$ M_1[End]. For the axioms in CUA to hold there must be sequences of event tokens, types, and role functions of the following form:

$:C_1$	$:C_1$	$:C_3$	$:C_3$	$:C_5$	$:C_5$	...
E_1	E_2	E_3	E_4	E_5	E_6	...
		f_{3i}		f_{5i}		...

such that

For all j, $:C_j \in M_1[E_j]$

For odd j, E_j and E_{j+1} are compatible

For odd j, $j \geq 3$, E_{j-1} is a direct component of E_j, and $:C_{j-2} = M_1[f_{ji}](:C_j)$

This sequence must terminate with a $:C_n$ such that $:C_n \in M_1[\text{End}]$. Otherwise, some E_i must appear more than once in the sequence, which would mean that a cycle exists in H; but we have supposed that H is acyclic. Therefore $:C_1$ is a component* of C_n.

Q.E.D.

Theorem 3.13 (No Infinite Chains)

If M_1 is a covering model of H such that $C_1 \in M_1[E]$, then there is a $:C_n$ such that $:C_n \in M_1[\text{End}]$ and $:C_1$ is a component* of $:C_n$.

Proof

By Theorems 3.9 and 3.8 M_1 is a model of $H \cup EXA \cup DJA \cup CUA$; by Theorem 3.12 there is such a $:C_n$.

Q.E.D.

Theorem 3.14 (Decomposition Completeness)

M_1 is a covering model of H if and only if M_1 is a model of $H \cup EXA \cup DJA \cup CUA$.

Proof

The "only if" half follows from Theorems 3.9 and 3.11. We prove that if M_1 is a model of $H \cup EXA \cup DJA \cup CUA$, then it is a covering model.

Suppose not. By Theorem 3.9, M_1 is A-closed, so there must be an A-closed M_2 which defeats M_1 for minimality in H_E–{End}. M_1 and M_2 agree on all functions and predicates outside of H_E–{End}, but there exists (at least one) $E_1 \in H_E$–{End} and event $:C_1$ such that

$$:C_1 \in M_1[E_1]$$
$$:C_1 \notin M_2[E_1]$$

By Theorem 3.12 there is a $:C_n$ such that

$$:C_n \in M_1[\text{End}]$$

and $:C_1$ is a component* of $:C_n$. By the construction used in the proof of Theorem 3.12 there are sequences

$$\begin{array}{lllllll} :C_1 & :C_1 & :C_3 & :C_3 & \dots & :C_n & :C_n \\ E_1 & E_2 & E_3 & E_4 & \dots & E_n & E_{n+1} = \text{End} \\ & & f_{3i} & & \dots & f_{ni} & \end{array}$$

such that

For all j, $1 \le j \le n+1$, $:C_j \in M_1[E_j]$
For odd j, E_j and E_{j+1} are compatible
For odd j, $3 \le j \le n$
 E_{j-1} is a direct component of E_j
 $:C_{j-2} = M_1[f_{ji}](:C_j) = M_2[f_{ji}](:C_j)$

Because M_1 and M_2 agree on End,

$$:C_n \in M_2[\text{End}]$$

Now for any odd j, $3 \le j \le n$, if

$$:C_j \in M_2[E_j]$$

then since H_D holds in M_2,

$$:C_{j-2} \in M_2[E_{j-1}]$$

<u>If</u> we prove that for all odd j, $1 \le j \le n$

$$:C_j \in M_2[E_{j+1}] \Rightarrow :C_j \in M_2[E_j]$$

we will be done; because we would then know that

$:C_n \in M_2[End] \Rightarrow$
$:C_n \in M2[E_{n+1}] \Rightarrow$
$:C_n \in M2[E_n] \Rightarrow$
$:C_{n-2} \in M2[E_{n-1}] \Rightarrow$
$\ldots \Rightarrow$
$:C_3 \in M_2[E_3] \Rightarrow$
$:C_1 \in M_2[E_2] \Rightarrow$
$:C_1 \in M_2[E_1] \Rightarrow$

which yields the desired contradiction. So lets prove that

$:C_j \in M_2[E_{j+1}] \Rightarrow :C_j \in M_2[E_j]$

Assume the antecedent

$:C_j \in M_2[E_{j+1}]$

Because M_2 is A-closed, there is a unique $E_b \in H_{EB}$ such that

$:C_j \in M_2[E_b]$

Because M_2 defeats M_1,

$:C_j \in M_1[E_b]$

Since M_1 is A-closed, Theorem 3.7 tells us that because

$:C_j \in M_1[E_j]$

it must be the case that E_j abstracts* E_b. But then since H_A holds in M_2,

$:C_j \in M_2[E_j]$

and we are done.

Q.E.D.

Theorem 3.15 (Computability of C-Entailment)

There is a computable function **cl** which maps a hierarchy H into a set of axioms with the property that

$$Q \;{}_H\!\models_c P$$

if and only if

$$cl(H) \cup Q \models P$$

Proof

The function simply computes $H \cup EXA \cup DJA \cup CUA$, which is a finite, recursively enumerable set. Theorem 3.14 guarantees the equivalence.

Theorem 3.16 (Not Predicate Completion)

Theorem 3.11 cannot be strengthened by considering only axioms in which E appears as a component, instead of all axioms in which event types compatible with E appear as components.

Proof

Consider the following hierarchy:

H_A

$$\forall x . E_1(x) \supset AnyEvent(x)$$
$$\forall x . E_2(x) \supset AnyEvent(x)$$
$$\forall x . E_3(x) \supset E_1(x)$$
$$\forall x . E_3(x) \supset E_2(x)$$
$$\forall x . End(x) \supset AnyEvent(x)$$
$$\forall x . E_4(x) \supset End(x)$$
$$\forall x . E_5(x) \supset End(x)$$

H_D

$$\forall x . E_4(x) \supset E_1(f_4(x)) \wedge P$$
$$\forall x . E_5(x) \supset E_2(f_5(x)) \wedge Q$$

The modified statement would not hold in the covering model

$$\{ AnyEvent(D), End(D), E_4(D), E_1(f_4(D)), E_3(f_4(D)), E_2(f_4(D)), AnyEvent(f_4(D)) \}$$

because even though $E_1(f_4(D))$ is true, there is no instance of E_5.

Q.E.D.

Theorem 3.17 (C-entailment and Circumscription)

For a given a hierarchy H, a statement P is c-entailed by Q if and only if P follows from the following schema:

$$
\begin{aligned}
&Q \wedge \mathrm{Circum}(\ H \wedge \mathrm{Circum}(\ \mathrm{Circum}(\ H_A\ ,\ H_E - H_{EB}\ ), \\
&\qquad\qquad\qquad\qquad\qquad \{\mathrm{AnyEvent}\}), \\
&\qquad\qquad H_E - \{\mathrm{End}\})
\end{aligned}
$$

Note that the two inner circumscriptions apply only to the abstraction hierarchy, while the final circumscription applies to all of H.

Proof

See [Etherington 86] for proof of the correctness of circumscription. Since the result of all the minimizations can be described by a finite set of axioms, we strongly suspect that circumscription is complete in this case. (No proof, however, has appeared in the literature.)

Q.E.D.

Appendix B
Chapter 4 Proofs

Theorem 4.1 (Minimum Cardinality Defaults)

Consider the following sequences of statements.

MA_0. $\forall x . \neg End(x)$

MA_1. $\forall x,y . End(x) \wedge End(y) \supset x=y$

MA_2. $\forall x,y,z . End(x) \wedge End(y) \wedge End(z)$
$\supset (x=y) \vee (x=z) \vee (y=z)$

...

The first asserts that no End events exist; the second, no more than one End event exists; the third, no more than two; and so on. Let P be any sentence, Γ any sentence or set of sentences, and H a hierarchy. Suppose there is a minimum covering model in which the extension of End is finite. Then

$$\Gamma\ {}_{H}\!\models_{mc} \Omega$$

if and only if

$$\Gamma \cup cl(H) \cup MA_i \vdash \Omega$$

where i is the smallest integer such that left-hand side of the provability relation is consistent.

