A Formal Theory of Plan Recognition and its Implementation

Henry A. Kautz

Author's address: AT&T Labs, Room A-257, 180 Park Ave., Florham Park, NJ 07932. Telephone: (973) 360-8310. E-mail: kautz@research.att.com.

Published in *Reasoning About Plans,* by J.F. Allen, H.A. Kautz, R.N. Pelavin, and J.D. Tenenberg, Morgan Kaufmann Publishers, San Mateo, CA, 1991, pages 69-126 (chapter 2).

1. Introduction

1.1. Background

While there have been many formal studies of plan synthesis in general [McCarthy & Hayes 1969, Moore 1977, Pednault 1988, Tenenberg 1990, Pelavin 1990], nearly all work on the inverse problem of plan recognition has focused on specific kinds of recognition in specific domains. This includes work on story understanding [Bruce 1981, Schank 1975, Wilensky 1983], psychological modelling [Schmidt 1978], natural language pragmatics [Allen 1983, Carberry 1983, Litman 1987, Grosz & Sidner 1987], and intelligent computer system interfaces [Genesereth 1979, Huff & Lesser 1982, Goodman & Litman 1990]. In each case, the recognizer is given an impoverished and fragmented description of the actions performed by one or more agents and expected to infer a rich and highly interrelated description. The new description fills out details of the setting and predicts the goals and future actions of the agents. Plan recognition can be used to generate summaries, to provide help, and to build up a context for use in disambiguating natural language. This chapter develops a formal theory of plan recognition. The analysis provides a formal foundation for part of what is loosely called "frame–based inference" [Minsky 1975], and accounts for problems of ambiguity, abstraction, and complex temporal interactions that were ignored by previous research.

Plan recognition problems can be classified as cases of either “intended” or “keyhole” recognition (the terminology developed by [Cohen, Perrault, & Allen 1981]). In the first kind but not the second the recognizer 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 1986]. This chapter concentrates on keyhole recognition of correct plans where the observer has complete knowledge.

Plan synthesis can be viewed as a kind of hypothetical reasoning, where the planner tries to find some set of actions whose execution would entail some goal. Some previous work on plan recognition views it as a similar kind of hypothetical reasoning, where the recognizer tries to find some plan whose execution would entail the performance of the observed actions [Charniak 1985]. This kind of reasoning is sometimes called “abduction”, and the conclusions “explanations”. But it is not clear what the recognizer should conclude if many different plans entail the observations. Furthermore, even if the recognizer has complete knowledge and the agent makes no errors cases naturally occur where no plan actually entails the observations. For example, suppose that the recognizer knows about a plan to get food by buying it at a supermarket. The recognizer is told that the agent walks to the A&P on Franklin Street. The plan to get food does not entail this observation; it entails going to some supermarket, but not the A&P in particular. One can try to fix this problem by giving it a more general plan schema that can be instantiated for any particular supermarket. But the entailment still fails, because the plan still fails to account for the fact that the agent chooses to walk instead of driving. Instead of finding a plan that entails the observations, one can only find a plan that entails some weaker statement entailed by the observations. In order to make abduction work, therefore, the plan (or explanation) must be able to also include almost any kind of assumption (for example, that the agent is walking); yet the assumptions should not be strong as to trivially imply the observations. The abductive system described in [Hobbs & Stickel 88] implements this approach by assigning costs to various kinds of assumptions, and searching for an explanation of minimum cost. The problems of automatically generating cost assignments, and of providing a theoretical basis for combining costs, remain open.

Other approaches to plan recognition describe it as the result of applying unsound rules of inference that are created by reversing normally sound implications. From the fact that a particular plan entails a particular action, one derives the unsound rule that that action “may” imply that plan [Allen 1983]. Such unsound rules, however, generate staggering numbers of possible plans. The key problems of deciding which rules to apply and when to stop applying the rules remain outside the formal theory.

By contrast, the framework presented in this chapter specifies what conclusions are absolutely justified on the basis of the observations, the recognizer's knowledge, and a number of explicit "closed world" assumptions. The conclusions follow by ordinary deduction from these statements. If many plans could explain the observations, then the recognizer is able to conclude whatever is common to all the simplest such plans. The technical achievement of this work is the ability to specify the assumptions the recognizer makes without recourse to a control mechanism lying outside the theory.

Another natural way to view plan recognition is as a kind of probabilistic reasoning [Charniak & Goldman 1989]. The conclusions of the recognizer are simply those statements that are assigned a high probability in light of the evidence. A probabilistic approach is similar to the approach taken in this chapter in that reasoning proceeds directly from the observations to the conclusions and avoids the problems described above with the construction of explanations. The closed world assumptions employed by our system correspond to closure assumptions implicitly made in a Bayesian analysis, where the set of possible hypotheses is assumed to be disjoint and exhaustive. A major strength of the probabilistic approach over ours is that it allows one to capture the fact that certain plans are a priori more likely than others. While much progress is being made in mechanizing propositional probabilistic reasoning, first-order probabilistic reasoning is much more difficult. A propositional system can include a data element representing every possible plan and observation, and the effect of the change in probability of any element on every other element can be computed. This is not always possible in a first-order system, where the language can describe an infinite number of plans. The problem is not just one of selecting between hypotheses, but also selectively instantiating the first-order axioms that describe the parameterized plans. In our purely logical theory one can simply deduce the parameters of the plans that are recognized.

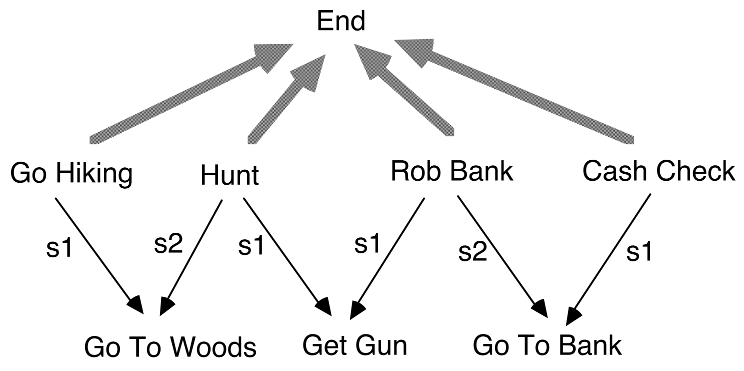
1.2. Overview

In this chapter plans and actions are uniformly referred to as *events*. The recognizer's knowledge is represented by a set of first-order statements called an *event hierarchy*, which defines the abstraction, specialization, and functional relationships between various types of events. The functional, or "role", relationships include the relation of an event to its *component* events. There is a distinguished type, End, which holds of events that are not components of any other events. Recognition is the problem of describing the End events that generate a set of observed events.[1]

In this work we are limited to recognizing instances of plans whose types appear in the hierarchy; we do not try to recognize new plans created by chaining together the preconditions and effects of other plans (as is done in [Allen 1983]). Therefore it is appropriate for domains where one can enumerate in advance all the ways of achieving a goal; in other words, where one wants to recognize stereotypical behavior, rather than understand truly unique and idiosyncratic behavior. This assumption of complete knowledge on the part of the system designer is fundamental to the approach. While abandoning this assumption might increase a system's flexibility, it would also lead to massive increase in the size of the search space, since an infinite number of plans could be constructed by chaining on preconditions and effects. We decided to maintain the assumption of complete knowledge, and only construct plans by specialization and decomposition (as described below), until we have developed methods of controlling the combinatorial problem.

An event hierarchy does not by itself justify inferences from observations to End events. Consider the following set of plans. There are four kinds of End events (hiking, hunting, robbing banks, and cashing checks) and three other kinds of events (going to the woods, getting a gun, and going to a bank). The event hierarchy can be illustrated by the following network, where the thick grey arrows denote abstraction or "is a", and the thin black arrows denote component or "has part". The labels "s1" and "s2" serve to distinguish the component arcs; they denote the functions which map an event to the respective component. (The labels do not, by themselves, formally indicate the temporal ordering of the components.)

[1] The idea of an End event may be problematic in general; see our comments at the end of this chapter.



We encode this event hierarchy in first-order logic with the following axioms.

$$\begin{array}{l}
\forall\ x\ .\ \text{GoHiking}(x) \supset \text{End}(x) \\
\forall\ x\ .\ \text{Hunt}(x) \supset \text{End}(x) \\
\forall\ x\ .\ \text{RobBank}(x) \supset \text{End}(x) \\
\forall\ x\ .\ \text{CashCheck}(x) \supset \text{End}(x) \\
\forall\ x\ .\ \text{GoHiking}(x) \supset \text{GoToWoods}(s1(x)) \\
\forall\ x\ .\ \text{Hunt}(x) \supset \text{GetGun}(s1(x)) \wedge \text{GoToWoods}(s2(x)) \\
\forall\ x\ .\ \text{RobBank}(x) \supset \text{GetGun}(s1(x)) \wedge \text{GoToBank}(s2(x)) \\
\forall\ x\ .\ \text{CashCheck}(x) \supset \text{GoToBank}(s1(x))
\end{array}$$

The symbols "s1" and "s2" are functions which map a plan to its steps. Suppose GetGun(C) is observed. This statement together with the axioms does not entail $\exists$x.Hunt(x), or $\exists$x.[Hunt(x) $\vee$ RobBank(x)], or even $\exists$x.End(x). The axioms let one infer that getting a gun is implied by hunting or going to the bank, but not vice versa.

It would not help to strengthen the implications to biconditionals in the last four axioms above, in order to make them state sufficient as well as necessary conditions for the execution of the End events. Even if every single step of a plan were observed, one could not deduce that the plan occurred. For example, suppose that the recognizer learns {GetGun(C), GoToBank(D)}, and believes

$$\forall\ x\ .\ \text{RobBank}(x) \equiv \text{GetGun}(s1(x)) \wedge \text{GoToBank}(s2(x))$$

The statement $\exists$x.RobBank(x) still does not follow, because of the missing premise that $\exists x.C=s1(x) \wedge D=s2(x)$. One could further strengthen the axiom to say that whenever someone gets a gun and goes to a bank he or she commits a robbery:

$$[\exists\ x\ .\ \text{RobBank}(x)] \equiv [\exists\ y, z\ .\ \text{GetGun}(y) \wedge \text{GoToBank}(z)]$$

This change allows the recognizer to conclude $\exists x.\mathrm{RobBank}(x)$, but does not really solve the problem. First, one cannot always give sufficient conditions for every kind of event. Second, the recognizer is still required to observe every step of the plan to be recognized. This latter condition is rarely the case; indeed, a primary motivation for plan recognition is the desire to *predict* the actions of the agent. Finally, such an axiom doesn't allow for a case where a person cashes a check on his way to a hunting trip.

Nonetheless it does seem reasonable to conclude that someone is either hunting or robbing a bank on the basis of GetGun(C). 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. We will show how to generate a set of completeness or closed-world assumptions for a given hierarchy. The assumption needed for this example is simply

$$\forall x . \mathrm{GetGun}(x) \supset [\exists y . \mathrm{Hunt}(y) \wedge x=\mathrm{s1}(y)] \vee [\exists y . \mathrm{RobBank}(y) \wedge x=\mathrm{s1}(y)]$$

We will also show that making these assumptions is equivalent to *circumscribing the event hierarchy* in a particular way [McCarthy 1980]. We borrow the model theory of circumscription to provide a model theory for plan recognition. Whatever deductively follows from the observations, event hierarchy, and assumptions holds in all "covering models" of the observations and event hierarchy. (The term "covering model" comes from the fact that every event in such a model is explained or "covered" by some End event which contains it.) In this example, the only covering models are isomorphic to (or contain a submodel isomorphic to) one of the two models

$$\{\mathrm{End}(A), \mathrm{Hunt}(A), \mathrm{GetGun}(\mathrm{s1}(A)), \mathrm{GoToWoods}(\mathrm{s2}(A))\}$$

$$\{\mathrm{End}(A), \mathrm{RobBank}(A), \mathrm{GetGun}(\mathrm{s1}(A)), \mathrm{GoToBank}(\mathrm{s2}(A))\}$$

Any model containing just an instance of GetGun but no corresponding End event is not a covering model.

When several events are observed, additional assumptions are needed. Suppose that {GetGun(C), GoToBank(D)} is observed. These formulas together with the assumptions

$$\forall x . \mathrm{GetGun}(x) \supset [\exists y . \mathrm{Hunt}(y) \wedge x=\mathrm{s1}(y)] \vee [\exists y . \mathrm{RobBank}(y) \wedge x=\mathrm{s1}(y)]$$

$$\forall x . \mathrm{GoToBank}(x) \supset [\exists y . \mathrm{CashCheck}(y) \wedge x=\mathrm{s1}(y)] \vee [\exists y . \mathrm{RobBank}(y) \wedge x=\mathrm{s2}(y)]$$

do not entail that an instance of bank robbery occurs. The first observation could be a step of a plan to hunt and the second could be a step of a plan to cash a check. Yet in this example the RobBank plan is simpler than the conjunction of two other unrelated plans. It is reasonable to assume that unless there is reason to believe otherwise, all the observations are part of the same End event. Given the assumption

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

the conclusion $\exists x.\mathrm{RobBank}(x)$ deductively follows.[2] In model theoretic terms, this assumption corresponds to selecting out the covering models that contain a minimum number of End events: these are the *minimum covering models* of the observations and hierarchy. Furthermore, this assumption can be blocked if necessary. If it is known that the agent is not robbing the bank, that is, if the input is $\{\mathrm{GetGun}(C), \mathrm{GoToBank}(D), \neg\exists\, y \,.\, \mathrm{RobBank}(y)\}$, then the strongest simplicity assumption is that there are two distinct unrelated plans. It follows that these plans are hunting and cashing a check.

