GENERALIZED PLAN RECOGNITION

Henry A. Kautz
James F. Allen

Department of Computer Science
University of Rochester
Rochester, New York 14627

ABSTRACT

This paper outlines a new theory of plan recognition that is significantly more powerful than previous approaches. Concurrent actions, shared steps between actions, and disjunctive information are all handled. The theory allows one to draw conclusions based on the class of possible plans being performed, rather than having to prematurely commit to a single interpretation. The theory employs circumscription to transform a first-order theory of action into an action taxonomy, which can be used to logically deduce the complex action(s) an agent is performing.

1. Introduction

A central issue in Artificial Intelligence is the representation of actions and plans. One of the major modes of reasoning about actions is called plan recognition, in which a set of observed or described actions is explained by constructing a plan that contains them. Such techniques are useful in many areas, including story understanding, discourse modeling, strategic planning and modeling naive psychology. In story understanding, for example, the plans of the characters must be recognized from the described actions in order to answer questions based on the story. In strategic planning, the planner may need to recognize the plans of another agent in order to interact (co-operatively or competitively) with that agent.

Unlike planning, which often can be viewed as purely hypothetical reasoning (i.e. if I did A, then P would be true), plan recognition models must be able to represent actual events that have happened as well as proposing hypothetical explanations of actions. In addition, plan recognition inherently involves more uncertainty than in planning. Whereas in planning, one is interested in finding any plan that achieves the desired goal, in plan recognition, one must attempt to recognize the particular plan that another agent is performing. Previous plan recognition models, as we shall see, have been unable to deal with this form of uncertainty in any significant way.

A truly useful plan recognition system must, besides being well-defined, be able to handle various forms of uncertainty. In particular, often a given set of observed actions may not uniquely identify a particular plan, yet many important conclusions can still be drawn and predictions about future actions can still be made.

For example, if we observe a person in a house picking up the car keys, we should be able to infer that they are going to leave the house to go to the car, even though we cannot tell if they plan to drive somewhere, or simply to put the car in the garage. On the basis of this information, we might ask the person to take the garbage out when they leave. To accomplish this, a system cannot wait until a single plan is uniquely identified before drawing any conclusions. On the other hand, a plan recognizer should not prematurely jump to conclusions either. We do not want to handle the above example by simply inferring that the person is going to put the car into the garage when there is no evidence to support this interpretation over the one involving driving to the store.

This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, and the Air Force Office of Scientific Research, under Contract No. F30602-85-C-0008, and the National Science Foundation under grant DCR-8502481.

In addition, a useful plan recognizer in many contexts cannot make simplistic assumptions about the temporal ordering of the observations either. In story understanding, for example, the actions might not be described in the actual order that they occurred. In many domains, we must also allow actions to occur simultaneously with each other, or allow the temporal ordering not to be known at all. Finally, we must allow the possibility that an action may be executed as part of two independent plans. We might, for example, make enough pasta one night in order to prepare two separate meals over the next two days.

None of the previous models of plan recognition can handle more than a few of these situations in any general way. Part of the problem is that none of the frameworks have used a rich enough temporal model to support reasoning about temporally complex situations. But even if we were able to extend each framework with a general temporal reasoning facility, there would still be problems that remain. Let us consider three of the major approaches briefly, and discuss these other problems.

The explanation-based approaches, outlined formally by [Cha85] all attempt to explain a set of observations by finding a set of assumptions that entails the observations. The problem with this is that there may be many such sets of assumptions that will have this property, and the theory says nothing as to how to select among them. In practice, systems based on this framework (e.g. [Wil83]) will over-commit, and select the first explanation found, even though it is not uniquely identified by the observations. In addition, they are not able to handle disjunctive information.