Proof

We'll prove that M_1 is a minimum cover of Γ relative to H (with finite extension of End) if and only if M_1 is a model of $\Gamma \cup cl(H) \cup MA_i$. The theorem then follows from completeness and correctness of first-order logic.

Suppose M_1 is a model of $\Gamma \cup cl(H) \cup MA_i$. By Theorem 3.14, M_1 is a covering model. Clearly $|M_1[End]| \leq i$. Suppose that M_1 were defeated for minimum cardinality in End by M_2. Then $|M_2[End]| = j < i$. But by Theorem 3.14 M_2 is a model of $\Gamma \cup cl(H)$. Because $|M_2[End]| = j$, M_2 is also a model of MA_j. But this is impossible, because we've assumed that $\Gamma \cup cl(H) \cup MA_j$ is inconsistent. So M_1 cannot be defeated, and must be a minimum cover.

Suppose M_1 is such a minimum cover, where $|M_1[End]| = n$. As before, M_1 must be model of $\Gamma \cup cl(H)$. Suppose it were not a model of MA_i. Then $|M_1[End]| > i$. But because $\Gamma \cup cl(H) \cup MA_i$ is consistent, it has a model M_2, which is also be a covering model, with $|M_2[End]| \leq i$. But this is impossible, because M_2 would defeat M_1. Therefore MA_i must also be true in M_1.

Q.E.D.

Theorem 4.2 (Cardinality Circumscription)

Let α include all the predicate, function, and constant symbols in our language other than End. Suppose that all models of H are infinite, and in some model of $\Gamma \cup cl(H)$, End has a finite extension. If

$$\text{Circum}(\Gamma \cup cl(H), \{End\}, \alpha) \vdash \Omega$$

then

$$\Gamma_H \models_{mc} \Omega$$

where $\text{Circum}(\Gamma \cup cl(H), \{End\}, \alpha)$ means to circumscribe with α varying. The "if" is strengthened to "if and only if" if it is true that circumscription is complete in this case.

Proof

We prove that M_1 is a minimum covering model of Γ relative to H if and only if M_1 is minimal in End among models of $\Gamma \cup cl(H)$ where α varies. The theorem then follows from correctness of circumscription. The strengthened version of the theorem depends upon the truth of the proposition that circumscription is complete when the result of the circumscription is equivalent to a finite set of first-order formulas. Although this proposition is very likely to be true, no proof has yet been discovered [Etherington 87].

(if) Suppose M_1 is a minimum covering model of Γ. By Theorem 3.14 M_1 is a model of $\Gamma \cup cl(H)$. Suppose M_1 were *not* minimal in End where α varies; that is, there is an M_2 such that

1. M_2 is a model of $\Gamma \cup cl(H)$
2. $domain(M_1) = domain(M_2)$
3. $M_2[End] \subset M_1[End]$

But (1) implies M_2 is a covering model, and (2) implies that $|M_2[End]| < |M_1[End]|$, so M_2 would defeat M_1's candidacy as a minimum covering model. So there can be no such M_2, and M_1 *is* minimal in End where α varies.

(only if) Suppose M_1 is minimal in End among models of $\Gamma \cup cl(H)$ where α varies. Suppose M_1 is *not* a minimum covering model of Γ. Since M_1 is a covering model, there must be an M_2 which defeats M_1's candidacy to be a minimum covering model, with

$$|M_2[End]| < |M_1[End]|$$

We will construct a model which defeats M_1's candidacy for minimality in End where α varies, yielding the desired contradiction. But M_2 itself is not suitable, because its domain and M_1's domain may differ, and may not even be of the same size. So consider the following two cases:

1. Suppose $|Domain(M_1)| \leq |Domain(M_2)|$. Apply the Downward Tarski-Löwenheim-Skolem Theorem [Barwise 77] and let M_3 be an elementary submodel of M_2 such that $|Domain(M_1)| = |Domain(M_3)|$.

2. Otherwise |Domain(M_1)| > |Domain(M_2)|. Apply the Upward Tarski-Löwenheim-Skolem Theorem and let M_3 be an elementary extension of M_2 such that |Domain(M_1)| = |Domain(M_3)|.

Because M_3 is an elementary submodel or extension, and M_2 is a model of $\Gamma \cup$ cl(H), M_3 is a model of $\Gamma \cup$ cl(H). Furthermore, we have assumed that $|M_2[\text{End}]|$ is finite, let's say of size n. Then the sentence MA_n is true in M_2, and therefore also in M_3. This means that $|M_3[\text{End}]| \leq |M_2[\text{End}]|$. Thus M_3 is a covering model which defeats M_1's candidacy to be a minimum covering model.

Let M_4 be a homomorphism of M_3 such that Domain(M_4)=Domain(M_1). Now we have a model of $\Gamma \cup$ cl(H) which has the same domain as M_1, but is smaller in the size of its extension of End. But we need a model where the extension of End which is a subset of that in M_1.

Suppose

$M_1[\text{End}] = \{ :A_1, :A_2, \ldots :A_j, :C_1, :C_2, \ldots :C_k, :C_{k+1}, \ldots \}$

$M_4[\text{End}] = \{ :A_1, :A_2, \ldots :A_j, :B_1, :B_2, \ldots :B_k \}$

That is, $:A_1$ through $:A_j$ (where possibly j=0) are the elements in End that M_1 and M_2 have in common. Elements $:C_1$ and up are in End only in M_1, and $:B_1$ through $:B_k$ are only in End in M_4. The postfix substitution operator {:A/:B} replaces all instances of :A by instances of :B. Let θ be the substitution which swaps all the :Bi with :Ci.

$\theta = \{:B_1/:C_1, \ldots :B_k/:C_k, :C_1/:B_1, \ldots :C_k/:B_k\}$

Define M_5 as follows.

$M_5[p] = M_2[p]\theta$ for all predicates p

$M_5[c] = M_2[c]\theta$ for all constants c

$M_5[f] = \lambda x_1, x_2, \ldots x_n \,.\, (M_2[f](x_1\,\theta, x_2\,\theta, \ldots x_n\,\theta))\theta$

for all n-ary functions f

M_5 is a homomorphism of M_4, and is therefore a model of $\Gamma \cup$ cl(H). The proper subset condition is satisified because

$M_5[\text{End}] = \{:A_1, \ldots :A_j, :C_1, \ldots :C_k\} \subset M_1[\text{End}]$

Because α includes all symbols other than End, M_1 and M_5 do not have to agree on the interpretation of any symbols. So M_5 defeats M_1's candidacy for minimality in End where α varies, which is a contradiction. Therefore M_1 is a minimum covering model of Γ.

Q.E.D.

Appendix C
Chapter 5 Proofs

Theorem 5.1 (Non-Universal Conclusions)

Let Ω be a sentence which, when written with all quantifiers in initial position, contains no universal quantifiers. Then

$$\Gamma_H \models_{mcs} \Omega$$

if and only if

$$\Gamma_H \models_{mc} \Omega$$

Proof

(Only if) Trivial, since a minimum cover is an mcs-model.

(If) Ω can be written

$$\exists\, x_1, \ldots x_n \,.\, \omega$$

where ω is a sentence containing no quantifiers in which $x_1, \ldots x_n$ appear free. Suppose Ω holds in all minimum covers. Let M_1 be any mcs-model. Then M_1 contains submodel M_2 in which

$$\exists\, x_1, \ldots x_n \,.\, \omega$$

But this is only the case if there are

$$:C_1, \ldots :C_n \in \mathrm{Domain}(M_2)$$

such that ω is true in $M_2\{x_1/:C_1, \ldots x_n/:C_n\}$. A straightforward inductive argument shows that ω is true in $M_1\{x_1/:C_1, \ldots x_n/:C_n\}$ because all terms in ω are interpreted by $M_1\{x_1/:C_1, \ldots x_n/:C_n\}$ as individuals in the domain of M_2. Therefore Ω holds in M_1, and in all mcs-models.

Q.E.D.