Why care about a formal theory of plan recognition? One advantage of this approach is that the proof and model theories apply to almost any situation. They handle disjunctive information, concurrent plans, steps shared between plans, and abstract event descriptions. We will illustrate the theory with examples of plan recognition from the domains of cooking and operating systems. The general nature of the theory suggests that it can be applied to problems other than plan recognition. We will show how a medical diagnosis problem can be represented in our system, by taking events to be diseases and symptoms rather than plans and actions. The similarity between the kind of reasoning that goes on in plan recognition and medical diagnosis has been noted by Charniak [1983]. Reggia, Nau, and Wang [1983] have proposed that medical diagnosis 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. They work within a purely propositional logic and do not include an abstraction hierarchy. Extending their formal framework to first-order would make it quite close to the one presented here.

The formal theory is independent of any particular implementation or algorithm. It specifies the goal of the computation and provides an abstract mapping from the input

2 (The careful reader may note that the assumption $\forall x.\neg\mathrm{Hunt}(x)\vee\neg\mathrm{RobBank}(x)$ is also needed to make the proof goes through; this kind of assumption will also be developed below.)

information to the output. The last section of this chapter provides specific algorithms for plan recognition. The algorithms implement the formal theory, but are incomplete; they are, however, much more efficient than a complete implementation which simply used a general-purpose theorem prover. While the proof theory specifies a potentially infinite set of justified conclusions, the algorithms specify which conclusions are explicitly computed. The algorithms use a compact graph-based representation of logical formulas containing both conjunctions and disjunctions. Logical operations (such as substitution of equals) are performed by graph operations (such as graph matching). The algorithms use a temporal representation which is related to but different from that discussed in Part 1. The times of specific instances of events are represented by numeric bounds on the starting and ending instants. This metric information is constrained by symbolic constraints recorded in the interval algebra.

1.3. Plan Recognition and the Frame Problem

Although the work described in this chapter is the first to suggest that closed world reasoning and in particular circumscription are relevant to plan recognition, the insight behind the connection is implicit in work on the “frame problem”. Given an axiomatization of the changes actions make on the world, one wants to generate axioms that describe the properties that are *not* changed by each action. For example, in the situation calculus the frame axiom that states that the color (c) of an object (o) is not changed by picking it up is often written as follows:

$$\forall\ s, c, o\ .\ \mathrm{Color}(o,c,s) \supset \mathrm{Color}(o, c, \mathrm{result}(\ \mathrm{move}(\ \mathrm{pickup}(o)\ ), s))$$

(In this particular representation, the last argument to a fluent such as “Color” is the state (s) in which the fluent holds. The function “result” maps an action (pickup(o)) and a state (s) to the resulting state.) One of the primary motivations for the development of circumscription was to be a formal tool for specifying such frame axioms.

Several researchers have observed that there is another way of writing frame axioms [Haas 1987, Schubert 1989, Pelavin 1990 (this volume, section 4.6)]. This is to state that if a particular property did change when any action was performed, then that action must be one of the actions known to change that property. In this example, suppose painting and burning are the only actions known to change the color of an object. The frame axiom for Color then becomes:

$$\forall\ s, c, o, a\ .\ [\ \mathrm{Color}(o,c,s) \wedge \neg\mathrm{Color}(o,c,\mathrm{result}(a,s))\ ] \supset [\ a{=}\mathrm{paint}(o) \vee a{=}\mathrm{burn}(o)\ ]$$

This frame axiom looks very much like the assumptions that are needed for plan recognition. The frame axioms lead from the premise that a change occurred to the disjunction of all the actions that could make that change. The recognition assumptions lead from the premise that an action occurred to the disjunction of all plans that could contain that action as a substep.

Many difficult technical problems have arisen in applying circumscription to the frame problem. Apparently obvious ways of using circumscription can lead to conclusions that are much weaker than desirable [Hanks & McDermott 1986], and the formalism is in general unwieldy. This chapter shows how circumscription can be successfully and efficiently applied to a knowledge base of a particular kind to generate conclusions of a particular form. Like all formalisms, circumscription is of interest only if it is of use; and we hope that the use it has found in the present work is of encouragement to those continuing to work on understanding and extending circumscription.

2. Representing Event Hierarchies

2.1. The Language

As described in Part 1, the representation language we will use is first-order predicate calculus with equality. We make the following extension to the notation: a prefix $\wedge$ (similarly $\vee$) applied to a set of formulas stands for the conjunction (similarly disjunction) of all the formulas in that set. We will first introduce a standard semantics for this language and then extend it to deal with the plan recognition problem. A model interprets 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 is made explicit by applying M to a term, function, or predicate. For example, for any model M:

$$\text{Loves(Sister(Joe),Bill) is true in M if and only if} \\ \langle M[\text{Sister}](M[\text{Joe}]), M[\text{Bill}]\rangle \in M[\text{Loves}]$$

The domain of discourse of the model M is written Domain(M). The fact that M interprets the constant "Joe" as an individual in its domain is written

$$M[\text{Joe}] \in \text{Domain}(M)$$

Meta-variables (not part of the language) that stand for domain individuals begin with a colon. Models map free variables in sentences to individuals. The expression M{x/:C} means the model that 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. Proofs in this chapter use natural deduction, freely appealing to obvious lemmas and transformations. It is convenient to distinguish a set of constant symbols called *Skolem constants* for use in the deductive rule of existential elimination. The rule allows one to replace an existentially-quantified variable by a Skolem constant that appears at no earlier point in the proof. Skolem constants are distinguished by the prefix "*". No Skolem constants 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.

2.2. Representation of Time, Properties, and Events

Most formal work on representing action has relied on versions of the situation calculus [McCarthy & Hayes 1969]. This formalism is awkward for plan recognition: convolutions are needed to state that some particular action *actually occurred* at a particular time (but see see [Cohen 1984]). We therefore adopt the "reified" representation of time and events described in detail in Chapter 1.

Recall from Chapter 1 that time is linear, and time *intervals* are individuals, each pair related by one of Allen's interval algebra relations: *Before, Meets, Overlaps*, etc. The names of several relations may be written in place of a predicate, in order to stand for the disjunction of those relations. For example, BeforeMeets(T1,T2) abbreviates Before(T1,T2) $\vee$ Meets(T1,T2). Intervals can be identified with pairs of rational numbers on some universal clock; two intervals Meet when the first point of one is the same as the last point of the other [Ladkin & Maddux 1988].

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

$$\text{ReadBook(C)} \wedge \text{object(C)=WarAndPeace} \wedge \text{time(C)=T2}$$

may be used to used to represent the fact that an instance of booking reading occurs; the book read is *War and Peace*; and the time of the reading is (the interval) T2. Role functions are also used to represent the steps of plans (or any other kind of structured event). For example, suppose that reading a book is a kind of plan, one of whose steps is to pick up the book. The following formula could be used to represent the fact that two events have occurred, where one is reading a book, and the other is the substep of picking up the book.

$$\text{ReadBook(C)} \wedge \text{pickupStep(C)=D} \wedge \text{Pickup(D)}$$

All other facts are represented by ordinary predicates. For example, the fact that John is a human may be represented by the formula Human(John). Circumstances that change over time are also represented by predicates whose last argument is a time interval. For example, the fact that John is unhappy over the interval T1 will be represented by the formula Unhappy(John, T1)

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 [Hayes 1985]. These axioms represent the abstraction and decomposition relations between event types. This section defines the formal parts of an event hierarchy, and the next provides a detailed example. 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. Any member of the extension of an event predicate is called an *event token*. AnyEvent is the most general event type, and End is the type of all events which are not part of some larger event.

• 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. The inverse of abstraction is specialization.

•H_{EB} is the set of basic event type predicates, those members of H_E that 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. Neither End nor any of its specializations may appear as a direct component.

•H_G is the set of general axioms, those that do not contain any member of H_E. H_G includes the axioms for the temporal interval relations, as well as any other facts not specifically relating to events.

Two event types are *compatible* if there is an event type they both abstract or are equal to.

The *parameters* of an event token are the values of those role functions mentioned in a decomposition axiom for any type of that token applied to the token.

The direct component relation may be applied to event tokens in a model *M* as follows. Suppose :Ci 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) $:C_i = M[f_i](:C_0)$

In other words, one token is a direct component of another just in case it is the value of one of the role functions applicable to the former token. 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 $:C_0$. The component relation over event tokens does not correspond to the transitive closure of the direct-component meta-relation over event types because a token may be of more than one type.

A hierarchy is *acyclic* if and only if it contains no series of event predicates $E_1, E_2, \ldots, E_n$ of odd length (greater than 1) such that:

(i) E_i is compatible with E_{i+1} for odd i, $1 \leq j \leq n-2$

(ii) E_{i-1} is a direct component of E_i for odd i, $3 \le i \le n$

(iii) $E_n = E_1$

Roughly speaking, an event hierarchy is acyclic if no event token may have a component of its own type. The definition above allows for the fact that all events are of type *AnyEvent*, and therefore any event token will share at least the type *AnyEvent* with its components. This chapter only considers acyclic event hierarchies.

2.4. Example: The Cooking World

The actions involved in cooking form a simple but interesting domain for planning and plan recognition. The specialization relations between various kinds of foods are mirrored by specialization relations between the actions that 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, the cook knows certain actions that are needed to create any cream-based sauce, as well as certain conditions (constraints) that 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; for example, one should slowly stir in grated cheese at a certain point in the recipe.

We are assuming that the cook and the observer have the same knowledge of cooking, a hierarchically arranged cookbook. 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. 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.

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. 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.

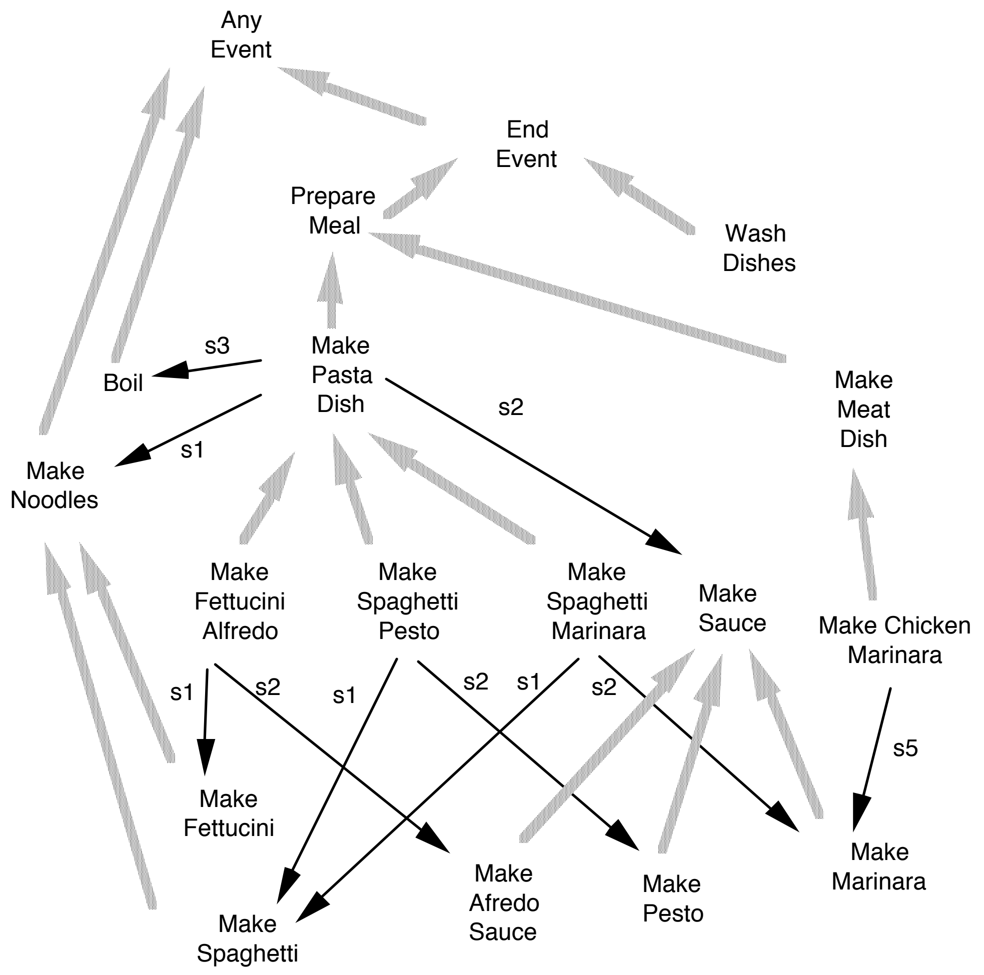


figure 1: Cooking event hierarchy. (The abstraction arc from MakeSauce to AnyEvent is omitted for clarity.)

The diagram illustrates some, but not all, of the information in the axioms for the cooking domain. The formal description of the hierarchy is as follows.

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

•The abstraction axioms, H_A, relate event types to their abstractions. For instance, *MakeNoodles* is an abstraction of both *MakeSpaghetti* and *MakeFettucini*. A traditional planning system might call *MakeSpaghetti* and *MakeFettucini* different *bodies* of the

MakeNoodles plan. This relationship is represented by asserting that every instance of the more specialized type is also an instance of the more abstract type. For example,

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

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

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

•The decomposition axioms, H_D, include information that does not appear in the diagram. Following is an incomplete 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;[3] 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.