The approaches based on parsing (e.g. [Huf82, Sid81]) view actions as sequences of subactions and essentially model this knowledge as a context-free rule in an "action grammar". The primitive (i.e. non-decomposable) actions in the framework are the terminal symbols in the grammar. The observations are then treated as input to the parser and it attempts to derive a parse tree to explain the observations. A system based on this model would suffer from the problem of over-commitment unless it generates the set of possible explanations (i.e. all possible parses). While some interesting temporal aspects in combining plans can be handled by using more powerful grammars such as shuffle grammars, each individual plan can only be modelled as a sequence of actions. In addition, every step of a plan must be observed -- there is no capability for partial observation. It is not clear how more temporally complex plans could be modelled, such as those involving simultaneous actions, or how a single action could be viewed as being part of multiple plans.

The final approach to be discussed is based on the concept of "likely" inference (e.g. [All83, Pol84]). In these systems a set of rules is used of the form: "If one observes act A, then it may be that it is part of act B". Such rules outline a search space of actions that produces plans that include the observations. In practice, the control of this search is hidden in a set of heuristics and thus is hard to define precisely. It is also difficult to attach a semantics to such rules as the one above. This rule does not mean that if we observe A, then it is probable that B is being executed, or even that it is possible that B is being executed. The rule is valid even in situations where it is impossible for B to be in execution. These issues are decided entirely by the heuristics. As such, it is hard to make precise claims as to the power of this formalism.

In this paper we outline a new theory of plan recognition that is significantly more powerful that these previous approaches in that it can handle many of the above issues in an intuitively satisfying way. Furthermore, there are no restrictions on the temporal relationships between the observations. Another important result is that the implicit assumptions that appear to underly all plan recognition processes are made explicit and precisely defined within a formal theory of action. Given these assumptions and a specific body of knowledge about the possible actions and plans to be considered, this theory will give us the strongest set of conclusions that can be made given a set of observations. As such, this work lays a firm base for future work in plan recognition.

Several problems often associated with plan recognition are not considered in the current approach, however. In particular, beyond some simple simplicity assumptions, the framework does not distinguish between a priori likely and non-likely plans. Each logically possible explanation, given the assumptions, is treated equally within the theory. It also can only recognize plans that are constructed out of the initial library of actions defined for a particular domain. As a result, novel situations that arise from a combination of existing plans may be recognized, but other situations that require generalization techniques, or reasoning by analogy cannot be recognized.

2. A New View of Plan Recognition

It is not necessary to abandon logic, or to enter the depths of probabilistic inference, in order to handle the problematic cases of plan recognition described above. Instead, we propose that plan recognition be viewed as ordinary deductive inference, based on a set of observations, an action taxonomy, and one or more simplicity constraints.

An action taxonomy is an exhaustive description of the ways in which actions can be performed, and the ways in which any action can be used as step of a more complex action. Because the taxonomy is complete, one can infer the disjunction of the set of possible plans which contain the observations, and then reason by cases to reduce this disjunction.

An action taxonomy is obtained by applying two closed-world assumptions to an axiomatization of an action hierarchy. The first assumption states that the known ways of performing an action are the only ways of performing that action. The assumption is actually a bit more general, in that it states the known ways of *specializing* an action are the only ways. Each time an abstract action is specialized, more is known about how to perform it. For example, because the action type "throw" specializes the action type "transfer location", we can think of throwing as a way to transfer location.

The second assumption states that all actions are purposeful, and that all the possible reasons for performing an action are known. This assumption is realized by stating that if an action A occurs, and P is the set of more complex actions in which A occurs as a substep, then *some* member of P also occurs.

These assumptions can be stated using McCarthy's circumscription scheme. The action hierarchy is transformed by first circumscribing the ways of specializing an act, and then circumscribing the ways of using an act. The precise formulation of this operation is described in section 6 below.

The simplicity constraints become important when we need to recognize a plan which integrates several observations. The top of the action taxonomy contains actions which are done for their own sake, rather than as steps of more complex actions. When several actions are observed, it is often a good heuristic to assume that the observations are all part of the same top level act, rather than each being a step of an independent top level act. The simplicity constraint which we will use asserts that as few top level actions occur as possible. The simplicity constraint can be represented by a formula of second-order logic which is similar to the circumscription formula. In any particular case the constraint is instantiated as a first-order formula which asserts "there are no more than n top level acts", which n is a particular constant choosen to be as small as possible, and still allow the instantiated constraint to be consistent with the observations and taxonomy.