Theorem 5.2 (Incremental Recognition)

In the case of a single observation, imc-entailment is the same as mcs-entailment.

$$(\Gamma_1)\ {}_H\models_{imc} \Omega$$

if and only if

$$\Gamma_1\ {}_H\models_{mcs} \Omega$$

In the case of multiple observations, imc-entailment is monotonic.

$$(\Gamma_1, \ldots \Gamma_{n-1})\ {}_H\models_{imc} \Omega$$

implies

$$(\Gamma_1, \ldots \Gamma_{n-1}, \Gamma_n)\ {}_H\models_{imc} \Omega$$

Proof

The first part of the theorem follows from the base case of the definition of imc-entailment. The second part of the theorem follows from the fact that an incremental minimum cover of $(\Gamma_1, \ldots \Gamma_{n-1}, \Gamma_n)$ must also be an incremental minimum cover of $(\Gamma_1, \ldots \Gamma_{n-1})$. Therefore if Ω holds in all models of the latter sort, it must also hold in all models of the former sort.

Appendix D
Temporal Constraint Logic

The following rules translate Allen's temporal interval operators into an algebra on fuzzy interval constraints. A *time interval* is represented by a term which is *interpreted* as a pair of real numbers. The functions **first-instant** and **last-instant** apply to a interval and yield the lower and upper bounds respectively. A *fuzzy interval constraint* is a tuple of four real numbers. A time interval *satisfies* a fuzzy constraint, written

$$I \in (\text{start-min, start-max, end-min, end-max})$$

if the first instance of I falls between start-min and start-max inclusive, and the final instance of I falls between end-min and end-max inclusion. That is,

$$(\text{start-min} \leq \text{first-instant}(I) \leq \text{start-max}) \wedge$$
$$(\text{end-min} \leq \text{last-instant}(I) \leq \text{end-max})$$

While start-min, start-max, etc., are represented by *numerals*, first-instant(I) and last-instant(I) remain *symbolic* quantities.

The form of each each rule is as follows: given that T_1 satisfies fuzzy constraint Z_1, and T_2 satisfies fuzzy constraint Z_2, and T_1 is related by a given binary-operator to T_2, then it must be the case that T_1 satisfies fuzzy constraint Z_3. The logical form of the rules is as follows, where $Z_3 = \text{Ftransitive}(Z_1, \text{binary-op}, Z_2)$.

Where T_1, T_2 are time intervals, and Z_1,Z_2 are fuzzy constraints:

$$T_1 \in Z_1 \wedge T_2 \in Z_2 \wedge T_1 \text{ binary-op } T_2 \supset$$
$$T_1 \in \text{Ftransitive}(Z_1, \text{binary-op}, Z_2)$$

Suppose that T_1 and T_2 are related by a *disjunction* of temporal operators. Then one calculates Z_3 for each alternative, and then combine all the answers with the **Funion** function, which is defined below. In formal terms:

$$T_1 \in Z_1 \wedge T_2 \in Z_2 \wedge T_1 \ (op_1\ op_2 \ldots op_n)\ T_2 \supset$$
$$T_1 \in \text{Funion}(\ \text{Ftransitive}(Z_1, op_1, Z_2),$$
$$\text{Ftransitive}(Z_1, op_2, Z_2),$$
$$\ldots, \text{Ftransitive}(Z_1, op_n, Z_2)\ )$$

Where T_1 (op_1 op_2 ... op_n) $T_2 \equiv$

$$(T_1\ op_1\ T_2) \vee (T_1\ op_2\ T_2) \vee \ldots \vee (T_1\ op_n\ T_2)$$

Consider the case where an interval is known to satisfy two different fuzzy constraints. Then it must be the case that the interval satisfies the Fintersection of the constraints. Formally:

$$T_1 \in Z_1 \wedge T_1 \in Z_2 \supset T_1 \in \text{Fintersection}(Z_1, Z_2)$$

Finally, the predicate **empty** is true of a fuzzy constraint just in case no interval could satisfy it.

The following tables define the **Funion, Fintersection, empty** and **Ftransitive** functions. Their correctness can be verified by elementary algebra.

Funion((a_1 b_1 c_1 d_1), ..., (a_n b_n c_n d_n)) =

$$(\ \min(a_1, \ldots, a_n)\ \ \max(b_1, \ldots, b_n)\ \ \min(c_1, \ldots, c_n)\ \ \max(d_1, \ldots, d_n)\ )$$

Fintersection((a_1 b_1 c_1 d_1), ..., (a_n b_n c_n d_n)) =

$$(\ \max(a_1, \ldots, a_n)\ \ \min(b_1, \ldots, b_n)\ \ \max(c_1, \ldots, c_n)\ \ \min(d_1, \ldots, d_n)\ )$$

empty((a b c d)) =

$$a > b \vee c > d \vee a > d$$

Ftransitive((a b c d), op, (e f g h)) = (w x y z)

op	w	x	y	z
before	a	min(b,f)	c	min(d,f)
meets	a	min(b,f)	max(c,e)	min(d,f)
overlaps	a	min(b,f)	max(c,e)	min(d,h)
starts	max(a,e)	min(b,f)	max(c,e)	min(d,h)
during	max(a,e)	min(b,h)	max(c,e)	min(d,h)
finishes	max(a,e)	min(b,h)	max(c,g)	min(d,h)
overlapped by	max(a,e)	min(b,h)	max(c,g)	d
met by	max(a,g)	min(b,h)	max(c,g)	d
after	max(a,g)	b	max(c,g)	d

Appendix E
Transcripts

Following are transcripts of the implementation running on some of the examples discussed in this thesis.

Hunt/Rob Example

```
;; First observation:  get–gun(GET-GUN1).  Builds graph named g1.

Command: (EXPLAIN–OBSERVATION '(GET-GUN1 GET–GUN) 'G1)
 T


;; Display e-graph g1.
;; There are two alternatives, with GET-GUN1 as component of rob–bank or of hunt.
Command: (DRAW–GRAPH)
graph G1
END226
    ?= HUNT228
       ?= HUNT227
          S1 –> GET-GUN1
    ?= ROB–BANK225
       ?= ROB–BANK224
          S1 –> GET-GUN1   ^
```

The e-graphs are printed out linearly, where the symbol ?= represents an ***alternative*** *link and* ***rolename->*** *represents a component (and) link. The symbol ^ indicates a link to structure already printed above. This description is equivalent to the following diagram:*