[3] It is not necessarily the case that the agent of an event be the same as the agent of each of its components. One could as easily write constraints that they be different. For example, a plan for a cook could include steps to be carried out by the cook's helper.

$\forall x$. *MakePastaDish(x)* $\supset$

	MakeNoodles(step1(x)) $\wedge$
Components	*MakeSauce(step2(x))* $\wedge$
	Boil(step3(x)) $\wedge$
Equality	*agent(step1(x)) = agent(x)* $\wedge$
Constraints	*result(step1(x)) = input(step3(x))* $\wedge$
Temporal	*During(time(step1(x)), time(x))* $\wedge$
Constraints	*BeforeMeets(time(step1(x)), time(step3(x)))* $\wedge$
	Overlaps(time(x), postTime(x)) $\wedge$
Preconditions	*InKitchen(agent(x), time(x))* $\wedge$
	Dexterous(agent(x)) $\wedge$
Effects	*ReadyToEat(result(x), postTime(x))* $\wedge$
	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 that 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.

$\forall$x . MakeSpaghettiMarinara(x) $\supset$
MakeSpaghetti(step1(x)) $\wedge$
MakeMarinaraSauce(step2(x)) $\wedge$ …

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, “Yesterday Joe made the noodles on the table ” may be represented as follows:

MakeNoodle(Make33) $\wedge$
agent(Make33) = Joe $\wedge$
result(Make33) = Noodles72 $\wedge$
OnTable(Noodles72, Tnow) $\wedge$
During(time(Make33), Tyesterday)

3. The Formal Theory of Recognition

We have seen that the kind of inferences performed in plan recognition do not follow from an event hierarchy alone. 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 the hierarchy encodes all of our knowledge of events. If the hierarchy is enlarged, the assumptions must be revised. Finally, a simplicity assumption is used to combine information from several observations. We now consider the various kinds of assumptions in detail.

3.1. Exhaustiveness Assumptions (EXA)

Suppose you know that the agent is making some kind of sauce that is not Alfredo sauce and not pesto. Then you can reasonably conclude that the agent is making marinara sauce. Such a conclusion is justified by the assumption that the known ways of specializing an event type are the only ways of specializing it. In this case, the assumption is

$$\begin{aligned} &\forall x \,.\, \mathrm{MakeSauce}(x) \supset \\ &\qquad \mathrm{MakeMarinara}(x) \vee \\ &\qquad \mathrm{MakeAlfredoSauce}(x) \vee \\ &\qquad \mathrm{MakePesto}(x) \end{aligned}$$

Another way to write this same statement is

$$\begin{aligned} &\forall x \,.\, \mathrm{MakeSauce}(x) \wedge \neg \mathrm{MakeAlfredoSauce}(x) \wedge \neg \mathrm{MakePesto}(x) \supset \\ &\qquad \mathrm{MakeMarinara}(x) \end{aligned}$$

This kind of assumption allows one to determine that a particular kind of event has occurred by eliminating all other possibilities. Fans of Sherlock Holmes will recognize it as an instance of his dictum, “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” [Doyle 1890]

The set EXA of exhaustiveness assumptions are all statements of the following form, where E_0 is a predicate in H_A, and $\{E_1, E_2, \ldots, E_n\}$ are all the predicates directly abstracted by E_0 :

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

3.2. Disjointness Assumptions (DJA)

Suppose the agent is making a pasta dish. It is clear from the example hierarchy that this particular event is not an instance of making a meat dish. That is, we make the assumption that

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

Why is this only an assumption? Suppose a new type were added to the hierarchy that specialized both MakePastaDish and MakeMeatDish:

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

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

Then the assumption that meat dishes and pasta dishes are disjoint would no longer be reasonable. Assuming that one's knowledge of events is complete, however, it is reasonable to assume that two types are disjoint, unless one abstracts the other, or they abstract a common type; that is, if they are compatible. The disjointness assumptions together with the exhaustiveness assumptions entail that every event has a unique basic type, where the basic types are the leaves of the abstraction hierarchy.

The set DJA of disjointness assumptions consists of all statements of the following form, where event predicates E_1 and E_2 are not compatible:

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

3.3. Component/Use Assumptions (CUA)

The most important assumptions for recognition let one infer the disjunction of the possible "causes" for an event from its occurrence. They state that a plan or action implies the disjunction of the plans which use it as a component. The simplest case is when only a single type could have a particular event as a direct component. For instance, from the fact that the agent is boiling water one can conclude that the agent is making a pasta dish. Formally, the assumption is that

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

More generally, the conclusion of this kind of assumption is the disjunction of all events that have a component that is compatible with the premise. Consider the assumption for MakeSauce. This type is compatible with itself and all of its specializations: MakeAlfredoSauce, MakePesto, and MakeMarinara. The following formula describes all the events that could have a component of those types:

$$\begin{aligned}&\forall x \,.\, \mathrm{MakeSauce}(x) \supset \\ &\quad (\exists y \,.\, \mathrm{MakePastaDish}(y) \wedge x = \mathrm{step2}(y)) \vee \\ &\quad (\exists y \,.\, \mathrm{MakeFettuciniAlfredo}(y) \wedge x = \mathrm{step2}(y)) \vee \\ &\quad (\exists y \,.\, \mathrm{MakeSpaghettiPesto}(y) \wedge x = \mathrm{step2}(y)) \vee \\ &\quad (\exists y \,.\, \mathrm{MakeSpaghettiMarinara}(y) \wedge x = \mathrm{step2}(y)) \vee \\ &\quad (\exists y \,.\, \mathrm{MakeChickenMarinara}(y) \wedge x = \mathrm{step5}(y))\end{aligned}$$

Note that MakeChickenMarinara has MakeMarinara as a component, which is compatible with MakeSauce. The formula can be simplified by using the abstraction axioms for MakePastaDish.

$$\begin{aligned}&\forall x \,.\, \mathrm{MakeSauce}(x) \supset \\ &\quad (\exists y \,.\, \mathrm{MakePastaDish}(y) \wedge x = \mathrm{step2}(y)) \vee \\ &\quad (\exists y \,.\, \mathrm{MakeChickenMarinara}(y) \wedge x = \mathrm{step5}(y))\end{aligned}$$

This example demonstrates that making such a component/use assumption is not the same as predicate completion in the style of Clark [1978]. Predicate completion yields

$$\begin{aligned}&\forall x \,.\, \mathrm{MakeSauce}(x) \supset \\ &\quad (\exists y \,.\, \mathrm{MakePastaDish}(y) \wedge x = \mathrm{step2}(y))\end{aligned}$$

This assumption is too strong, because it omits the use of MakeSauce as specialized by MakeMarinara that is implicit in event hierarchy.

The definition of the set CUA of component/use assumptions follows. For any $E \in H_E$, define Com(E) as 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 formula is the component/use assumption for *E:*

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

CUA is the set of all such formulas for a given hierarchy. It is usually possible to remove redundant subexpressions from the right-hand side of these formulas, as in the example above. Throughout the rest of this chapter such simplifications will be made.

3.4. Minimum Cardinality Assumptions (MCA)

The assumptions described above do not combine information from several observations. Suppose that the agent is observed to be making spaghetti and making marinara sauce. The first observation is explained by applying the component/use assumption that the agent is making spaghetti marinara or making spaghetti pesto. The second observation is similarly explained by the conclusion that the agent is making spaghetti marinara or chicken marinara. The conclusion cannot be drawn, however, that the agent is making spaghetti marinara. The End event that explains the first observation could be distinct from the End event that explains the second. The theory as outlined so far sanctions only the statement

$$
\begin{aligned}
&\exists x \,.\, [MakeSpaghettiMarinara(x) \lor MakeSpaghettiPesto(x)] \land \\
&\exists y \,.\, [MakeSpaghettiMarinara(y) \lor MakeChickenMarinara(y)]
\end{aligned}
$$

In many cases it is reasonable to assume that the observations are related. A simple heuristic is to assume that there is a minimal number of distinct End events. In this example, all the types above are specializations of End. The statement above, the disjointness assumptions, and the assumption that there is no more than one End event entails the conclusion that the agent is making spaghetti marinara.

The three different kinds of assumptions discussed above are computed from the event hierarchy before any observations are made. The appropriate minimum cardinality assumption (MCA) is based on both the hierarchy and the specific observations that have been made. 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 the observations be represented by a set of formulas Γ. The minimum cardinality assumption appropriate for H and Γ is the formula MA_i, where i is the smallest integer such that

$$\Gamma \cup H \cup EXA \cup DJA \cup CUA \cup MA_i$$

is consistent. (This consistency test is, in general, undecidable; the algorithms described later in this chapter create separate data structures corresponding to the application of each assumption to the observations, and prune the data structures when an inconsistency is noticed. At any time the conclusions of the system are represented by the data structure corresponding to the strongest assumption.)

3.5. Example: The Cooking World

The following example shows how the different components of the event hierarchy and kinds of assumptions interact in plan recognition. As noted earlier, constants prefixed with a "*" stand in place of existentially-quantified variables. Suppose the observer initially knows that the agent will not be making Alfredo sauce. Such knowledge could come from information the observer has about the resources available to the agent; for example, a lack of cream. Further suppose that the initial observation is that the agent is making some kind of noodles.

Observation

[1] *MakeNoodles(Obs1)*

Component/Use Assumption [1] & Existential Instantiation

[2] *MakePastaDish(*I1)* $\wedge$ *step1(*I1)=Obs1*

Abstraction [2]

[3] *PrepareMeal(*I1)*

Abstraction [3]

[4] $End(*I1)$

Although the recognized plan is not fully specified, enough is known to allow the observer to make predictions about future actions of the agent. For example, the observer can predict that the agent will boil water:

Decomposition [2]

[5] $Boil(\ step3(*I1)\ ) \wedge After(time(Obs1), time(step3(*I1))\ )$

The observer may choose to make further inferences to refine the hypothesis. The single formula "MakePastaDish(*I1)" above does not summarize all the information gained by plan recognition. The actual set of conclusions is always infinite, since it includes all formulas that are entailed by the hierarchy, the observations, and the assumptions. (The algorithms discussed later in this chapter perform a limited number of inferences and generate a finite set of conclusions.) Several inference steps are required to reach the conclusion that the agent must be making spaghetti rather than fettucini.

Given Knowledge

[6] $\forall x\ .\ \neg MakeAlfredoSauce(x)$

Exhaustiveness Assumption [2]

[7] $MakeSpaghettiMarinara(*I1) \vee MakeSpaghettiPesto(*I1)$
$\vee MakeFettuciniAlfredo(*I1)$

Decomposition & Universal Instantiation

[8] $MakeFettuciniAlfredo(*I1) \supset MakeAlfredoSauce(step2(*I1))$

Modus Tollens [6,8]

[9] $\neg MakeFettuciniAlfredo(*I1)$

Disjunction Elimination [7,9]

[10] $MakeSpaghettiMarinara(*I1) \vee MakeSpaghettiPesto(*I1)$

Decomposition & Universal Instantiation

[11] $MakeSpaghettiMarinara(*I1) \supset\ MakeSpaghetti(step1(*I1))$

[12] $MakeSpaghettiPesto(*I1) \supset\ MakeSpaghetti(step1(*I1))$

Reasoning by Cases [10,11,12]

[13] $MakeSpaghetti(step1(*I1))$

Suppose that the second observation is that the agent is making marinara sauce. The minimal cardinality assumption allows the observer to intersect the possible explanations for the first observation with those for the second, in order to reach the conclusion that the agent is making spaghetti marinara.

Second Observation

[14] $MakeMarinara(Obs2)$

Component/Use Assumption [14] & Existential Instantiation

[15] $MakeSpaghettiMarinara(*I2) \vee MakeChickenMarinara(*I2)$

Abstraction [15]

[16] $MakePastaDish(*I2) \vee MakeMeatDish(*I2)$

Abstraction [16]

[17] $PrepareMeal(*I2)$

Abstraction [17]

[18] $End(*I2)$

Minimality Assumption

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

Universal Instantiation & Modus Ponens [4,17,19]

[20] $*I1 = *I2$

Substitution of Equals [2,30]

[21] $MakePastaDish(*I2)$

Disjointness Assumption

[22] $\forall x \,.\, \neg MakePastaDish(x) \vee \neg MakeMeatDish(x)$

Disjunction Elimination [21,22]

[23] $\neg MakeMeatDish(*I2)$

Abstraction & Existential Instantiation

[24] $MakeChickenMarinara(*I2) \supset MakeMeatDish(*I2)$

Modus Tollens [23,24]

[25] *¬MakeChickenMarinara(*I2)*

Disjunction Elimination [15,25]

[26] *MakeSpaghettiMarinara(*I2)*

3.6. Circumscription and Plan Recognition

Earlier we discussed the relation of circumscription to plan recognition in informal terms. Now we will make that relation precise, and in so doing, develop a model theory for part of the plan recognition framework.

Circumscription is a syntactic transformation of a set of sentences representing an agent's knowledge. Let $S[\pi]$ be a set of formulas containing a list of predicates π. The expression $S[\sigma]$ is the set of formulas 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,π), is the second-order formula

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

Circumscription has a simple and elegant model theory. Suppose M_1 and M_2 are models of S which are identical except that the extension in M_2 of one or more of the predicates in π is a proper subset of the extensions of those predicates in M_1. This is denoted by the expression $M_1 >> M_2$ (where the expression is relative to the appropriate S and π). We say that M_1 is minimal in π relative to S if there is no such M_2.