While one can imagine many other heuristic rules for choosing between interpretations of a set of observed actions, the few given here cover a great many common cases, and seem to capture the "obvious" inferences one might make. More fine grained plan recognition tasks (such as strategic planning) would probably require some sort of quantitative reasoning.

3. Representing Action

The scheme just described requires a representation of action that includes:

--the ability to assert that an action actually occurred at a time;

--a specialization hierarchy;

--a decomposition (substep) hierarchy.

Action instances are individuals which occur in the world, and are classified by action types. The example domain is the world of cooking, which includes a very rich action hierarchy, as well as a token bit of block stacking. (See figure 1.) The broad arrows indicate that one action type is a specialization of another action type, whereas the thin arrows indicates the decomposition of an action into subactions. We will see how to represent this information in logic presently. The diagram does not indicate other conditions and constraints which are also part of an action decomposition. Instances of action types are also not shown. We introduce particular instances of actions using formulas such as

$$\#(\text{E9}, \text{makePastaDish})$$

to mean that E9 is a real action instance of type MakePastaDish. (The symbol # is the "occurs" predicate.) The structure of a particular action can be specified by a set of role functions. In particular, the function T applied to an action instance returns the interval of time over which the action instance occurs. Other roles of an action can also be represented by functions; e.g., Agent(E9) could be the agent causing the action, and Result(E9) could be the particular meal produced by E9. (For simplicity we will assume in this paper that all actions are performed by the same agent.) To record the observation of the agent making a pasta dish at time I7, one would assert:

$$\exists\, e\,.\ \#(e, \text{makePastaDish})\ \&\ T(e) = I7$$

Action types need not all be constants, as they are here; often it is useful to use functions to construct types, such as Move(x,y). For simplicity, all the actions used in the examples in this paper use only constant action types.

Action specialization is easy to represent in this scheme. In the cooking world, the act of making a pasta dish specializes the act of preparing a meal, which in turn specializes the class of top level acts. Specialization statements are simply universally-quantified implications. For example, part of the hierarchy in figure 1 is represented by the following axioms:

[1] $\forall\, e\,.\ \#(e, \text{PrepareMeal}) \supset \#(e, \text{TopLevelAct})$

[2] $\forall\, e\,.\ \#(e, \text{MakePastaDish}) \supset \#(e, \text{PrepareMeal})$

[3] $\forall\, e\,.\ \#(e, \text{MakeFettuciniMarinara}) \supset \#(e, \text{MakePastaDish})$

[4] $\forall\, e\,.\ \#(e, \text{MakeFettucini}) \supset \#(e, \text{MakeNoodles})$

[5] $\forall$ e . #(e, MakeSpaghetti) $\supset$ #(e, MakeNoodles)

[6] $\forall$ e . #(e, MakeChickenMarinara) $\supset$
#(e, MakeMeatDish)

The first statement, for example, means that any action instance which is a PrepareMeal is also a TopLevelAct.

The decomposition hierarchy is represented by implications which assert necessary (and perhaps sufficient) conditions for an action instance to occur. This may include the fact that some number of subactions occur, and that various facts hold at various times [All84]. These facts include the preconditions and effects of the action, as well as various constraints on the temporal relationships of the subactions [All83a].

For the level of analysis in the present paper, we do not need to distinguish the minimal necessary set of conditions for an action to occur, from a larger set which may include facts which could be deduced from the components of the act. It is also convenient to eliminate some existentially quantified variables by introducing a function S(i,e) which names the i-th subaction (if any) of action e. (The actual numbers are not important; any constant symbols can be used.) For example, the makePastaDish action is decomposed as follows:

```
[7]  ∀ e . #(e, MakePastaDish) ⊃
         ∃ tn . #(S(1,e), MakeNoodles) &
               #(S(2,e), Boil) &
               #(S(3,e), MakeSauce) &
               Object(S(2,e)) = Result(S(1,e)) &
               hold(noodle(Result(S(1,e)), tn) &
               overlap(T(S(1,e)), tn) &
               during(T(S(2,e)), tn)
```

This states that every instance of MakePastaDish consists of (at least) three steps: making noodles, boiling them, and making a sauce. The result of making noodles is an object which is (naturally) of type noodle, for some period of time which follows on the heels of the making. (Presumably the noodles cease being noodles after they are eaten.) Furthermore, the boiling action must occur while the noodles are, in fact, noodles. A complete decomposition of MakePastaDish would contain other facts, such that result of the MakeSauce act must be combined at some point with the noodles, after they are boiled.