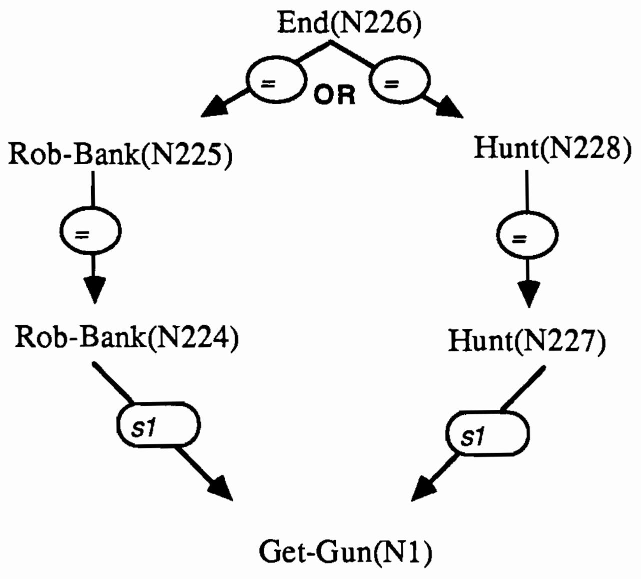


figure E.1: E-Graph for Get-Gun

The one difference from the algorithm described in Chapter 7 is the inclusion of nodes which "abstract away" the steps of any node which has explicit component links. In this example, HUNT228 abstracts HUNT227, but doesn't include the component S1. These extra nodes do not change the semantics of e-graphs, but they do allow a more compact representation of the graphs when the type to be explained can fill several different roles in a user type. In such a case, a different user node is created for each use, but all can become alternatives for the same abstract node of that user type.

```
;; Turn on tracing.

Command: (SETQ *NOISY* T)
 T

;; Second observation: go–to–bank(GO-TO-BANK2), in graph G2.

Command: (EXPLAIN–OBSERVATION '(GO-TO-BANK2 GO–TO–BANK) 'G2)
 ... creating node GO-TO-BANK2
 ... searching up from GO-TO-BANK2
... creating node CASH–CHECK229
... considering GO-TO-BANK2 as S1 of CASH–CHECK229
... checking constraints on CASH–CHECK229
... CASH–CHECK229 is okay
... searching up from CASH–CHECK229
... creating node CASH–CHECK230
... searching up from CASH–CHECK230
... creating node END231
... searching up from END231
... creating node ROB–BANK232
... considering GO-TO-BANK2 as S2 of ROB–BANK232
... checking constraints on ROB–BANK232
... ROB–BANK232 is okay
... searching up from ROB–BANK232
... creating node ROB–BANK233
... searching up from ROB–BANK233
... merging graph at END231
 T

;; Show g2, again two alternatives:  rob–bank or cash–check.

Command: (DRAW–GRAPH)
graph G2
END231
    ?= ROB–BANK233
       ?= ROB–BANK232
          S2 –> GO-TO-BANK2
    ?= CASH–CHECK230
       ?= CASH–CHECK229
          S1 –> GO-TO-BANK2  ^

;; Matching the graphs yields just the rob–bank alternative.

Command: (MATCH–GRAPHS 'G1 'G2 'G1+G2)
```

```
... trying to match END226 with END231
... creating node END234
... checking constraints on END234
... END234 is okay
... checking constraints on END234
... END234 is okay
... trying to match HUNT228 with ROB–BANK233
... trying to match HUNT228 with CASH–CHECK230
... trying to match ROB–BANK225 with ROB–BANK233
... creating node ROB–BANK235
... checking constraints on ROB–BANK235
... ROB–BANK235 is okay
... checking constraints on ROB–BANK235
... ROB–BANK235 is okay
... trying to match ROB–BANK224 with ROB–BANK232
... creating node ROB–BANK236
... checking constraints on ROB–BANK236
... ROB–BANK236 is okay
... creating node GO–TO–BANK237
... checking constraints on GO–TO–BANK237
... GO–TO–BANK237 is okay
... checking constraints on GO–TO–BANK237
... GO–TO–BANK237 is okay
... copying GO-TO-BANK2 yields GO–TO–BANK237
... creating node GET–GUN238
... checking constraints on GET–GUN238
... GET–GUN238 is okay
... checking constraints on GET–GUN238
... GET–GUN238 is okay
... copying C1 yields GET–GUN238
... checking constraints on ROB–BANK236
... ROB–BANK236 is okay
... successful match of ROB–BANK224 and ROB–BANK232 yields
ROB–BANK236
```

```
... successful match of ROB-BANK225 and ROB-BANK233 yields
ROB-BANK235
... trying to match ROB-BANK225 with CASH-CHECK230
... successful match of END226 and END231 yields END234
END234


Command: (DRAW-GRAPH)
graph G1+G2
END234
    ?= ROB-BANK235
       ?= ROB-BANK236
          S1 -> GET-GUN238
          S2 -> GO-TO-BANK237
```

Cooking Examples

```
;; First observation is make-sauce(OBS-SAUCE2), with agent Joe, during
;; time interval beginning between 4 and 5, and ending between 6 and 7.
```

Note that fuzzy time constraints are represented by VECTORS with TIME as first element. The parameters of the observation are specified by a list of (role value) pairs. The first use of the symbol "time" below refers to the role Time, while the second merely identifies the vector value as a time constraint.

```
Command: (EXPLAIN-OBSERVATION '(OBS-SAUCE2 MAKE-SAUCE
                        (AGENT JOE) (TIME #(TIME 4 5 6 7))))
... creating node OBS-SAUCE2
... searching up from OBS-SAUCE2
... creating node MAKE-PASTA-DISH246
... considering OBS-SAUCE2 as S2 of MAKE-PASTA-DISH246
... checking constraints on MAKE-PASTA-DISH246
... MAKE-PASTA-DISH246 is okay
... searching up from MAKE-PASTA-DISH246
... creating node MAKE-PASTA-DISH247
... searching up from MAKE-PASTA-DISH247
... creating node PREPARE-MEAL248
... searching up from PREPARE-MEAL248
... creating node END249
... searching up from END249
... creating node MAKE-MARINARA250
... checking constraints on MAKE-MARINARA250
... MAKE-MARINARA250 is okay
... searching up from MAKE-MARINARA250
... creating node MAKE-CHICKEN-MARINARA251
... considering MAKE-MARINARA250 as S5 of
MAKE-CHICKEN-MARINARA251
... checking constraints on MAKE-CHICKEN-MARINARA251
... MAKE-CHICKEN-MARINARA251 is okay
... searching up from MAKE-CHICKEN-MARINARA251
... creating node MAKE-CHICKEN-MARINARA252
... searching up from MAKE-CHICKEN-MARINARA252
... creating node MAKE-MEAT-DISH253
... searching up from MAKE-MEAT-DISH253
... merging graph at PREPARE-MEAL248
 T

;; View the graph.
```