The circumscription Circum(S,π) is true in all models of S that are minimal in the π [Etherington 1986]. Therefore to prove that some set of formulas $S \cup T$ entails Circum(S,π) it suffices to show that every model of $S \cup T$ is minimal in π relative to S.

The converse does not always hold, because the notion of a minimal model is powerful enough to capture such structures as the standard model of arithmetic, which cannot be axiomatized [Davis 1980]. In the present work, however, we are only concerned with cases where the set of minimal models can be described by a finite set of first-order formulas. The following assertion about the completeness of circumscription appears to be true, although we have not uncovered a proof:

Supposition: If there is a finite set of first-order formulas T such that the set of models of $S \cup T$ is identical to the set of models minimal in π relative to S, then that set of models is also identical to the set of models of Circum(S,π). Another way of saying this is that circumscription is complete when the minimal-model semantics is finitely axiomatizable.

Given this supposition, to prove that Circum(S,π) entails some set of formulas $S \cup T$ it suffices to show that T holds in every model minimal in π relative to S.

3.6.1. Propositions

The major stumbling block to the use of circumscription is the lack of a general mechanical way to determine how to instantiate the predicate parameters in the second-order formula. The following propositions demonstrate that the first three classes of assumptions discussed above, exhaustiveness, disjointness, and component/use, are equivalent to particular circumscriptions of the event hierarchy. (Note: the bidirectional entailment sign $\Leftrightarrow$ is used instead of the equivalence sign $\equiv$ because the left hand side of each statement is a set of formulas rather than a single formula.)

The first proposition states that the exhaustiveness assumptions (EXA) are obtained by circumscribing the non-basic event types in the abstraction hierarchy. Recall that the abstraction axioms say that every instance of an event type is an instance of the abstractions of the event type. This circumscription minimizes all the event types, except those which cannot be further specialized. Therefore, something can be an instance of a non-basic event type only if it is also an instance of a basic type, and the abstraction axioms entail that it is an instance of the non-basic type. In other words, this circumscription generates the implications from event types to the disjunctions of their specializations.

1. $H_A \cup EXA \Leftrightarrow$ Circum(H_A , $H_E - H_{EB}$)

The second proposition states that the disjointness assumptions (DJA) are obtained by circumscribing all event types other than “AnyEvent”, the most general event type, in the resulting set of formulas. The minimization means that something is an instance of an event type only if it has to be, because it is an instance of AnyEvent, and the exhaustiveness assumptions imply that it is also an instance of a chain of specializations of AnyEvent down to some basic type. In other words, no instance is a member of two different types unless those types share a common specialization; that is, unless they are compatible.

2. $H_A \cup EXA \cup DJA \Leftrightarrow$ Circum($H_A \cup EXA$, H_E–{AnyEvent})

The third proposition states that the component/use assumptions (CUA) result from circumscribing all event types other than End relative to the complete event hierarchy together with the exhaustiveness and disjointness assumptions. Note that in this case both the decomposition and abstraction axioms are used. Intuitively, events of type End “just happen”, and are not explained by their occurrence as a substep of some other event. Minimizing all the non-End types means that events occur only when they are End, or a step of (a subtype of) End, or a step of step of (a subtype of) End, etc. This is equivalent to saying that a event entails the disjunction of all events which could have the first event as a component.

3. $H \cup EXA \cup DJA \cup CUA \Leftrightarrow$ Circum($H \cup EXA \cup DJA$, H_E–{End})

The minimum cardinality assumption cannot be generated by this kind of circumscription. The minimum cardinality assumption minimizes the number of elements in the extension of End, while circumscription performs setwise minimization. A model where the extension of End is {:A, :B} would not be preferred to one where the extension is {:C}, because the the latter extension is not a proper subset of the former.

3.6.2. Covering Models

The various completeness assumptions are intuitively reasonable and, as we have seen, can be defined independently of the circumscription schema. The propositions above might therefore be viewed as a technical exercise in the mathematics of circumscription, rather than as part of an attempt to gain a deeper understanding of plan recognition. On the other hand, the propositions do allow us to use the model theory of circumscription to construct a model theory for the plan recognition. The original event hierarchy is missing information needed for recognition — or equivalently, the original hierarchy has too many models. Each circumscription throws out some group of models that contain extraneous events. The models of the final circumscription can be thought of as "covering models", because every event in each model is either of type End or is "covered" by an event of type End that has it as a component. (This in fact is the lemma that appears below in the proof of proposition 3.)

The minimum cardinality assumption further narrows the set of models, selecting out those containing the smallest number of End events. Therefore the conclusions of the plan recognition theory are the statements that hold in all "minimum covering models" of the hierarchy and observations. This model-theoretic interpretation of the theory suggests its similarity to work on diagnosis based on minimum set covering models, such as that of Reggia, Nau, & Wang [1983].

3.6.3. Proof of Proposition 1

$$H_A \cup EXA \Leftrightarrow \text{Circum}(\, H_A \,,\ H_E - H_{EB}\,)$$

($\Leftarrow$) Suppose $\{E_1, E_2, \ldots, E_n\}$ are all the predicates directly abstracted by E_0 in H_A. We claim that 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 that are minimal in $H_E - H_{EB}$. Let M_1 be a model of H_A in which the statement does not hold. Then 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 on the right-hand side is plainly true in M_2. Axioms of the form

$$\forall x \,.\, E_i(x) \supset E_0(x) \qquad 1 \leq i \leq n$$

are false in M_2 only 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. Otherwise if $:D = :C$, then $:D \in M_2[E_i] \Rightarrow :D \in M_1[E_i] \Rightarrow :C \in M_1[E_i]$ which also is a contradiction. Therefore there can be no such :D, so M_2 is a model of H_A and M_1 is not minimal.

($\Rightarrow$) First we prove the following **lemma:** Every event in a model of $H_A \cup EXA$ is of at least one basic type. That is, 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. The proof is 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

$$:C \in M_1[E_1] \vee \ldots \vee :C \in M_1[E_n]$$

Without loss of generality, 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. This completes the proof of the lemma.

We prove that if M_1 is a model of $H_A \cup EXA$, then M is a model of H_A minimal in H_E-H_{EB}. Suppose not. Then there is an M_2 such that $M_1 >> M_2$, and 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 the lemma, 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}, $:C \in M_2[E_b]$, and because E_0 abstracts E_b, $:C \in M_2[E_0]$, which is a contradiction. Therefore there can be no such M_2, and M_1 is minimal. This completes the proof of the proposition.

3.6.4. Proof of Proposition 2

$$H_A \cup EXA \cup DJA \Leftrightarrow \text{Circum}(H_A \cup EXA, H_E-\{AnyEvent\})$$

($\Leftarrow$) We claim that 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 $H_A \cup EXA$ that are minimal in $H_E-\{AnyEvent\}$. Let M_1 be a model of $H_A \cup EXA$ in which the statement is false and :C be an event such that

$$E_1(x) \wedge E_2(x)$$

is true in M1{x/:C}. Using the lemma from the proof of proposition 1, 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.

$\text{Domain}(M_2) = \text{Domain}(M_1)$

$M_2[Z] = M_1[Z]$ for $Z \notin H_E$

$M_2[E_i] =$ $M_1[E_i]$ if E_i abstracts= E_b

$M_1[E_i]-\{:C\}$ otherwise

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 $H_A \cup$ EXA.

(Proof that M_2 is a model of H_A.) Suppose not ; 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.

(Proof that M_2 is a model of EXA.) Suppose not; 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. Without loss of generality, suppose it is Ej_1. Then $:C \in M_1[E_b] \Rightarrow :C \in M_2[E_b] \Rightarrow :C \in M_2[Ej_1]$, a contradiction.

Note that because E_1 and E_2 are not compatible, E_2 cannot abstract= E_b. Thus $:C \notin M_2[E_2]$, so M_1 and M_2 differ at least on E_2. Therefore $M_1 >> M_2$ so M_1 is not minimal.

($\Rightarrow$) First we note the following **lemma:** Every event in a model of $H_A \cup$ EXA $\cup$ DJA is of exactly one basic type. By the lemma in the proof of proposition 1 there is at least one such basic type, and by DJA no event is of two basic types.

We prove that if M_1 is an model of $H_A \cup$ EXA $\cup$ DJA, then M_1 is minimal in H_E–{AnyEvent} relative to $H_A \cup$ EXA. Suppose there is an M_2 such that $M_1 >> M_2$.and there exists (at least one) $E_0 \in H_E$–{AnyEvent} and event :C such that

$:C \in M_1[E_0] \wedge :C \notin M_2[E_0]$

Then $:C \in M_1[E_0] \Rightarrow :C \in M_1[\text{AnyEvent}] \Rightarrow :C \in M_2[\text{AnyEvent}]$. By the lemma in the proof of proposition 1 there is some $E_b \in H_{EB}$ such that $:C \in M_2[E_b]$. Since $M_1 >> M_2$, it must be the case that $:C \in M_1[E_b]$. By the lemma above, 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 $:C \in M_2[E_b] \Rightarrow :C \in M_2[E_0]$, a contradiction. Therefore there can be no such M_2, and M_1 must be minimal. This completes the proof of proposition 2.

3.6.5. Proof of Proposition 3

$$H \cup EXA \cup DJA \cup CUA \Leftrightarrow \text{Circum}(H \cup EXA \cup DJA , H_E - \{\text{End}\})$$

($\Leftarrow$) First we prove the following **lemma:** Suppose M_1 is a model of $H \cup EXA \cup DJA$ that is minimal in $H_E-\{\text{End}\}$. If $:C_1 \in M_1[E_1]$ for any event predicate E_1, then either $:C_1 \in M_1[\text{End}]$ or there exists some event token $:C_2$ such that $:C_1$ is a direct component of $:C_2$. Suppose the lemma were false. 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. We will show that $H \cup EXA \cup DJA$ holds in M_2 which means that $M_1 >> M_2$, a contradiction. 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

$E_{j0}(x) \wedge \{\neg E_{j1}(f_{j1}(x)) \vee \neg E_{j2}(f_{j2}(x)) \vee \ldots \vee \neg\kappa \}$

is true in $M_2\{x/:C_2\}$. M_1 and M_2 agree on κ, so it must the case that for some j and i, $M_2[f_{ji}](:C_2) \notin M_2[E_{ji}]$ 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 $:C_1$ *is* a component of $:C_2$ in M_1, contrary to our original assumption. This completes the proof of the lemma.

Consider now an arbitrary member of CUA as defined above, which has predicate E on its left hand side. Let M be a model of H ∪ EXA ∪ DJA that is minimal in H_E–{End} such that :C ∈ M[E] and :C ∉ M[End]. By the lemma above 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}. Because M is a model of DJA, 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 the choice of M and the member of CUA was arbitrary, all of CUA is true in all models of H ∪ EXA ∪ DJA that are minimal in H_E–{End}.

(⇒) We prove that if M_1 is a model of H ∪ EXA ∪ DJA ∪ CUA, then it is a model of H ∪ EXA ∪ DJA which is minimal in H_E–{End}. Suppose not; then there is an M_2 such that $M_1 >> M_2$, and there exists (at least one) $E_1 \in H_E$–{End} and event $:C_1$ such that

$:C_1 \in M_1[E_1]$

$:C_1 \notin M_2[E_1]$

We claim that there is a $:C_n$ such that $:C_n \in M_1[\text{End}]$ and $:C_1$ is a component= of $:C_n$. This is obviously the case if $:C_1 \in M_1[\text{End}]$; otherwise, 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 $:C_n \in M_1[\text{End}]$ because H is acyclic. Therefore $:C_1$ is a component= of C_n.

Because M_1 and M_2 agree on End, $:C_n \in M_2[\text{End}]$. Now for any odd j, $3 \leq j \leq n$, if $:C_j \in M_2[E_j]$, then since H_D holds in M_2, $:C_{j-2} \in M_2[E_{j-1}]$. *If* we prove that for all odd j, $1 \leq j \leq n$

$$:C_j \in M_2[E_{j+1}] \Rightarrow :C_j \in M_2[E_j]$$

we will be done; because we would then know that $:C_n \in M_2[\text{End}] \Rightarrow :C_n \in M2[E_{n+1}] \Rightarrow :C_n \in M2[E_n] \Rightarrow :C_{n-2} \in M2[E_{n-1}] \Rightarrow \ldots \Rightarrow :C_3 \in M_2[E_3] \Rightarrow :C_1 \in M_2[E_2] \Rightarrow :C_1 \in M_2[E_1]$ which yields the desired contradiction. So assume the antecedent $:C_j \in M_2[E_{j+1}]$. Because M_2 is a model of $H_A \cup EXA \cup DJA$, there is a unique $E_b \in H_{EB}$ such that $:C_j \in M_2[E_b]$. Because $M_1 >> M_2$, $:C_j \in M_1[E_b]$. Since M_1 is a model of $H_A \cup EXA \cup DJA$, by the lemma in the proof of proposition 2 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. This completes the proof of proposition 3.

4. Examples

4.1. An Operating System

Several research groups have examined the use of plan recognition in "smart" computer operating systems that could answer user questions, watch what the user was doing, and make suggestions about potential pitfalls and more efficient ways of accomplishing the same tasks [Huff & Lesser 1982, Wilensky 1983]. A user often works on several different tasks during a single session at a 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.