The constraint that all the subactions of an action occur during the time of the action is expressed for all acts by the axiom:

[8] $\forall$ i,e . during(T(S(i,e)), T(e))

It is important to note that a decomposable action can still be further specialized. For example, the action type MakeFettuciniMarinara specializes MakePastaDish and adds additional constraints on the above definition. In particular, the type of noodles made in step 1 must be fettucini, while the sauce made in step 3 must be marinara sauce.

A final component of the action hierarchy are axioms which state action-type disjointedness. Such axioms are expressed with the connective "not and", written as ∇:

[9] $\forall$ e . #(e,MakeFettuciniAlfredo)
∇ #(e,MakeFettuciniMarinara)

This simply says that a particular action cannot be *both* an instance of making fettucini Alfredo *and* an instance of making fettucini Marinara. Disjointedness axioms can be compactly represented and used in resolution-based inference using techniques adapted from [Ten86].

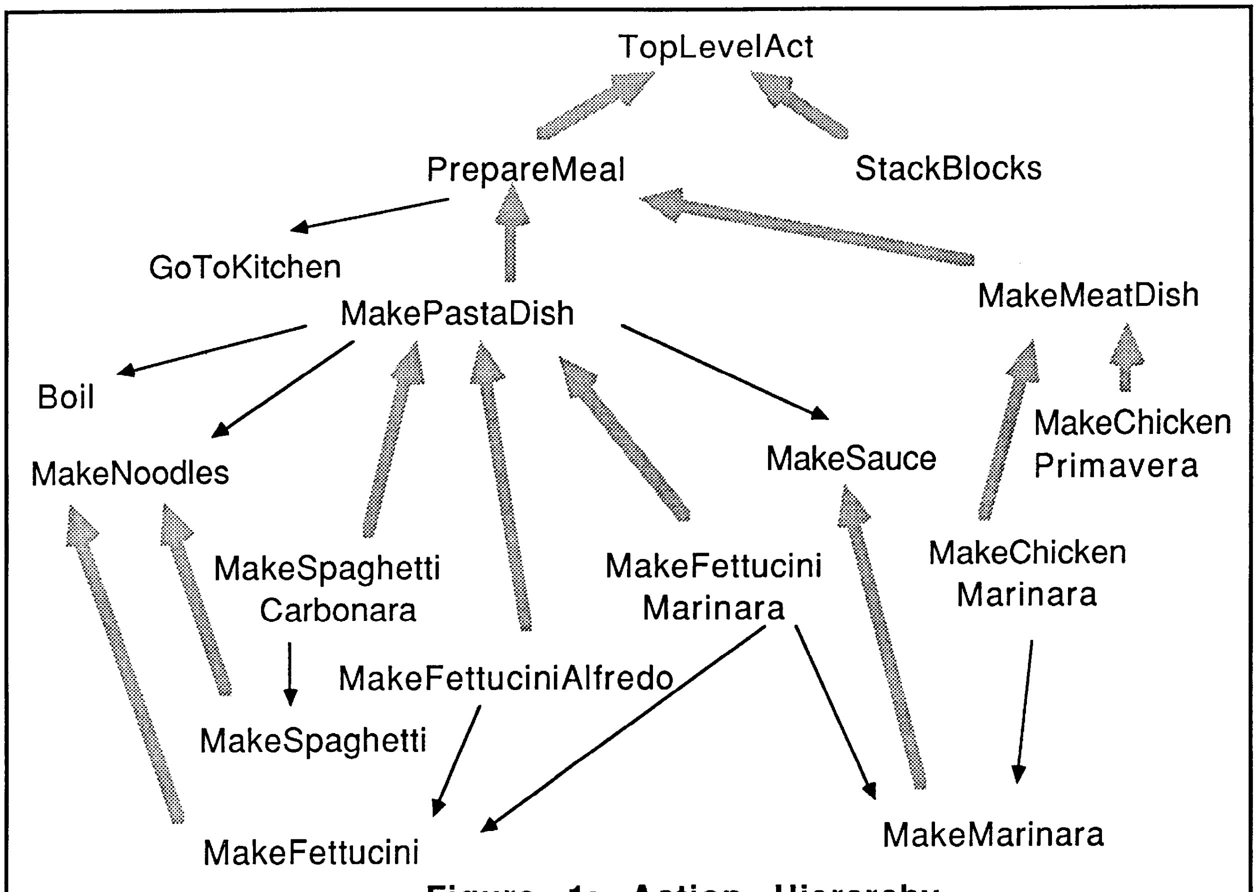

Figure 1: Action Hierarchy

4. Creating the Taxonomy

The assumptions necessary for plan recognition can now be specified more precisely by considering the full action hierarchy presented in figure 1. Let KB1 be the set of axioms schematically represented by the graph, including the axioms mentioned above. KB1 will be transformed into a taxonomy by applying the completeness assumptions discussed above. The result of the first assumption (all ways of specializing an action are known) is the database KB2, which includes all of KB1, together with statements which assert specialization completeness. These include the following, where the symbol ⊕ is the connective "exclusive or".

[10] ∀ e . #(e, TopLevelAct) ⊃
#(e, PrepareMeal) ⊕
#(e, StackBlocks)

[11] ∀ e . #(e, PrepareMeal) ⊃
#(e, MakePastaDish) ⊕
#(e, MakeMeatDish)

[12] ∀ e . #(e, MakePastaDish) ⊃
#(e, MakeFettuciniMarinara) ⊕
#(e, MakeFettuciniAlfredo) ⊕