The parameters of each node appear in a list of role/value pairs after the node name. Only parameters for which a value has been calculated are included.

```
Command: (DRAW-GRAPH T)
graph G2
END249 NIL
    ?= PREPARE-MEAL248 ((AGENT JOE) (TIME #(TIME -INF 5 6 +INF)))
    ?= MAKE-MEAT-DISH253
              ((AGENT JOE) (TIME #(TIME -INF 5 6 +INF)))
       ?= MAKE-CHICKEN-MARINARA252
                   ((AGENT JOE) (TIME #(TIME -INF 5 6 +INF)))
          ?= MAKE-CHICKEN-MARINARA251
                   ((AGENT JOE) (TIME #(TIME -INF 5 6 +INF)))
             S5 -> MAKE-MARINARA250
                         ((AGENT JOE) (TIME #(TIME 4 5 6 7)))
    ?= MAKE-PASTA-DISH247
              ((AGENT JOE) (TIME #(TIME -INF 5 6 +INF)))
       ?= MAKE-PASTA-DISH246
                   ((AGENT JOE) (TIME #(TIME -INF 5 6 +INF)))
          S2 -> OBS-SAUCE2 ((AGENT JOE) (TIME #(TIME 4 5 6 7)))

;; The second observation is making noodles.

Command: (EXPLAIN-OBSERVATION '(OBS-NOODLES3 MAKE-NOODLES
                   (AGENT JOE) #(TIME 6 8 7 9)))
 T


Command: (DRAW-GRAPH T)
graph G3
END257 NIL
   ?= PREPARE-MEAL256
              ((AGENT JOE) (TIME #(TIME -INF 8 7 +INF)))
      ?= MAKE-PASTA-DISH255
              ((AGENT JOE) (TIME #(TIME -INF 8 7 +INF)))
         ?= MAKE-PASTA-DISH254
                   ((AGENT JOE) (TIME #(TIME -INF 8 7 +INF)))
            S1 -> OBS-NOODLES3 ((AGENT JOE) (TIME #(TIME 6 8 7 9)))

;; We can merge G2 and G3 together: they can be steps of the same action.
```

```
Command: (MATCH-GRAPHS 'G2 'G3 'G2+G3)
 ... trying to match END249 with END257
 ... creating node END258
 ... checking constraints on END258
 ... END258 is okay
 ... checking constraints on END258
 ... END258 is okay
 ... trying to match PREPARE-MEAL248 with PREPARE-MEAL256
 ... creating node PREPARE-MEAL259
 ... checking constraints on PREPARE-MEAL259
 ... PREPARE-MEAL259 is okay
 ... checking constraints on PREPARE-MEAL259
 ... PREPARE-MEAL259 is okay
 ... trying to match MAKE-MEAT-DISH253 with MAKE-PASTA-DISH255
 ... trying to match MAKE-PASTA-DISH247 with MAKE-PASTA-DISH255
 ... creating node MAKE-PASTA-DISH260
 ... checking constraints on MAKE-PASTA-DISH260
 ... MAKE-PASTA-DISH260 is okay
 ... checking constraints on MAKE-PASTA-DISH260
 ... MAKE-PASTA-DISH260 is okay
 ... trying to match MAKE-PASTA-DISH246 with MAKE-PASTA-DISH254
 ... creating node MAKE-PASTA-DISH261
 ... checking constraints on MAKE-PASTA-DISH261
 ... MAKE-PASTA-DISH261 is okay
 ... creating node MAKE-NOODLES262
 ... checking constraints on MAKE-NOODLES262
 ... MAKE-NOODLES262 is okay
 ... checking constraints on MAKE-NOODLES262
 ... MAKE-NOODLES262 is okay
 ... copying OBS-NOODLES3 yields MAKE-NOODLES262
 ... creating node MAKE-SAUCE263
 ... checking constraints on MAKE-SAUCE263
 ... MAKE-SAUCE263 is okay
 ... checking constraints on MAKE-SAUCE263
... MAKE-SAUCE263 is okay
... copying OBS-SAUCE2 yields MAKE-SAUCE263
... checking constraints on MAKE-PASTA-DISH261
... MAKE-PASTA-DISH261 is okay
... successful match of MAKE-PASTA-DISH246 and MAKE-PASTA-DISH254
yields MAKE-PASTA-DISH260
... successful match of MAKE-PASTA-DISH247 and MAKE-PASTA-DISH255
yields MAKE-PASTA-DISH261
... successful match of PREPARE-MEAL248 and PREPARE-MEAL256 yields
PREPARE-MEAL259
... successful match of END249 and END257 yields END258
 END258


Command: (DRAW-GRAPH T)
graph G2+G3
END258 NIL
    ?= PREPARE-MEAL259 ((AGENT JOE) (TIME #(TIME -INF 5 7 +INF)))
        ?= MAKE-PASTA-DISH260
                ((AGENT JOE) (TIME #(TIME -INF 5 7 +INF)))
            ?= MAKE-PASTA-DISH261
                    ((AGENT JOE) (TIME #(TIME -INF 5 7 +INF)))
                S2 -> MAKE-SAUCE263
                            ((AGENT JOE) (TIME #(TIME 4 5 6 7)))
                S1 -> MAKE-NOODLES262
                            ((AGENT JOE) (TIME #(TIME 6 8 7 9)))

;; Now consider an observation of make-noodles with a different agent.  The constants
;; Joe and Sally are rigid designators, and therefore unequal.

Command: (EXPLAIN-OBSERVATION '(OBS-NOODLES4 MAKE-NOODLES
                        (AGENT SALLY) #(TIME 6 8 7 9)))
 T
```

```
;; Try to match this with the original make-sauce.  It will fail, because agents differ.

Command: (MATCH-GRAPHS 'G2 'G4 'G2+G4)
 ... trying to match END249 with END267
 ... creating node END268
 ... checking constraints on END268
 ... END268 is okay
 ... checking constraints on END268
 ... END268 is okay
 ... trying to match PREPARE-MEAL248 with PREPARE-MEAL266
 ... creating node PREPARE-MEAL269
 ... equality constraint violated JOE = SALLY
 ... PREPARE-MEAL269 fails
 ... END268 fails
  NIL

;; Lets check out the temporal constraints, now.  We'll observe a boiling event
;; with time BEFORE the make-noodles event.  This conflict will prevent a match.

Command: (EXPLAIN-OBSERVATION '(OBS-BOILING5 BOIL
                               (TIME #(TIME 1 1 2 2))) 'G5)
  T


Command: (DRAW-GRAPH T)
graph G5
END273 NIL
    ?= PREPARE-MEAL272 ((TIME #(TIME -INF 1 2 +INF)))
       ?= MAKE-PASTA-DISH271 ((TIME #(TIME -INF 1 2 +INF)))
          ?= MAKE-PASTA-DISH270 ((TIME #(TIME -INF 1 2 +INF)))
             S3 -> OBS-BOILING5 ((TIME #(TIME 1 1 2 2)))

;; Try to match this with G2 and G3.  This will fail because of temporal constraint
violation.
```

```
Command: (MATCH-GRAPHS 'G2+G3 'G5 'G2+G3+G5)
 ... trying to match END258 with END273
 ... creating node END274
 ... checking constraints on END274
 ... END274 is okay
 ... checking constraints on END274
 ... END274 is okay
 ... trying to match PREPARE-MEAL259 with PREPARE-MEAL272
 ... creating node PREPARE-MEAL275
 ... checking constraints on PREPARE-MEAL275
 ... PREPARE-MEAL275 is okay
 ... checking constraints on PREPARE-MEAL275
 ... PREPARE-MEAL275 is okay
 ... trying to match MAKE-PASTA-DISH260 with MAKE-PASTA-DISH271
 ... creating node MAKE-PASTA-DISH276
 ... checking constraints on MAKE-PASTA-DISH276
 ... MAKE-PASTA-DISH276 is okay
 ... checking constraints on MAKE-PASTA-DISH276
 ... MAKE-PASTA-DISH276 is okay
 ... trying to match MAKE-PASTA-DISH261 with MAKE-PASTA-DISH270
 ... creating node MAKE-PASTA-DISH277
 ... checking constraints on MAKE-PASTA-DISH277
 ... MAKE-PASTA-DISH277 is okay
 ... creating node MAKE-NOODLES278
 ... checking constraints on MAKE-NOODLES278
 ... MAKE-NOODLES278 is okay
 ... checking constraints on MAKE-NOODLES278
 ... MAKE-NOODLES278 is okay
 ... copying MAKE-NOODLES262 yields MAKE-NOODLES278
 ... creating node MAKE-SAUCE279
 ... checking constraints on MAKE-SAUCE279
 ... MAKE-SAUCE279 is okay
 ... checking constraints on MAKE-SAUCE279
... MAKE-SAUCE279 is okay
... copying MAKE-SAUCE263 yields MAKE-SAUCE279
... creating node BOIL280
... checking constraints on BOIL280
... BOIL280 is okay
... checking constraints on BOIL280
... BOIL280 is okay
... copying OBS-BOILING5 yields BOIL280
... checking constraints on MAKE-PASTA-DISH277
... time constraint violated #(TIME 6 8 7 9) BEFOREMEET #(TIME 1 1 2 2)
... MAKE-PASTA-DISH277 fails
... MAKE-PASTA-DISH276 fails
... PREPARE-MEAL275 fails
... END274 fails
 NIL

;; Now let's find a LATER boiling event.  It will match okay.

Command: (EXPLAIN-OBSERVATION '(OBS-BOILING6 BOIL
                        (TIME #(TIME 9 10 11 12))) 'G6)
 T


Command: (MATCH-GRAPHS 'G2+G3 'G6 'G2+G3+G6)
 END285


Command: (DRAW-GRAPH T)
graph G2+G3+G6
END285 NIL
    ?= PREPARE-MEAL286 ((AGENT JOE) (TIME #(TIME -INF 5 11 +INF)))
       ?= MAKE-PASTA-DISH287
              ((AGENT JOE) (TIME #(TIME -INF 5 11 +INF)))
          ?= MAKE-PASTA-DISH288
```

```
                  ((AGENT JOE) (TIME #(TIME -INF 5 11 +INF))
      S3 -> BOIL291 ((TIME #(TIME 9 10 11 12)))
      S2 -> MAKE-SAUCE290
                        ((AGENT JOE) (TIME #(TIME 4 5 6 7)))
      S1 -> MAKE-NOODLES289
                        ((AGENT JOE) (TIME #(TIME 6 8 7 9)))
```