4.1.1. Representation

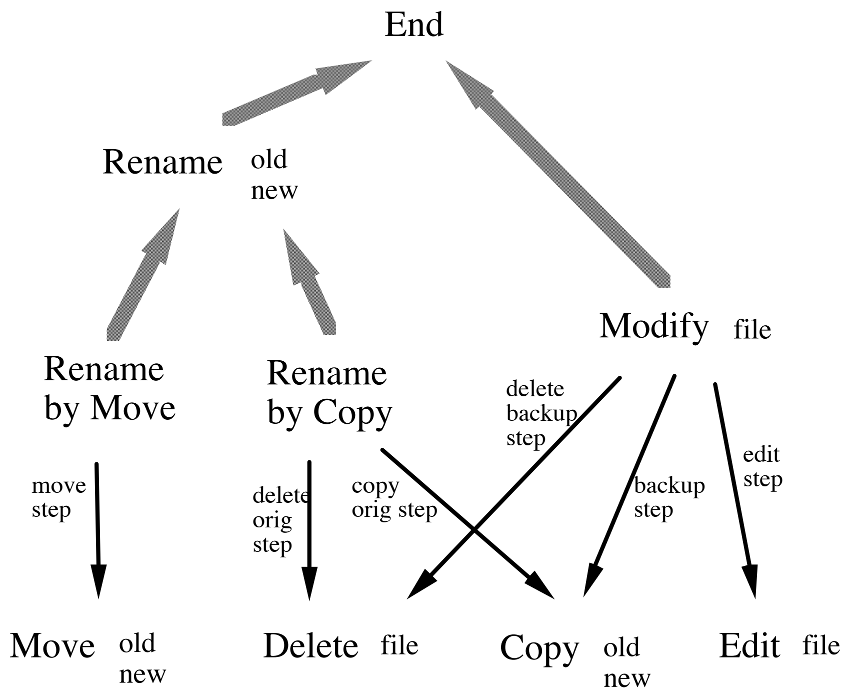


figure 2: Operating System Hierarchy.

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.

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

$$\begin{aligned}
&\forall x\ .\ \mathrm{RenameByCopy}(x) \supset \\
&\quad \mathrm{Copy}(s1(x)) \wedge \\
&\quad \mathrm{Delete}(s2(x)) \wedge \\
&\quad \mathrm{old}(s1(x)) = \mathrm{old}(x) \wedge \\
&\quad \mathrm{new}(s1(x)) = \mathrm{new}(x) \wedge \\
&\quad \mathrm{file}(s2(x)) = \mathrm{old}(x) \wedge \\
&\quad \mathrm{BeforeMeet}(\mathrm{time}(s1(x)), \mathrm{time}(s2(x))) \wedge \\
&\quad \mathrm{Starts}(\mathrm{time}(s1(x)), \mathrm{time}(x)) \wedge \\
&\quad \mathrm{Finishes}(\mathrm{time}(s2(x)), \mathrm{time}(x))
\end{aligned}$$

A better way to rename a file is to RenameByMove, which simply uses the Move command.[4] A helpful system might suggest that a user try the Move command if it recognizes many instances of RenameByCopy.

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

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

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.

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

[4] Another way to represent this information would be to make Move a specialization of Rename. This would mean that *every* Move would be recognized as a Rename, and therefore as an End event. This alternative representation would not be appropriate if there were End events other than Rename which included Move as a component.

4.1.2. 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.

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

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

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

4.1.3. 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 do not 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:

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

The component/use assumption 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.

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

Statement (2) is encoded:

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

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

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

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

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

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:

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

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.

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

In this case the formula is instantiated as follows.

*I1=*I2 ∨ *I1=*I3 ∨ *I2=*I3

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

*I1=*I3 ∨ *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.

```
End(*I2) ∧
(       (RenameByCopy(*I2) ∧ C2=s1(*I2) ∧ C3=s2(*I2) ∧
 ✘      old(*I2) = jack ∧ old(*I2) = foo )
∨
        (Modify(*I2) ∧ C2=s1(*I2) ∧ C3=s3(*I2) ∧
        new(s1(*I2)) = jack ∧
        file(*I2) = jack ∧
        backup(*I2) = sprat ∧
        file(s3(*I2)) = foo ∧
 ✘      backup(*I2) = file(s3(*I2)) )
)
```