#(e, MakeSpaghettiCarbonara)

[13] ∀ e . #(e, MakeMeatDish) ⊃
#(e, MakeChickenMarinara) ⊕
#(e, MakeChickenPrimavera)

[14] ∀ e . #(e, MakeNoodles) ⊃
#(e, MakeFettucini) ⊕
#(e, MakeSpaghetti)

These state that every top level action is either a case of preparing a meal, or of stacking blocks; that every meal preparation is either a case of making a pasta dish or making a meat dish; and so on, all the way down to particular, basic types of meals. Not all actions, of course, are specializations of TopLevelAct. For example, axiom [14] states that every MakeNoodles can be further classified as a MakeFettucini or as a MakeSpaghetti, but it is not the case that any MakeNoodles can be classified as a TopLevelAct.

The second assumption asserts that the given decompositions are the only decompositions. KB2 is transformed to the final taxonomy KB3, which includes all of KB2, as well as:

[15] ∀ e . #(e, MakeNoodles) ⊃
∃ a . #(a, MakePastaDish) & e = S(1,a)

[16] ∀ e . #(e, MakeMarinara) ⊃
∃ a . [#(a, MakeFettuciniMarinara) & e = S(3,a)] ∨
[#(a, MakeChickenMarinara) & e = S(3,a)]

[17] ∀ e . #(e, MakeFettucini) ⊃
∃ a . [#(a, MakeFettuciniMarinara) & e = S(1,a)] ∨
[#(a, MakeFettuciniAlfredo) & e = S(1,a)]

Axiom [15] states that whenever an instance of MakeNoodles occurs, then it must be the case that some instance of MakePastaDish occurs. Furthermore, the MakeNoodles act which is required as a substep of the MakePastaDish is in fact the given instance of MakeNoodles. Cases like this, where an action can only be used in one possible super-action, usually occur at a high level of action abstraction. It is more common for many uses for an action to occur in the taxonomy. The given hierarchy has two distinct uses for the action MakeMarinara, and this is captured in axiom [16]. From the fact that the agent is making Marinara sauce, one is justified in concluding that an action instance will occur which is *either* of type MakeFettuciniMarinara or of type MakeChickenMarinara.