Operating System Examples

```
;; User enters: % copy foo bar.

Command: (EXPLAIN-OBSERVATION '(OBS-COPY1 COPY
                  (OLD FOO) (NEW BAR)) 'C1)
 T

;; Could be part of rename by copy, or of modify file.

Command: (DRAW-GRAPH T)
graph C1
END294 NIL
   ?= RENAME297 ((NEW BAR) (OLD FOO))
      ?= RENAME-BY-COPY296 ((NEW BAR) (OLD FOO))
         ?= RENAME-BY-COPY295 ((NEW BAR) (OLD FOO))
            COPY-ORIG-STEP -> OBS-COPY1 ((NEW BAR) (OLD FOO))
   ?= MODIFY293 ((FILE FOO))
      ?= MODIFY292 ((FILE FOO))
         BACKUP-STEP -> OBS-COPY1 ((NEW BAR) (OLD FOO))  ^

;; User enters: % copy jack sprat.

Command: (EXPLAIN-OBSERVATION '(OBS-COPY2 COPY
                  (OLD JACK) (NEW SPRAT)) 'C2)
```

```
T

;; Try (and fail) to unify these commands. File names are rigid designators,
;;  and so cannot be matched unless identical.

Command: (MATCH-GRAPHS 'C1 'C2 'C1+C2)
 ... trying to match END294 with END300
 ... creating node END304
 ... checking constraints on END304
 ... END304 is okay
 ... checking constraints on END304
 ... END304 is okay
 ... trying to match RENAME297 with RENAME303
 ... creating node RENAME305
 ... equality constraint violated BAR = SPRAT
 ... RENAME305 fails
 ... trying to match RENAME297 with MODIFY299
 ... trying to match MODIFY293 with RENAME303
 ... trying to match MODIFY293 with MODIFY299
 ... creating node MODIFY306
 ... equality constraint violated FOO = JACK
 ... MODIFY306 fails
 ... END304 fails
 NIL


;; So, there must be two different plans going on.
;; User enters: % delete foo

Command: (EXPLAIN-OBSERVATION '(OBS-DELETE3 DELETE
                        (FILE FOO)) 'C3)
 T
```

```
Command: (DRAW-GRAPH T)
graph C3
END310 NIL
    ?= MODIFY312 NIL
       ?= MODIFY311 NIL
          DELETE-BACKUP-STEP -> OBS-DELETE3 ((FILE FOO))
    ?= RENAME309 ((OLD FOO))
       ?= RENAME-BY-COPY308 ((OLD FOO))
          ?= RENAME-BY-COPY307 ((OLD FOO))
             DELETE-ORIG-STEP -> OBS-DELETE3 ((FILE FOO))  ^

;; This delete can unify with command 1, but not command 2

Command: (MATCH-GRAPHS 'C1 'C3 'C1+C3)
 END313


Command: (DRAW-GRAPH T)
graph C1+C3
END313 NIL
    ?= RENAME314 ((NEW BAR) (OLD FOO))
       ?= RENAME-BY-COPY315 ((NEW BAR) (OLD FOO))
          ?= RENAME-BY-COPY316 ((NEW BAR) (OLD FOO))
             DELETE-ORIG-STEP -> DELETE318 ((FILE FOO))
             COPY-ORIG-STEP -> COPY317 ((NEW BAR) (OLD FOO))

;; Command 2 and 3 cannot be part of the end plan

Command: (MATCH-GRAPHS 'C2 'C3 'C2+C3)
... trying to match END300 with END310
... creating node END321
... checking constraints on END321
... END321 is okay
... checking constraints on END321
... END321 is okay
... trying to match RENAME303 with MODIFY312
... trying to match RENAME303 with RENAME309
... creating node RENAME322
... equality constraint violated JACK = FOO
... RENAME322 fails
... trying to match MODIFY299 with MODIFY312
... creating node MODIFY323
... checking constraints on MODIFY323
... MODIFY323 is okay
... checking constraints on MODIFY323
... MODIFY323 is okay
... trying to match MODIFY298 with MODIFY311
... creating node MODIFY324
... checking constraints on MODIFY324
... MODIFY324 is okay
... creating node COPY325
... checking constraints on COPY325
... COPY325 is okay
... checking constraints on COPY325
... COPY325 is okay
... copying OBS-COPY2 yields COPY325
... creating node DELETE326
... checking constraints on DELETE326
... DELETE326 is okay
... checking constraints on DELETE326
... DELETE326 is okay
... copying OBS-DELETE3 yields DELETE326
... checking constraints on MODIFY324
... equality constraint violated FOO = SPRAT
... MODIFY324 fails
... MODIFY324 fails
... MODIFY323 fails
... trying to match MODIFY299 with RENAME309
... END321 fails
 NIL
```

Language Examples

```
;; Joe says to Sally:   "Can you give me the salt?"

Command: (EXPLAIN-OBSERVATION '(TALK1 SURFACE-QUESTION
                          (SPEAKER JOE)
                          (HEARER SALLY)
                          (TIME #(TIME 4 4 5 5))
                          (CONTENT (CAN SALLY (GAVE SALLY JOE SALT))))
                       'UTTER1)
 ... creating node TALK1
 ... searching up from TALK1
 ... creating node DIRECT-REQUEST360
 ... considering TALK1 as S of DIRECT-REQUEST360
 ... checking constraints on DIRECT-REQUEST360
 ... DIRECT-REQUEST360 is okay
 ... searching up from DIRECT-REQUEST360
 ... creating node DIRECT-REQUEST361
 ... searching up from DIRECT-REQUEST361
 ... creating node REQUEST362
 ... searching up from REQUEST362
 ... creating node OBTAIN-BY-ASKING363
 ... considering REQUEST362 as S1 of OBTAIN-BY-ASKING363
 ... checking constraints on OBTAIN-BY-ASKING363
 ... equality constraint violated GAVE = INFORMEDIF
 ... OBTAIN-BY-ASKING363 fails
 ... OBTAIN-BY-ASKING363 fails
 ... creating node FINDOUT-BY-ASKING364
 ... considering REQUEST362 as S1 of FINDOUT-BY-ASKING364
... checking constraints on FINDOUT-BY-ASKING364
... FINDOUT-BY-ASKING364 is okay
... searching up from FINDOUT-BY-ASKING364
... creating node FINDOUT-BY-ASKING365
... searching up from FINDOUT-BY-ASKING365
... creating node FINDOUT366
... searching up from FINDOUT366
... creating node END367
... searching up from END367
... creating node INDIRECT-REQUEST368
... considering TALK1 as S of INDIRECT-REQUEST368
... checking constraints on INDIRECT-REQUEST368
... INDIRECT-REQUEST368 is okay
... searching up from INDIRECT-REQUEST368
... creating node INDIRECT-REQUEST369
... searching up from INDIRECT-REQUEST369
... creating node REQUEST370
... searching up from REQUEST370
... creating node OBTAIN-BY-ASKING371
... considering REQUEST370 as S1 of OBTAIN-BY-ASKING371
... checking constraints on OBTAIN-BY-ASKING371
... OBTAIN-BY-ASKING371 is okay
... searching up from OBTAIN-BY-ASKING371
... creating node OBTAIN-BY-ASKING372
... searching up from OBTAIN-BY-ASKING372
... creating node OBTAIN373
... searching up from OBTAIN373
... merging graph at END367
... creating node FINDOUT-BY-ASKING374
... considering REQUEST370 as S1 of FINDOUT-BY-ASKING374
... checking constraints on FINDOUT-BY-ASKING374
... equality constraint violated INFORMEDIF = GAVE
... FINDOUT-BY-ASKING374 fails
... FINDOUT-BY-ASKING374 fails
```

```
 T

;; The statement is ambiguous, as we see:

Command: (DRAW-GRAPH T)
graph UTTER1
END367 NIL
    ?= OBTAIN373
                ((OBJECT SALT)
                (AGENT JOE)
                (TIME #(TIME -INF 4 5 +INF)))
                (PRETIME #(TIME -INF 4 -INF +INF))
        ?= OBTAIN-BY-ASKING372
                        ((AGENT JOE) (OBJECT SALT)
                        (TIME #(TIME -INF 4 -INF +INF))
                        (PRETIME #(TIME -INF 4 -INF +INF)))
            ?= OBTAIN-BY-ASKING371
                        ((AGENT JOE) (OBJECT SALT)
                        (TIME #(TIME -INF 4 -INF +INF))
                        (PRETIME #(TIME -INF 4 -INF +INF)))
                S1 -> REQUEST370
                                ((REQGOAL (GAVE SALLY JOE SALT))
                                (HEARER SALLY)
                                (SPEAKER JOE)
                                (TIME #(TIME 4 4 5 5)))
                    ?= INDIRECT-REQUEST369
                                ((SPEAKER JOE) (HEARER SALLY)
                                (REQGOAL (GAVE SALLY JOE SALT))
                                (TIME #(TIME 4 4 5 5)))
                        ?= INDIRECT-REQUEST368
                                ((SPEAKER JOE)
                                (HEARER SALLY)
                                (REQGOAL (GAVE SALLY JOE SALT))
                                (TIME #(TIME 4 4 5 5)))
                    S -> TALK1
                              ((CONTENT (CAN SALLY
                                        (GAVE SALLY JOE SALT)))
                              (HEARER SALLY)
                              (SPEAKER JOE)
                              (TIME #(TIME 4 4 5 5)))
?= FINDOUT366
          ((INFO (CAN SALLY (GAVE SALLY JOE SALT)))
          (AGENT JOE)
          (PRETIME #(TIME -INF 4 -INF +INF))
          (TIME #(TIME -INF 4 5 +INF)))
    ?= FINDOUT-BY-ASKING365
                    ((AGENT JOE)
                    (INFO (CAN SALLY (GAVE SALLY JOE SALT)))
                    (TIME #(TIME -INF 4 5 +INF))
                    (PRETIME #(TIME -INF 4 -INF +INF)))
        ?= FINDOUT-BY-ASKING364
                    ((AGENT JOE)
                    (INFO (CAN SALLY (GAVE SALLY JOE SALT)))
                    (TIME #(TIME -INF 4 5 +INF))
                    (PRETIME #(TIME -INF 4 -INF +INF)))
            S1 -> REQUEST362
                    ((REQGOAL (INFORMEDIF SALLY JOE
                              (CAN SALL (GAVE SALLY JOE SALT)))
                    (HEARER SALLY)
                    (SPEAKER JOE)
                    (TIME #(TIME 4 4 5 5)))
                ?= DIRECT-REQUEST361
                              ((SPEAKER JOE)
                              (HEARER SALLY)
                              (REQGOAL (INFORMEDIF SALLY JOE
                                        (CAN SALLY (GAVE SALLY JOE SALT))))
                              (TIME #(TIME 4 4 5 5)))
                    ?= DIRECT-REQUEST360
                    ((SPEAKER JOE)
                    (HEARER SALLY)
                    (REQGOAL (INFORMEDIF SALLY JOE
                         (CAN SALLY (GAVE SALLY JOE SALT))))
                    (TIME #(TIME 4 4 5 5)))
              S -> TALK1
                    ((CONTENT (CAN SALLY
                              (GAVE SALLY JOE SALT)))
                    (HEARER SALLY)
                    (SPEAKER JOE)
                    (TIME #(TIME 4 4 5 5)))

;; There are 2 (of 4) alternatives not eliminated:  an attempt to find out if Sally
;;  can give Joe the salt, or an attempt to obtain the salt.

;; Now lets add the fact that at all times, Joe knows if  Sally can give him the salt.
```