But both disjuncts are impossible, since the files that 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) ∧
        new(s1(*I1)) = bar ∧
        backup(*I1) = bar ∧
        file(s3(*I1)) = foo ∧
 ✘      backup(*I1) = file(s3(*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
```

4.2. Medical Diagnosis

There are close links between the kinds of reasoning involved in plan recognition, and that employed in medical diagnosis. The vocabulary of events can be mapped to one appropriate for diagnosis in a straightforward way. Events are replaced by *pathological states* of a patient. An *abstraction* hierarchy over pathological states is known as a *nosology*. The *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 that can be directly observed are *symptoms*. (Note that a symptom may also cause other other states: for example, 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 entails the disjunction of its uses. Once several findings are obtained, the diagnostician attempts to find a small set of diseases that accounts for, or covers, all the findings. This step corresponds to the minimization of End events. 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 that play a central role in medical reasoning. In some ways medical diagnosis is easier than plan recognition. Medical knowledge can often be represented in propositional logic. There is usually no need for diseases to include parameters, the way plans do, and most medical expert systems do not need to deal with the passage of time.

The following example is drawn from [Pople 1982], 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 a pathological state (abstract event type) in order to explain a symptom, finding, or state, and the process of combining or unifying the differential diagnosis sets invoked by each finding.

4.2.1. 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 have added the type End as an abstraction of all pathological states that are not caused by other pathological states. The *basic* specializations of End are called *specific disease entities*. We have 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.

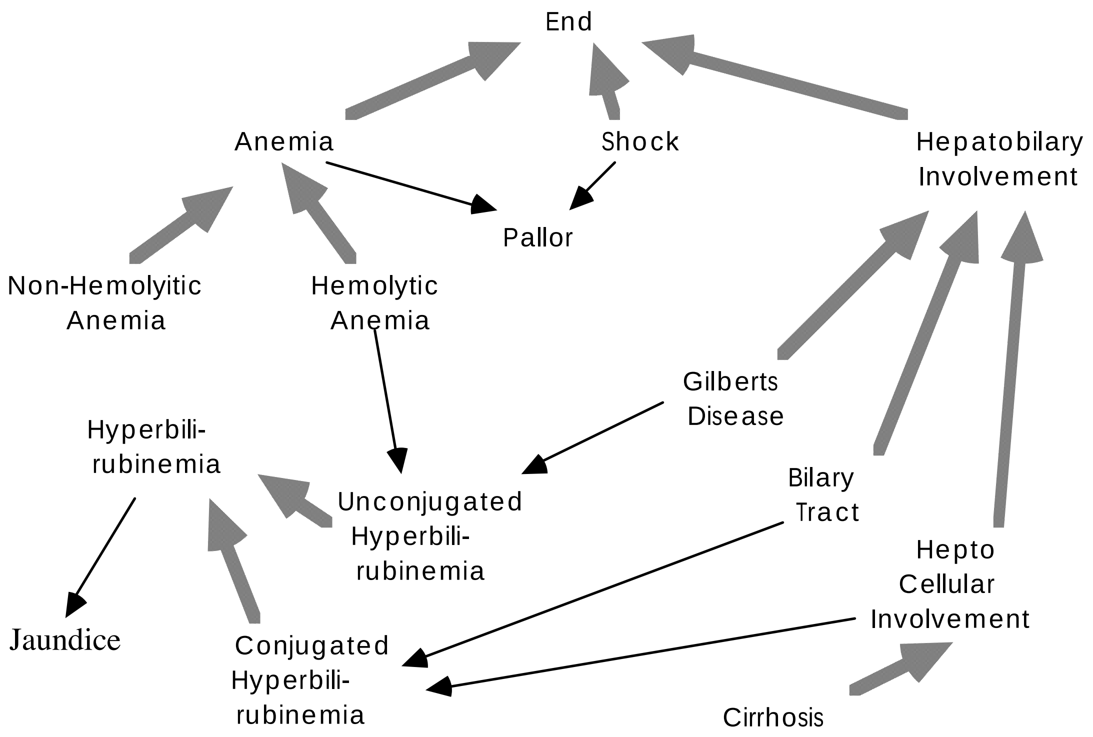


figure 3: Medical Hierarchy.

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. Constraints are not used in this example.

For the sake of clarity the names of the role functions have been omitted from component (symptom) arcs in the illustration, but such functions are needed in the logical encoding. We will use the letters j, h, p for role functions, where j is used for symptoms of type **jaundice**, h for symptoms of type **hyperbilirubinemia**, and p for symptoms of type **pallor**. The names used is really not important, except that all the different symptoms caused by a particular disease must be related to it by different role functions.

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. (This corresponds to the disjointness assumptions in our system.) But this would plainly be incorrect, if the two states were causally related.

4.2.2. 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**.

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

Disjointness 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**.

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

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

4.2.3. 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 have developed, however, allows us continue inference in each alternative. The first leads to either **hemolytic anemia** and then **anemia**, or **Gilbert's disease** and then **hepatobilary involvement**. The second leads to either **bilary tract**

disease or **hepto cellular involvement**, both of which lead to **hepatobilary involvement**. The following graph shows the final conclusion.

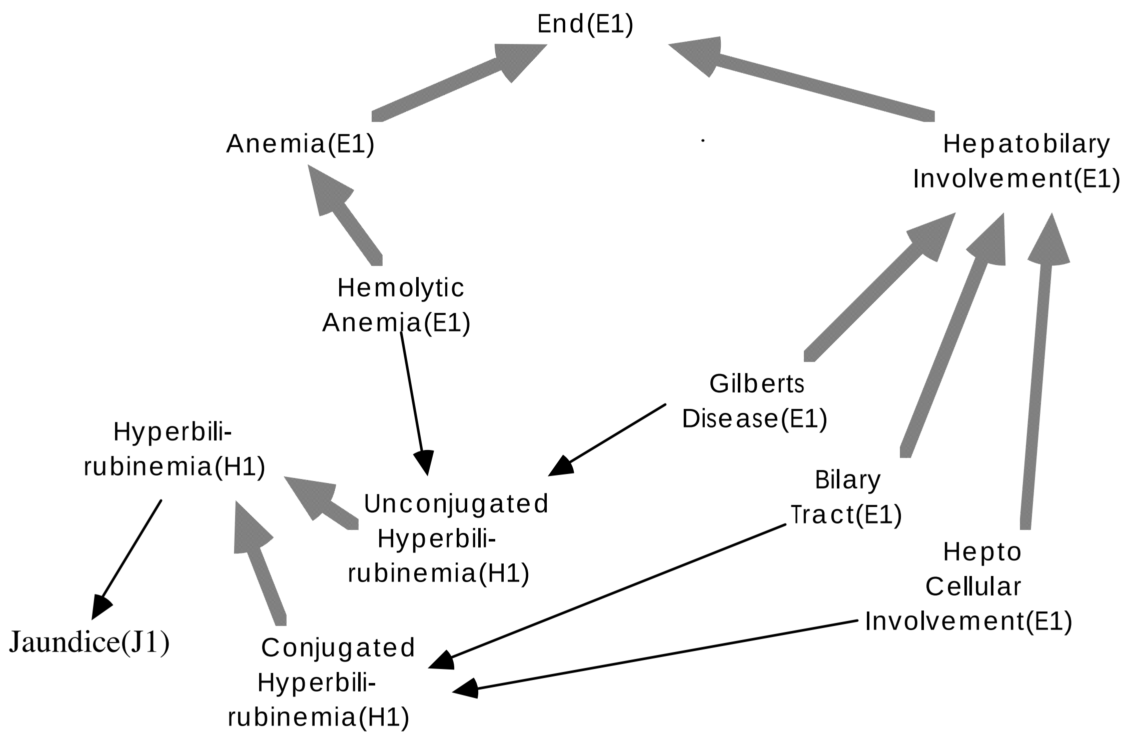

End(E1)
Anemia(E1)
Hepatobilary Involvement(E1)
Hemolytic Anemia(E1)
Gilberts Disease(E1)
Hyperbili-rubinemia(H1)
Unconjugated Hyperbili-rubinemia(H1)
Bilary Tract(E1)
Hepto Cellular Involvement(E1)
Jaundice(J1)
Conjugated Hyperbili-rubinemia(H1)

figure 4: Conclusions from Jaundice.

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

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

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

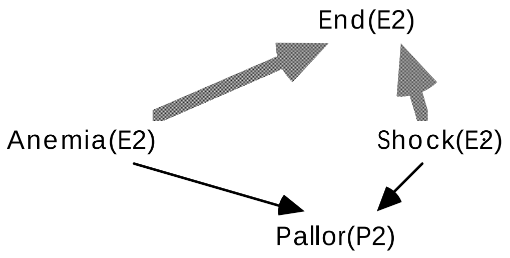


figure 5: Conclusions from Pallor.

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**.

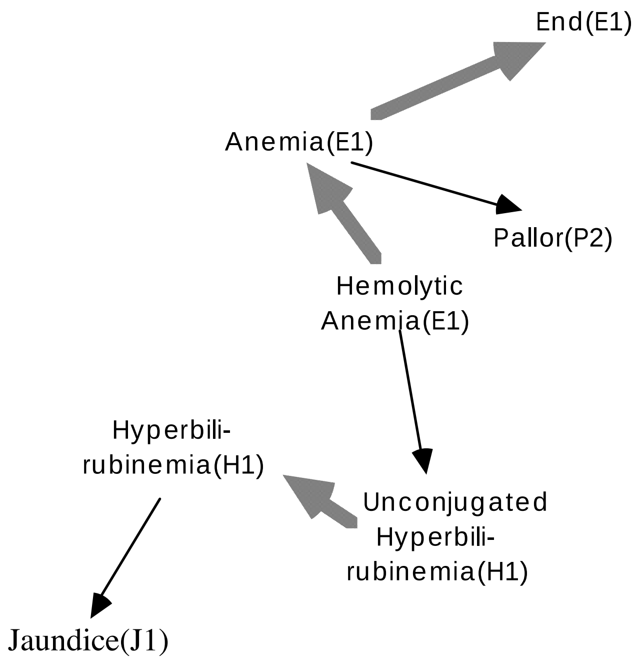


figure 6: Conclusions from Jaundice and Pallor.

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. In particular, we show how to "invert" a knowledge base which allows inferences from causes to effects (diseases to symptoms) to one which allows inferences from effects to causes. Like the minimum set covering model of [Reggia, Nau, & Wang 1983] we combine information from several symptoms by finding the smallest set(s) of diseases which explains all the symptoms. One should note that our framework depends on having a model of disease, but not a model of healthy processes. This is to be contrasted with work on "diagnosis from first principles", such as [Reiter 1987], which models the correct or healthy functioning of a system, and which derives a diagnosis by trying to resolve inconsistencies between this model and the actual observations.

5. Algorithms for Plan Recognition

5.1. Directing Inference

An implementation of this theory of plan recognition must limit and organize the inferences drawn from the observations, hierarchy, and assumptions. As noted before, the formal theory sanctions an infinite set of conclusions. An implementation, on the other hand, must compute some finite data structure in a bounded amount of time.

The framework developed in this chapter does not pose the plan recognition problem in such a way that a solution is simply the name of a specific plan. Instead, a solution is a partial description of the plans that are in progress. Different encodings of this description make aspects of the plan more or less explicit. For example, an encoding that leaves most aspects of the plans in progress implicit is simply the set of sentences in the observations, hierarchy, and assumptions. A more explicit encoding would be a disjunction of all the possible End plan types that explain the observations. A very explicit encoding may not only be expensive to compute, but may also involve a loss of detail. All information is contained in the observations, hierarchy, and assumptions, but the more explicit representation may include (for example) unnecessary disjunctions due to the incompleteness of the inference method employed. In fact, some degree of incompleteness is necessary in any implementation of our formal theory, because the definition of the minimum cardinality assumption appeals to the consistency of a set of sentences, an undecidable property.

The less inference the plan recognition system performs, the more inference must be formed by the programs that call the plan recognizer as a subroutine. The implementation described in this section tries to strike a balance between efficiency and power. The choices we have made are to some degree arbitrary. We have not tried to develop a formal theory of limited inference to justify these choices, although that would be an interesting and useful project. Instead, we have been guided by the examples of plan recognition discussed in this chapter and elsewhere in the literature, and have tried to design a system that makes most of inferences performed in those examples, and that outputs a data structure that makes explicit most of the conclusions drawn in those examples.

The implementation performs the following pattern of reasoning: from each observation, apply component/use assumptions and abstraction axioms until an instance of type End is reached. Reduce the number of alternatives 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 are recognized by considering all ways of grouping the observations. As noted earlier, it is not possible to detect all inconsistencies. Therefore the "best" explanation returned by the system (by the function **minimum-Hypoths**, described below) may in fact be inconsistent, because observations were grouped together which logically could not be combined.

The algorithms presented here only handle event hierarchies in which the abstraction hierarchy is a forest. That is, they do not deal with "multiple inheritance". (But note that the decomposition hierarchy is not necessarily a forest — one event type may be a component of many others. For example, MakeMarinara falls below both MakeSpaghettiMarinara and MakeChickenMarinara in the decomposition hierarchy.) Furthermore, in order to limit inference, the implementation does not perform inferences that correspond to applications of the decomposition axioms. This restriction limits the predictive power of the system to some degree. The implementation could not reach one of the conclusions drawn in the example presented in section 3.5, where the observer concludes that the agent is making spaghetti because the agent is known to be making either spaghetti marinara or spaghetti pesto. However, the implementation described here does make explicit all the other conclusions presented in the examples in this chapter.

5.2. Explanation Graphs

Chaining through all the component/use assumptions raises the prospect of generating extremely long disjunctive conclusions. The first assumption generates a disjunction; applying the assumptions to each disjunct in this formula generates another disjunction; and so on. In general this process multiplies out the event hierarchy from the bottom up, creating a number of literals exponential in the size of the original hierarchy. This problem has led other plan recognition systems to limit component/use type inferences to a single application.

Our implementation meets this problem by using a non-clausal representation of the plan description. The representation takes the form of a labeled acyclic graph, called an "explanation graph" or "e-graph". These graphs can encode disjunctive assertions in a compact form through structure-sharing. While in the worst case the e-graph generated by an observation can be exponential in the size of the event hierarchy, in all the examples we have considered structure sharing allows the e-graph to be smaller than the hierarchy.

The figure below shows the e-graph that is generated for an observation of MakeMarinara.

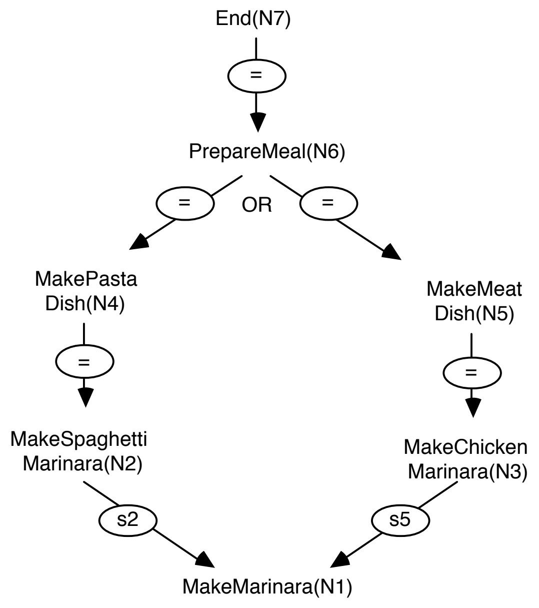


figure 7: e-graph for MakeMarinara(N1).

An e-graph contains the following three kinds of nodes. **Event nodes** consist of an event token and a single associated event type. An event token N appears at most once in an e-graph, so we can simply identify the token with the node, and recover the associated type with the form type(N). In the figure above, N1 through N7 are all event nodes. **Proper names** are unique names for objects other than events and times. **Fuzzy temporal bounds** are 4-tuples of real numbers. (No proper names or fuzzy temporal bounds appear in the figure above.)

There are two kinds of arcs, both of which can only lead out of event nodes. A **parameter** arc is labeled with a role function f_r and points to its value v. When the value is an event node or a proper name, the arc means that $f_r(N)=v$. The figure above contains two parameter arcs, labeled s2 and s5. When the value is a fuzzy temporal bound, the arc means that according to some universal clock, the beginning of the time interval $f_r(N)$ falls between the first two numbers and the end between the last two numbers. Below we will discuss how fuzzy time bounds can be combined, so that temporal constraint propagation can be performed without recourse to a separate temporal database program.

The second kind of arcs are **alternative** arcs, which always point at event nodes. The meaning of these arcs is that the (event token identified with the) origin node is equal to one of the nodes pointed to by an alternative arc. Note that event nodes (tokens) are not unique names for events; in general, several nodes in an e-graph will represent the same real-world event. The alternative arcs point from a node of a particular type to a node of a more specialized type. The alternative arcs in the figure above are labeled with an "=" sign.

Every e-graph G contains exactly one node N of type End. The translation of that graph into sentential logic is given by TRANS(N), where TRANS is given by the following recursive definition.

$\mathrm{TRANS}(n) =$

$\quad \mathrm{type}(n)\,(n) \wedge$

$\quad \bigwedge\{v=f_r(n) \wedge \mathrm{TRANS}(v) \mid \langle n,f_r,v\rangle \in G \text{ and } v \text{ is an event node}\} \wedge$

$\quad \bigwedge\{v=f_r(n) \mid \langle n,f_r,v\rangle \in G \text{ and } v \text{ is a proper name}\} \wedge$

$\quad \bigwedge\{r1 \;{}^2\; f_r(n)^- \;{}^2\; r2 \wedge r3 \;{}^2\; f_r(n)^+ \;{}^2\; r4 \mid \langle n,f_r, \langle r1, r2, r3, r4\rangle\rangle \in G\} \wedge$

$\quad \bigvee\{n=m \wedge \mathrm{TRANS}(m) \mid \langle n, =, m\rangle \in G\}$

In the first line of the translation, the expression "type(n)" stands for the event type predicate associated with node n. The full expression "type(n)(n)" means that this predicate is applied to the logical constant n. Arcs in the graph are represented by triples, consisting of a node, a label, and a node. In the definition above, $\langle n,f_r,v\rangle$ is an arc from event node n, labeled with role function f_r, to event node v. Similarly, $\langle n,f_r, \langle r1, r2, r3, r4\rangle\rangle$ is an arc from event node n, labeled with temporal parameter f_r, to fuzzy temporal bound $\langle r1, r2, r3, r4\rangle$. The triple $\langle n, =, m\rangle$ indicates that node m is an alternative for node n in the graph. The postfix functions – and + apply to a time interval and return the metric time of the start and end of interval respectively. The event tokens in the translation are interpreted as Skolem constants (existentially quantified variables). The translation of figure 7 is the sentence