All these transformations can easily be performed automatically given an action taxonomy of the form described in the previous section. The formal basis for these transformations is described in section 6.

5. Recognition Examples

We are now ready to work through some examples of plan recognition using the cooking taxonomy. In the steps that follow, existentially-quantified variables will be replaced by fresh constants. Constants introduced for observed action instances begin with E, and those for deduced action instances being with K. Simple cases typical of standard plan recognition are easily accounted for. In this section, we shall consider an extended example demonstrating some more problematic cases.

Let the first observation be disjunctive: the agent is observed to be either making fettucini or making spaghetti, but we cannot tell which. This is still enough information to make predictions about future actions. The observation is:

[18] #(E1, MakeFettucini) ∨ #(E1, MakeSpaghetti)

The abstraction axioms let us infer up the hierarchy:

[19]	#(E1, MakeNoodles)	abstraction axioms [4], [5]
[20]	#(K01, MakePastaDish)	decomposition axiom [15], and existential instantiation
[21]	#(K01, PrepareMeal)	abstraction axiom [2]
[22]	#(K01, TopLevelAct)	abstraction axiom [1]

Statement [20] together with [7] lets us make a prediction about the future: a Boil will occur:

[23] #(S(2,K01), boil) &
Object(S(2,K01)) = Result(E1) &
after(T(S(2,K01)),T(E1))

Thus even though the particular plan the agent is performing cannot be exactly identified, specific predictions about future activities can still be made.

The previous step showed how one could reason from a disjunctive observation up the abstraction hierarchy to a non-disjunctive conclusion. With the next observation, we see that going up the decomposition hierarchy from a non-disjunctive hierarchy can lead to a disjunctive conclusion. Suppose the next observation is:

[24] #(E3, MakeMarinara)

Applying axiom [16], which was created by the second completeness assumption, leads to the conclusion:

[25] #(K02, MakeFettuciniMarinara) ∨
#(K02, MakeChickenMarinara)

The abstraction hierarchy can again be used to collapse this disjunction:

[26] #(K02, MakePastaDish) ∨ #(K02, MakeMeatDish)

[27] #(K02, PrepareMeal)

[28] #(K02, TopLevelAct)

At this point the simplicity constraint comes into play. The strongest form of the constraint, that there is only one top level action in progress, is tried first:

[29] ∀ e1,e2 . #(e1, TopLevelAct) &
#(e2, TopLevelAct) ⊃ e1 = e2

Together with [22] and [28], this implies:

[30] K01 = K02

Substitution of equals yields:

[31] #(K02, MakePastaDish)

One of the disjointedness axioms from the original action hierarchy is:

[32] $\forall$ e . #(e, MakePastaDish) ∇ #(e, MakeMeatDish)

Statements [30], [31], and [6] let us deduce:

[33] $\neg$#(K02, MakeMeatDish)

[34] $\neg$#(K02, MakeChickenMarinara)

Finally, [34] and [25] let us conclude that only one plan, which contains both observations, is occurring:

[35] #(K02, MakeFettuciniMarinara)

Temporal constraints did not play a role in these examples, as they do in more complicated cases. For example, observations need not be received in the order in which the observed events occurred, or actions might be observed in an order where the violation of temporal constraints can allow the system to reject hypotheses. For example, if a Boil act at an unknown time were input, the system would assume that it was the boil act of the (already deduced) MakePastaDish act. If the Boil were constrained to occur before the initial MakeNoodles, then the strong simplicity constraint (and all deductions based upon it) would have to withdrawn, and two distinct top level actions postulated.

Different top level actions (or any actions, in fact) can share subactions, if such sharing is permitted by the particular domain axioms. For example, suppose every PrepareMeal action begins with GotoKitchen, and the agent is constrained to remain in the kitchen for the duration of the act. If the agent is observed performing two different instances of PrepareMeal, and is further observed to remain in the kitchen for an interval which intersects the time of both actions, then we can deduce that both PrepareMeal actions share the same initial step. This example shows the importance of including observations that certain states hold over an interval. Without the fact that the agent remained in the kitchen, one could not conclude that the two PrepareMeal actions share a step, since it would be possible that the agent left the kitchen and then returned.