The implementation did not include an explicit Holds predicate; instead, it included a fuzzy time constraint as the second argument to fact predicates. The following assertion means

$\forall\, t \in (-\infty\ \ -\infty\ +\infty\ +\infty)$.
Holds(t, (knowif Joe (can Sally (gave Sally Joe Salt))))

```
Command: (ADD-FACT '(KNOWIF #(TIME :-INF :-INF :+INF :+INF) JOE
            (CAN SALLY (GAVE SALLY JOE SALT)))
 T

;; Repeat the example.  Joe says to Sally: "Can you give me the salt?"

Command: (EXPLAIN-OBSERVATION '(TALK2 SURFACE-QUESTION
                  (SPEAKER JOE) (HEARER SALLY)
                  (CONTENT (CAN SALLY (GAVE SALLY JOE SALT))))
            'UTTER2)
 ... creating node TALK2
... searching up from TALK2
... creating node DIRECT-REQUEST375
... considering TALK2 as S of DIRECT-REQUEST375
... checking constraints on DIRECT-REQUEST375
... DIRECT-REQUEST375 is okay
... searching up from DIRECT-REQUEST375
... creating node DIRECT-REQUEST376
... searching up from DIRECT-REQUEST376
... creating node REQUEST377
... searching up from REQUEST377
... creating node OBTAIN-BY-ASKING378
... considering REQUEST377 as S1 of OBTAIN-BY-ASKING378
... checking constraints on OBTAIN-BY-ASKING378
... equality constraint violated GAVE = INFORMEDIF
... OBTAIN-BY-ASKING378 fails
... OBTAIN-BY-ASKING378 fails
... creating node FINDOUT-BY-ASKING379
... considering REQUEST377 as S1 of FINDOUT-BY-ASKING379
... checking constraints on FINDOUT-BY-ASKING379
... fact constraint violated (NEVER #(TIME -INF 4 -INF +INF) KNOWIF JOE
(CAN SALLY (GAVE SALLY JOE SALT)))
... FINDOUT-BY-ASKING379 fails
... REQUEST377 fails
... DIRECT-REQUEST376 fails
... DIRECT-REQUEST375 fails
... creating node INDIRECT-REQUEST380
... considering TALK2 as S of INDIRECT-REQUEST380
... checking constraints on INDIRECT-REQUEST380
... INDIRECT-REQUEST380 is okay
... searching up from INDIRECT-REQUEST380
... creating node INDIRECT-REQUEST381
... searching up from INDIRECT-REQUEST381
... creating node REQUEST382
... searching up from REQUEST382
... creating node OBTAIN-BY-ASKING383
... considering REQUEST382 as S1 of OBTAIN-BY-ASKING383
... checking constraints on OBTAIN-BY-ASKING383
... OBTAIN-BY-ASKING383 is okay
... searching up from OBTAIN-BY-ASKING383
... creating node OBTAIN-BY-ASKING384
... searching up from OBTAIN-BY-ASKING384
... creating node OBTAIN385
... searching up from OBTAIN385
... creating node END386
... searching up from END386
... creating node FINDOUT-BY-ASKING387
... considering REQUEST382 as S1 of FINDOUT-BY-ASKING387
... checking constraints on FINDOUT-BY-ASKING387
... equality constraint violated INFORMEDIF = GAVE
... FINDOUT-BY-ASKING387 fails
... FINDOUT-BY-ASKING387 fails
 T

;; The only interpretation is the indirect request.

Command: (DRAW-GRAPH T)
graph UTTER2
    ?= OBTAIN385
                ((OBJECT SALT)
                (AGENT JOE)
                (TIME #(TIME -INF 4 5 +INF)))
                (PRETIME #(TIME -INF 4 -INF +INF))
        ?= OBTAIN-BY-ASKING384
                        ((AGENT JOE) (OBJECT SALT)
                        (TIME #(TIME -INF 4 -INF +INF))
                        (PRETIME #(TIME -INF 4 -INF +INF)))
            ?= OBTAIN-BY-ASKING383
                        ((AGENT JOE) (OBJECT SALT)
          (TIME #(TIME -INF 4 -INF +INF))
          (PRETIME #(TIME -INF 4 -INF +INF)))
S1 -> REQUEST382
                  ((REQGOAL (GAVE SALLY JOE SALT))
                  (HEARER SALLY)
                  (SPEAKER JOE)
                  (TIME #(TIME 4 4 5 5)))
   ?= INDIRECT-REQUEST381
                  ((SPEAKER JOE) (HEARER SALLY)
                  (REQGOAL (GAVE SALLY JOE SALT))
                  (TIME #(TIME 4 4 5 5)))
      ?= INDIRECT-REQUEST380
                  ((SPEAKER JOE)
                  (HEARER SALLY)
                  (REQGOAL (GAVE SALLY JOE SALT))
                  (TIME #(TIME 4 4 5 5)))
         S -> TALK2
                  ((CONTENT (CAN SALLY
                                  (GAVE SALLY JOE SALT)))
                  (HEARER SALLY)
                  (SPEAKER JOE)
                  (TIME #(TIME 4 4 5 5)))
```

Appendix F Details of Correctness Proof of **Explain**

The key to the correctness of **explain** – the fact that *all* possible interpretations are considered – lies in the interaction of the **redundant** subroutine and the definition of the set Uses. **Redundant** blocks consideration of uses of an event which are covered by other, more apt uses. For example, if the primary event to be explained is of type MakeMarinara, the use (MakeSauce, s2, MakePastaDish) is redundant, while if the primary event to be explained is of type MakeSauce, the use (MakeMarinara, s2, MakeSpaghettiMarinara) is redundant.