```
End(N7) ∧ N7=N6 ∧ PrepareMeal(N6) ∧
    ( ( N6=N4 ∧ MakePastaDish(N4) ∧
            N4=N2 ∧ MakeSpaghettiMarinara(N2) ∧
            step2(N2)=N1 ∧ MakeMarinara(N1) )
    ∨ ( N6=N5 ∧ MakeMeatDish(N5) ∧
            N5=N3 ∧ MakeChickenMarinara(N3) ∧
            step5(N3)=N1 ∧ MakeMarinara(N1) ) )
```

An e-graph describes a single End event. When the plan recognizer determines that more than one End event is in progress it returns a set of End events, whose interpretation is the conjunction of the interpretations of each e-graph. When the observations can be grouped in different ways the recognizer returns a set of sets of End events, whose interpretation is the disjunction of the interpretation of each set. For example, given three observations where no plan contains all three, but some plans contain any two, the recognizer returns a set of the form {{g1&2, g3}, {g1, g2&3}, {g1&3, g2}}.

5.3. Implementing Component/Use Assumptions

A component/use assumption leads from an event to the disjunction of all events that have a component that is compatible with the premise. Is is desirable that this disjunction not be redundant; for example, it should not contain both an event type and a more specialized version of the event type. The algorithm below implements the component/use assumptions by first considering the cases that explicitly use the premise event type (plus a few others, as we shall explain in a moment); next considering the cases that explicitly use a type that specializes the premise type, and that are not redundant with respect to the first group of cases; and finally considering the cases that explicitly use a type that abstracts the premise type, and are not redundant with respect to the previous cases.

A *use* is a triple, $\langle E_c, f_r, E_u \rangle$, and stands for the possibility that (some instance of) E_c could fill the r role of some instance of E_u. The set *Uses* contains all such triples considered by the algorithm. We shall say that $\langle E_{ac}, r, E_{au} \rangle$ *abstracts* $\langle E_c, r, E_u \rangle$ exactly when E_{ac} abstracts E_c and E_{au} abstracts E_u. The inverse of abstraction is as before called specialization. This notion of abstraction is used to eliminate redundancy in the implementation of the component/use assumptions. For example, ⟨MakeNoodles, s_1, MakePastaDish⟩ abstracts ⟨MakeSpaghetti, s_1, MakeSpaghettiMarinara⟩. Therefore once the algorithm has considered the possibility that an event of type MakeSpaghetti is a component of an event of MakeSpaghettiMarinara, it would be redundant to consider the possibility that the more abstract description of the premise, MakeNoodles, is a component of an event of type MakePastaDish.

Uses contains the inverse of the direct component relation, together with certain other "implicit" uses. Consider the following modified cooking hierarchy.

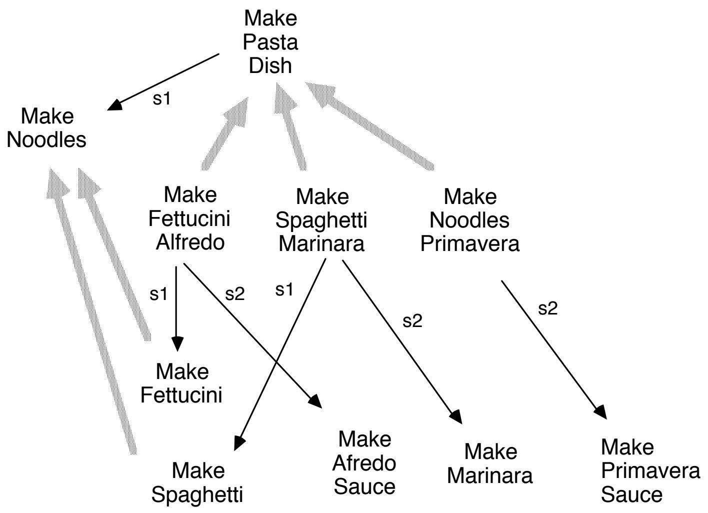


figure 8: Example of implicit Use.

This new hierarchy introduces the type MakeNoodlesPrimavera which has no direct component filling the s_1 role, and which therefore inherits the component MakeNoodles from MakePastaDish. Suppose the observation is of type MakeSpaghetti. It would be wrong to conclude

$$\exists\, x \,.\, \text{MakeSpaghettiMarinara}(x)$$

because the observation could have been a step of MakeNoodlesPrimavera. The conclusion

$$\exists\, x \,.\, \text{MakeSpaghettiMarinara}(x) \vee \text{MakePastaDish}(x)$$

is sound but too weak; the check for redundancy would notice that the type of the second disjunct abstracts the first. The correct conclusion is

$$\exists\, x \,.\, \text{MakeSpaghettiMarinara}(x) \vee \text{MakeNoodlesPrimavera}(x)$$

Therefore the set Uses should contain the implicit use $\langle$MakeSpaghetti, s_1, MakeNoodlesPrimavera$\rangle$ as well as $\langle$MakeSpaghetti, s_1, MakeSpaghettiMarinara$\rangle$.

Define *Explicit* as the set of explicit uses corresponding to the direct-component relation. Then Uses is the union of Explicit together with the following set:

{ $\langle E_c, r, E_i \rangle$ | (1) There is an E_u such that $\langle E_c, r, E_u \rangle \in$ Explicit and
(2) There is an $\langle E_{ac}, r, E_{au} \rangle \in$ Explicit that abstracts $\langle E_c, r, E_u \rangle$ and
(3) E_{au} abstracts E_i and
(4) There is no E_j that either specializes, abstracts, or is equal to E_i such that there is an $\langle E_d, r, E_j \rangle \in$ Explicit that specializes $\langle E_{ac}, r, E_{au} \rangle$ and
(5) There is no E_{ai} that abstracts E_i for which conditions (3) and (4) hold (with E_{ai} in place of E_i). }

In this construction, E_i is a type such as MakeNoodlesPrimavera that inherits a role without further specializing it. The final condition (5) prevents redundant uses that simply specialize this new use from being added (for example, if there were a type MakeSpicyNoodlesPrimavera).

5.4. Constraint Checking

The algorithms check the various constraints that appear in the decomposition axioms by failing to prove that the constraint is false. As a side effect of constraint checking, the values of non-component parameters are propagated to a node from its components. (The algorithms presented here do not propagate values *from* a node *to* its components.) There are three different kinds of constraints.

An **equality constraint** equates two roles of a node or its components whose values are neither events nor times. An equality constraint fails if the values of the two items are distinct proper names. As a side effect, when a value is known for a parameter of a component, but none for an equal parameter of the node, the value is assigned to the parameter of the node.

Temporal constraints take the form of binary predicates over the time parameters of a node or its components. As noted above, rather than passing around precise values for time parameters, the implementation passes fuzzy temporal bounds. It is straightforward to check if any particular temporal relation could hold between two times that have been assigned temporal bounds. For example, if *time(N)* is bounded by $\langle 1\ 3\ 7\ 9 \rangle$, and *time(step1(N))* is bounded by $\langle 4\ 5\ 6\ 7 \rangle$, it is clearly impossible for the relation *started-by(time(N), time(step1(N)))* to hold.

Furthermore, it is easy to generate a set of rules that can tighten the fuzzy bounds on an interval given its relation to another interval and the fuzzy bounds on that interval. One such rule is:

IF T_1 is bounded by ⟨a b c d⟩ and T_2 is bounded by ⟨e f g h⟩
and started-by(T_1, T_2) THEN
T_1 is bounded by ⟨max(a,e) min(b,f) max(c,g) d⟩

The implementation uses such rules to update the fuzzy bounds assigned to the temporal parameters of a node during constraint checking. The following table describes the rules for the 13 basic temporal relations, where ⟨a b c d⟩ is the original bound on T_1, ⟨e f g h⟩ is the bound on T_2, and ⟨i j k l⟩ is the updated bound on T_1.

relation	i	j	k	l
equals	max(a,e)	min(b,f)	max(c,g)	min(d,f)
before	a	min(b,f)	c	min(d,f)
after	max(a,g)	b	max(c,g)	d
meets	a	min(b,f)	max(c,e)	min(d,f)
met by	max(a,g)	min(b,h)	max(c,g)	d
overlaps	a	min(b,f)	max(c,e)	min(d,h)
overlapped by	max(a,e)	min(b,h)	max(c,g)	d
starts	max(a,e)	min(b,f)	max(c,e)	min(d,h)
started by	max(a,e)	min(b,f)	max(c,g)	d
during	max(a,e)	min(b,h)	max(c,e)	min(d,h)
contains	a	min(b,f)	max(c,g)	d
finishes	max(a,e)	min(b,h)	max(c,g)	min(d,h)
finished by	a	min(b,f)	max(c,g)	min(d,h)

All unknown times are implicitly bounded by ⟨−_ +_ −_ +_⟩. When the "matching" algorithm below equates two event tokens the "intersection" of the fuzzy time bounds is taken. An advantage of this approach over a purely symbolic implementation of Allen's temporal algebra is that the system does not need to maintain a table relating every interval to every other interval (as in [Allen 1983b]). A single data structure – the e-graph – maintains both temporal and non-temporal information. The use of fuzzy time bounds precludes the expression of certain kinds of relationships between specific event instances. For example, it is not possible to record an observation that the times of two event instances are disjoint, without saying that one is before or after the other. (It is possible, of course, to include a constraint that two times are disjoint in the decomposition axioms for an event *type*.) In many domains, however, it is reasonable to assume that specific observations can be "timestamped", and it seems important to provide some mechanism for metric information, even if symbolic information is not handled in full generality.

Fact constraints include the preconditions and effects of the event along with type information about the event's non-component parameters. Facts are checked only if values are known for all the arguments of the predicates in the constraint. The constraint is satisfied if a limited theorem prover fails to prove the negation of the constraint from a global database of known facts. A limitation of the implementation described here is that the global database is not augmented by the conclusions of the plan recognizer itself. A worthwhile extension of the system would make it assert in the global database the preconditions and effects of any unambiguously recognized plans. The would be useful for the kind of predictive reasoning discussed in Part 1. A difficult problem we have avoided dealing with is recognizing that a fact constraint is violated because it is inconsistent with all the possible disjunctions encoded in the e-graph.

5.5. Algorithms

The algorithm **explain-observation** implements the component/use assumptions, and the algorithms **match-graphs** and **group-observations** implement the minimum cardinality assumptions. Code for each algorithm is followed by commentary on its operation.

5.5.1. Explain-observation

```
/* explain-observation
        Ec : type of observed event
        parameters : list of role/value pairs that describe the observation
returns
        G : explanation graph
*/
function explain-observation(Ec, parameters) is
        Let G be a new empty graph
        G := explain(Ec, parameters, ∅, {Up, Down})
        return G
end build-explanation-graph
```

```
/* explain
        Ec : type of event to be explained
        parameters : list of ⟨role value⟩ pairs that describe the event
        visited : set of event types visited so far in moving through abstraction hierarchy
        direction : direction to move in abstraction hierarchy; subset of {Up, Down}
returns
        N : node that represents the event of type Ec
        newVisited: updated value of visited
*/
function explain(Ec, parameters, visited, direction) is
        visited := visited ∪ {Ec}
        if G has a node N of type Ec with matching parameters then
                return ⟨N, visited⟩
        Add a new node N of type Ec to G
        Add the parameters of N to G
        if Ec=End then return ⟨N, visited⟩
        Propagate constraints for N
        if constraints violated then return ⟨N, visited⟩
        for all ⟨Ec, r, Eu⟩ ∈ Uses do
                if ⟨Ec, r, Eu⟩ does not abstract or specialize a use
                        for  any member of visited then
                                explain(Eu, {⟨r,N⟩}, ∅, {Up, Down})
        if Down ∈ direction then
                for all Esc ∈ direct-specializations(Ec)
                        ⟨M, visited⟩ := explain(Esc, parameters, visited, {Down})
        if Up ∈ direction then {
                Eac := direct-abstraction(Ec)
                p := the role/value pairs for N restricted to those roles defined
                        for Eac or higher in the abstraction hierarchy
                ⟨M, visited⟩ := explain(Eac, p, visited, {Up})
                Add ⟨M, =, N⟩ to G }
        return ⟨N, visited⟩
end explain
```

The function **explain-observation** builds an e-graph on the basis of a single observation. For example, for the observation that Joe is making marinara sauce starting between times 4 and 5 and ending between times 6 and 7 the function call would be

explain-observation(MakeMarinara, {⟨agent Joe⟩, ⟨time ⟨4 5 6 7⟩⟩})

Explain-observation calls the subroutine **explain** which operates as follows:

• Check whether the graph under construction already contains a node of the given type that exactly matches the given parameters. If this is the case, then the graph merges at this point, rather than getting wider and wider as one moves upward. Consider the e-graph shown in figure 7 (section 5.3 above). 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** that created MakeMeatDish(N5) is considering abstractions of MakeMeatDish (see below), and recursively calls **explain** with type PrepareMeal. The specific call would be:

explain(PrepareMeal, {…}, {MakeMeatDish}, {Up})

This description exactly matches previously-created node N6, which is returned. Then N5 is made an alternative for N6 (in the third to last line of the procedure). 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 **Ec**, and link all the **parameters** to it.

• Check whether the type of the newly-created node is End, and return if so.

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

Fuzzy time bounds are also propagated. N2 is constrained to occur over an interval that contains the time of N1. Suppose the graph initially contains ⟨N1 time ⟨4 5 6 7⟩⟩. After this step, it also contains ⟨N2 time ⟨–_ 5 6 +_⟩⟩.

This step can also eliminate nodes. The agent of every specialization of MakePastaDish is constrained to be dexterous. If the general world knowledge base contains the assertion ¬Dexterous(Joe), **explain** does not continue to build a path to End from this node.

• Consider Uses of **Ec**, as described above. If **Ec** is MakeMarinara, then **explain** is recursively invoked for MakeSpaghettiMarinara and MakeChickenMarinara.

• Explain **Ec** by considering its specializations. This step is not performed if **Ec** was reached by abstracting some other type. Suppose **explain** were initially invoked with **Ec** 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 the test in the **if** statement within the first **for all** statement in the procedure) because it specializes ⟨MakeSauce, step2, MakePastaDish⟩. The use ⟨MakeMarinara, step5, MakeChickenMarinara⟩, however, *does* lead to a path to End.

• Explain **Ec** by considering its abstractions. This step is not performed if **Ec** was reached by specializing some other type. 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}, {Up})

which returns N6.

Not all abstractions lead to End; some are pruned, and do not appear in the final graph. Consider the invocation that 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 redundancy test. Therefore no node of type MakeSauce appears in the final graph.

The worst-case complexity of **explain** is exponential in the size of the event hierarchy, because an event can have several different components of the same type. In practice the first step in **explain** frequently finds a similar node and cuts off search. For example, suppose that the event hierarchy contains no non-component roles, and if a type has components, only its abstractions (but not itself or its specializations) appear as components of another type. Under this restriction, the worst case complexity of **explain** is $O(|H_E|)$. If the algorithm did not merge search paths at abstraction points, its worst-case complexity would still be exponential.

5.5.2. Match-graphs

```
/* match-graphs
        G1, G2 : graphs to be matched
returns
```

```
        G3 :  result of equating End nodes of G1 and G2 or FAIL if no match possible
*/
function match-graphs(G1, G2) is
        Create a new empty graph G3
        Initialize Cache, a hash-table that saves results of matching event nodes
        if match(End-node-of(G1), End-node-of(G2)) = FAIL
                then return FAIL
                else return G3
end match-graphs