Following is the statement which must be shown to hold in order for **explain** to be correct.

Statement of the Problem

Consider the invocation

explain(E_0, D, ∅, true, E_0)

For every (event type) E_u such that E_u has an r-direct component of type E_c which is compatible with E_0, and for every basic specialization* E_b of E_u whose instances could possibly have an r-component of E_0:

> the algorithm creates a node $E_a(n_a)$ such that E_a abstracts* E_b, whose r-component $E_a(n_a)$ is a node of type compatible with E_0.

Multiple Inheritance Proof

By inspection we see that **explain** does create a node $E_c(n_c)$, which holds one of the following relations to $E_0(n_0)$.

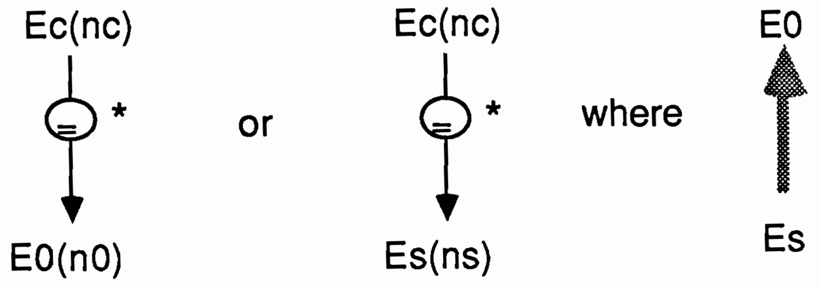


figure F.1: Relation of $E_c(n_c)$ to $E_0(n_0)$

If the component/use inference step performs a recursive call for use (E_c, r, E_u), then the condition is satisfied by $E_u(n_u)$ itself. If this doesn't occur, then the **redundant** test must have blocked the use. So consider the ways in which **redundant** could be satisfied.

Case 1. (E_c, r, E_u) abstracts a use for a type which abstracts* **primary**. There must be a least abstract such type, call it E_1, with use (E_1, r, E_d). This is illustrated below.

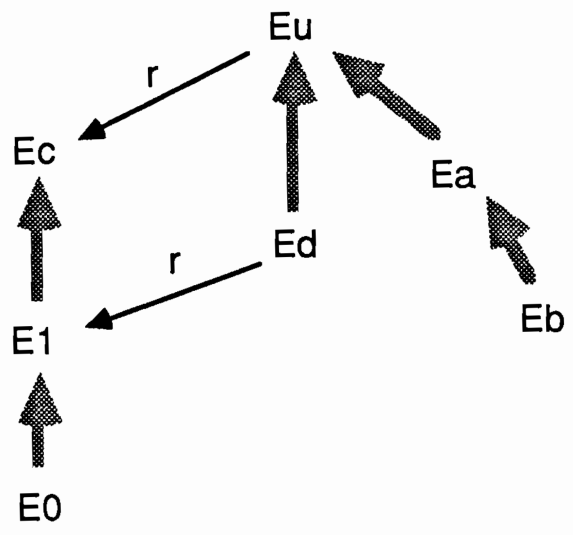


figure F.2: Case 1

When **explain** visited $E_1(n_1)$ it must have considered all r-uses of E_1, since by inspection **redundant** could not block them. Now we claim that there are r-uses which fulfil the condition.

Let E_b be the specialization of E_u in question. Consider the generation of Uses. Let E_a be the most general abstraction* of E_b such that:

i. All specializations* of E_a could have an r-th component of type E_1

ii. $(E_1, r, E_a) \in U_\alpha$

By definition of E_a, (E_1, r, E_a) appears in Uses. Thus node $E_a(n_a)$, with roleval (r, n_1), is exactly the node needed for the proof.

Case 2. (E_c, r, E_u) specializes a use for a type which specializes* **primary**. There must be a most general such type, call it E_1, with use (E_1, r, E_a). This is illustrated below.

$$\begin{tikzcd}
E0 & \\
 & Ea \arrow[dl, "r"'] \\
E1 \arrow[uu] & \\
 & Eu \arrow[uu] \arrow[dl, "r"'] \\
Ec \arrow[uu] & \\
 & Eb \arrow[uu]
\end{tikzcd}$$

figure F.3: Case 2

By inspection we see that the **redundant** test cannot block this use, so node $E_a(n_a)$ is the desired node, because E_u abstracts* E_b and E_a abstracts* E_u.

Case 3. (E_c, r, E_u) specializes a use for a type which abstracts E_0 and E_c does not abstract* or specialize* E_0. There must be a least general such type, call it E_1, with use (E_1, r, E_d). Let E_2 be the least general abstraction* of E_0 which has a use (E_2, r, E_f) which specializes* (E_1, r, E_d). (Possibly $E_2=E_1$.) This is illustrated below.

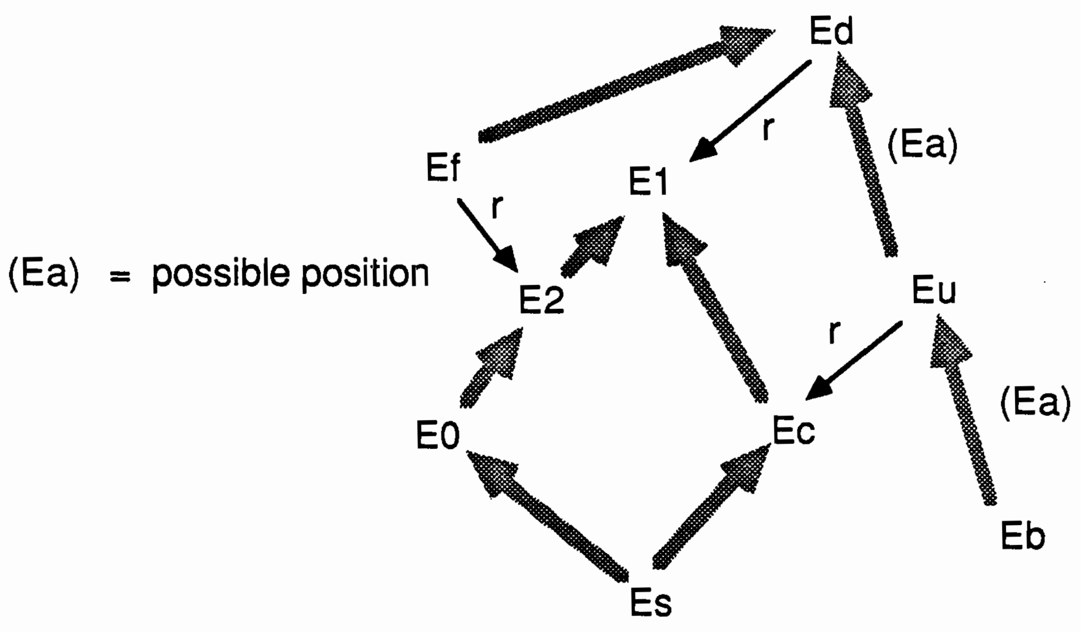


figure F.4: Case 3

Let E_a be the most general abstraction* of E_b such that

i. All specializations* of E_a could possibly have an r-th component of type E_2.

ii. $(E_2, r, E_a) \in U_\alpha$

By definition of E_a, (E_2, r, E_a) appears in Uses. By inspection we see that this use cannot be blocked by **redundant**, so $E_a(n_a)$, with roleval (r, n_2), is the desired node.

Single Inheritance Proof

Consider the single-inheritance version of **redundant**. There are two conditions under which **redundant** returns true:

Case 1. (etype, r, utype) abstracts a use for some member of **visited**. Because abstractions of **primary** are considered before specializations of **primary**, it must be the case that (etype, r, utype) abstracts a use for a type which abstracts* **primary**: that is, the first case above.

Case 2. (etype, r, utype) specializes a use for some member of **visited**. Because **visited** is grown depth first, it must be the case that (etype, r, utype) specializes a use for a type which specializes* **primary**: that is, the second case above.

This completes the details of the proof of correctness of **explain.**