/* match
        n1 , n2 : nodes to be matched from G1 and G2 respectively
returns
        n3 : node in G3 representing match or FAIL if no match
*/
function match(n1, n2) is
        if n1 and n2 are proper names then
                if n1=n2 then return n1 else return FAIL
        else if n1 and n2 are fuzzy temporal bounds then {
                ⟨a b c d⟩ := n1
                ⟨e f g h⟩ := n2
                ⟨i j k l⟩ :=  ⟨max(a,e) min(b,f) max(c,g) min(d,h)⟩
                if i > j or k > l or i > l then return FAIL else return ⟨i j k l⟩  }
        else if n1 and n2 are event nodes then {
                if Cache(n1,n2) is defined then return Cache(n1,n2)
                if type(n1) abstracts= type(n2)  then n3Type := type(n2)
                else if type(n2) abstracts type(n1)  then n3Type := type(n1)
                else {Cache(n1,n2) := FAIL
                        return FAIL }
                Add a new node n3 of n3Type to G3
                Cache(n1,n2) := n3
                for all roles r defined for n3Type or higher do {
                        Let V1 be the value such that ⟨n1,r,V1⟩∈ G1 (or undefined)
                        Let V2 be the value such that ⟨n2,r,V2⟩∈ G2 (or undefined)
                        if either V1 or V2 is defined then {
                                if V1 is defined but not V2 then V3 := match(V1,V1)
                                elseif V2 is defined but not V1 then V3 :=match(V2,V2)
                                else V3 := match(V1,V2)
                                if V3=FAIL then {    Cache(n1,n2) := FAIL
                                                     return FAIL  }
                                Add ⟨n3, r, V3⟩ to G3  }  }
                Propagate constraints for n3
                if constraints violated then {      Cache(n1,n2) := FAIL
                                                    return FAIL }
                alts1 := { A1 | ⟨n1, =, A1⟩∈ G1 }
                alts2 := { A2 | ⟨n2, =, A2⟩∈ G2 }
                if alts1 ∪ alts2 _ ∅ then {
                    if alts1 = ∅ then alts1 := {n1}
                    if alts2 = ∅ then alts2 := {n2}
                    noneMatched := TRUE
                    for all a1 ∈ alts1 do
                        for all a2 ∈ alts2 do {
                            A3 := match(A1, A2)
                            if A3 _ FAIL then {
                                Add ⟨n3, =, A3⟩ to G3
                                noneMatched := FALSE } }
                    if noneMatched then {       Cache(n1,n2):=FAIL
                                                return FAIL } }
                return n3 }
        else return FAIL
end match.
```

The function **match-graphs** creates a new e-graph that is the result of equating the End nodes of the two e-graphs it takes as inputs and propagating that equality. 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**.

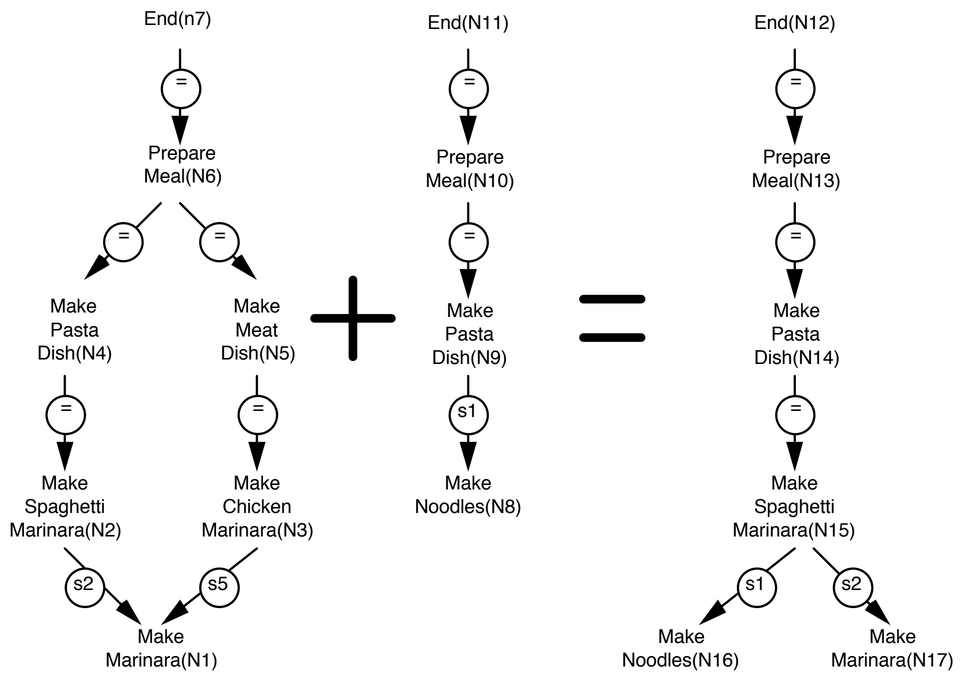


figure 9: Matching e-graphs.

• If the objects to be matched are proper names, they must be identical.

• If the objects are fuzzy temporal bounds, then take their intersection. If **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 that *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 most specific 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 parameter 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** parameter from N2 (a copy of N1, which is N17) and the **step1** parameter 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.

• 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.

• 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 MakeSpaghettiMarinara(N2) matches against MakePastaDish(N9) as well. If there were some alternatives but all matches failed, then **n3** fails as well.

Calling **match-graphs(G_1,G_2)** frequently returns a graph that is smaller than either G_1 or G_2. Unfortunately, sometimes the resulting graph can be larger. This occurs when two nodes are matched that have several alternatives, all of which are mutually compatible. Therefore the worst case complexity of match-graphs(G_1,G_2) is $O(|G_1|*|G_2|)$. The key feature of match-graphs is the use of a cache to store matches between nodes. Without this feature, match-graphs would always multiply the input digraphs into a tree, and the algorithm would be no better than $O(2^{|G_1|*|G_2|})$.

5.5.3. Group-observations

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

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

procedure group-observations is
```

```
        Hypoths := {∅}
        while more observations do {
                Observe event of type Ec with specified parameters
                Gobs := explain-observation(Ec, parameters)
                for all H ∈ Hypoths do {
                        remove H from Hypoths
                        Add H ∪ {Gobs} to Hypoths
                        for all G ∈ H do {
                                Gnew := match-graphs(Gobs, G)
                                if Gnew _ FAIL then
                                        Add (H – {G}) ∪ Gnew to Hypoths } } }
end group-observations
```

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 that accounts for all of the observations, using as few End events as possible. It works as follows:

• Input an observation and generate a new e-graph **Gobs**.

• *Conjoin* **Gobs** with each hypothesis. This handles the case where **Gobs** is *unrelated* to the previous observations. Note that this case is included in **Hypoths** even when **Gobs** matches one of the previous e-graphs. This is necessary because a later observation may be able to match **Gobs** alone, but *not* be able to match **Gobs** combined with that previous e-graph. In any case, the function **minimum-Hypoths** will select those members of **Hypoths** which are of minimum cardinality, so hypotheses containing an "extra" unmatched **Gobs** will be effectively invisible to user until needed.

• Try to **match Gobs** with each e-graph in each hypothesis. This handles the case where **obs-graph** is *related* to a previous observation.

• The current conclusion corresponds to the *disjunction* of all hypotheses of minimum size.

Sometimes a member of **Hypoths** may contain an undetected inconsistency. In this case the answer returned by **minimum-Hypoths** may be incorrect: for example, it may return an inconsistent e-graph which groups all the observations as part of a single End event, rather than as part of two or more distinct End events. However, if this inconsistency is detected when later observations are being incorporated, the system can usually recover. In the inner loop of the algorithm **group-observations**, the new e-graph **Gobs** will fail to match against every member of the inconsistent hypothesis **H**. Although **Hypoths** is updated to contain **H** ∪ **{Gobs}**, if any matches against other hypotheses containing the same or few number of e-graphs *do* succeed, **H** ∪ **{Gobs}** will not be returned by **minimum-Hypoths**. Furthermore, it would not be difficult to modify **group-observations** to explicitly eliminate **H** ∪ **{Gobs}** from **Hypoths** in this special case.

Some of the most intimidating complexity results arise from this algorithm. In the worst case there could be $O(2^n)$ consistent ways of grouping n observations, and Hypoths could contain that number of hypotheses. In practice it appears that stronger assumptions than simply minimizing the number of End events are needed. One such stronger assumption would be that the current observation is part of the End event whose previous steps were most recently observed— or if that is inconsistent, then the next more recent End event, and so on. This assumption limits Hypoths to size O(n). All previous plan recognition systems implement some version of this stronger assumption.

6. Conclusions & Caveats

This chapter has developed a framework for plan recognition that includes a proof theory, a model theory, and a set of algorithms. It would not be an exaggeration to say that the most difficult task was to define the rather amorphous problem of "plan recognition" at the most abstract level. Similar algorithms for plan recognition problems have been hashed over for years, since the early work of Schmidt and Genesereth. The formal theory we have developed suggests what some of these algorithms are algorithms *for*.

The theory is extremely general. It does not assume that there is a single plan underway that can be uniquely identified from the first input, nor that the sequence of observations is complete, nor that all the steps in a plan are linearly ordered. We know of no other implemented plan recognition system that handles arbitrary temporal relations between steps. On the other hand, there are some limitations inherent in our representation of plans. In particular, the current framework does not explicitly represent propositional attitudes, such as goals or beliefs. The use of quantification and disjunction in the event hierarchy is restricted, although some of these limitations can be easily circumvented. For example, existential quantifiers cannot appear in the axioms which make up the event hierarchy, but in most cases it should be possible to use function symbols instead. Disjunctions cannot appear in the body (the right-hand side) of a decomposition axiom. One way around this limitation is to create a new event event type to stand for the disjunctive formula, and then assert that each disjunct is a different specialization of the new type.

Another expressive limitation of the theory revolves around the whole idea of End events. We have used End events as a way to determine when "nothing more" needs to be explained. But in general it may prove difficult to define a set of End events (perhaps "stay alive" is the only one). Context must ultimately play a role in determine the scope of explanation.

It is important to note that the input to the algorithms presented here is still more expressively limited. For example, the algorithms do not handle theories in which an event type can have more than one direct abstraction (that is, multiple inheritance), and they do not explicitly handle "general axioms" which lie outside the event hierarchy. (The actual LISP implementation included some special purpose inference procedures; for example, to infer that certain pairs of predicates could not hold true of the same time period.)

The theory is limited in its ability to recognizing erroneous plans. We have assumed that all plans are internally consistent and that all acts are purposeful. Yet real people frequently make planning errors and change their minds in midcourse. Some simple kinds of errors can be handled by introducing an End event called Error. For each observable action there is a specialization of Error that contains that action as its only component. Therefore every observation can be recognized as being part of some meaningful plan or simply an error.

Another serious limitation of the theory is the inability to recognize new plans whose types do not already appear in the recognizer's knowledge base. One might argue that plan recognition essentially deals with the recognition of stereotypical behavior, and the understanding of new plans is better treated as an advanced kind of learning.

In some domains the theory described in this chapter is simply too weak. Rather than inferring the disjunction of all the plans that could explain the observations, the recognizer may need to know the most likely such plan. Nothing in this theory contradicts the laws of probability, and it should be possible to extend the theory with quantitative measures.

A more philosophical problem is the whole issue of what serves as primitive input to the recognition system. Throughout this chapter we have 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 — does not 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 — translated 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. 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.

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.

Charniak, Eugene (1983) Unpublished talk presented at the University of Rochester.

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

Charniak, Eugene & Robert Goldman (1989) A Semantics for Probabilistic Quantifier-Free First-Order Languages, with Particular Application to Story Understanding, *Proceedings of IJCAI-89*, pg. 1074, Morgan-Kaufmann.

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

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.

Doyle, Sir Authur Conan (1890) The Sign of Four, chapter 6.

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.

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

Goodman, Brad & Diane Litman (1990) Plan Recognition for Intelligent Interfaces, *Proceedings of the Sixth IEEE Conference on Artificial Intelligence Applications*, Santa Barbara, CA.

Haas, Andrew R. (1987) The Case for Domain-Specific Frame Axioms, in *The Frame Problem in Artificial Intelligence, Proceedings of the 1987 Workshop,* Lawrence, Kansas, F. M. Brown, ed., Morgan Kaufmann.

Hanks, Steven & Drew McDermott (1986) Default Reasoning, Nonmonotonic Logics, and the Frame Problem, *Proceedings of AAAI-86,* Morgan Kaufmann.

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.

Hobbs, J., & M. Stickel (1988) ???????????

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

Ladkin, Peter & R. Maddux (1988) Representation and Reasoning with Convex Time Intervals, Technical Report KES.U.88.2, Kestrel Institution, Palo Alto, CA.

Litman, Diane & James Allen (1987) A Plan Recognition Model for Subdialogues in Conversation, *Cognitive Science,* vol 11, pp. 163–200.

McCarthy, John (1980) Circumscription -- A Form of Non-Monotonic Reasoning, *Artificial Intelligence*, vol. 13, pp. 27-39.

McCarthy, J. & P. Hayes (1969) Some Philosophical Questions from the Standpoint of Artificial Intelligence, *Machine Intelligence 4*, Edinburg University Press, Edinburg, UK.

Minsky, Marvin (1975) A Framework for Representing Knowledge, in *The Psychology of Computer Vision*, McGraw-Hill, New York.

Moore, Robert (1977) Reasoning about Knowledge and Action, *Proceedings of IJCAI-77,* Morgan Kaufmann.

Pednault, E. P. D. (1988) Synthesizing Plans that Contain Actions with Context-dependent Effects, *Computational Intelligence*, vol 4, no 4, pp 356–372.

Pelavin, Richard. (1990) Planning with Concurrent Actions and External Events, in *Formal Models of Plan Reasoning,* by J. Allen, H. Kautz, R. Pelavin, & J. Tenenberg, Morgan Kaufman

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 (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.

Reiter, R. (1987) A Theory of Diagnosis from First Principles, *Artificial Intelligence,* vol. 32, no. 1, pg. 57.

Schank, R. (1975) *Conceptual Information Processing*, American Elsevier, New York.

Schmidt, C.F., N.S. Sridharan, & J.L. Goodson (1978) The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence, *Artificial Intelligence*, vol.11, pp. 45-83.

Schubert, Lenhart K. (1989) Monotonic Solution of the Frame Problem in the Situation Calculus: an Efficient Method for Worlds with Fully Specified Actions, in *Knowledge Representation and Defeasible Reasoning,* H. Kyburg, R. Loui, & G. Carlson, eds., Kluwer Press.

Tenenberg, Josh D. (1990) Abstraction in Planning, in *Formal Models of Plan Reasoning,* by J. Allen, H. Kautz, R. Pelavin, & J. Tenenberg, Morgan Kaufman

Wilensky, Robert (1983) *Planning and Understanding*, Addison-Wesley, Reading, MA.