6. Closing the Action Hierarchy

Our formulation of plan recognition is based on an explicitly asserting that the action hierarchy (also commonly called the "plan library") is complete. While the transformation of the hierarchy into a taxonomy can be automated, some details of the process are not obvious. It is not correct to simply apply predicate completion, in the style of [Cla78]. For example, even if action A is the only act which is stated to contain act B as a substep, it may not be correct to add the statement

$\forall$ e . #(e,B) $\supset$ $\exists$ e1 . #(e1,A)

if there is some act C which either specializes or generalizes B, and is used in an action other than A. For example, in our action hierarchy, the only *explicit* mention of MakeSauce appears in the decomposition of MakePastaDish. But the taxonomy should *not* contain the statement

$\forall$ e . #(e, MakeSauce) $\supset$
    $\exists$ a . #(a, MakePastaDish) & e = S(3,a)

because a particular instance of MakeSauce may also be an instance of MakeMarinara, and occur in the decomposition of the action MakeChickenMarinara. Only the weaker statement

$\forall$ e . #(e, MakeSauce) $\supset$
    $\exists$ a . [#(a, MakePastaDish) & e = S(3,a)] $\vee$
        [#(a, MakeChickenMarinara) & e = S(3,a)]

is justified. It *would* be correct, however, to infer from an observation of MakeSauce which is known *not* to be an instance of MakeMarinara that MakePastaDish occurs.

We would like, therefore, a clear understanding of the semantics of closing the hierarchy. McCarthy's notion of minimal entailment and circumscription [McC85] provides a semantic and proof-theoretic model of the process. The implementation described in section 7 can be viewed as an efficient means for performing the sanctioned inferences.

There is not space here to fully explain how and why the circumscription works; more details appear in [Kau85]. A familiarity with the technical vocabulary of circumscription is probably needed to make complete sense of the rest of this section. Roughly, circumscribing a predicate minimizes its extension. Predicates whose extensions are allowed to change during the minimization are said to *vary*. All other predicates are are called *parameters* to the circumscription. In anthropomorphic terms, the circumscribed predicate is trying to shrink, but is constrained by the parameters, who can choose to take on any values allowed by the original axiomatization. For example, circumscribing the predicate p in the theory:

$\forall$ x . p(x) $\equiv$ q(x)

where q acts as a parameter does *nothing*, because q can "force" the extension of p to be arbitrarily large. On the other hand, if q *varies* during the circumscription, then the circumscribed theory entails that the extension of p is *empty*.

As demonstrated above, the first assumption states that the known ways of specializing an action are all the ways. Let us call all action types which are not further specialized *basic*. Then another way of putting the assumption is to say that the "occurs" predicate, #, holds of an instance and an *abstract* action type only if it *has to*, because # holds of that instance and a basic action type which specializes the abstract type. So what we want is to circumscribe that part of # which applies to non-basic action types. This can be done by adding a predicate which is true of all basic action instances, and to let this predicate act as a parameter during the circumscription of #. In our example domain, the following two statements, which we will call Ψ, define such a predicate.

Let Ψ = { $\forall$ x . basic(x) $\equiv$
    x = MakeFettuciniMarinara $\vee$
    x = Boil $\vee$ ... ,

$\forall$ e,x . #basic(e,x) $\equiv$ #(e,x) & basic(x) }

KB1 is the set of axioms which make up the original action hierarchy. KB2 is then defined to be the circumscription of # relative to KB1 together with Ψ, where all other predicates (including #basic) act as parameters.

KB2 = Circumscribe(KB1 $\cup$ Ψ, #)

Interestingly, the process works even if there are many levels of abstraction hierarchy above the level of basic actions. Note that basic actions (such as MakeFettuciniMarinara) may be decomposable, even though they are not further specialized.

The second assumption states that any non-top-level action occurs only as part of the decomposition of some top-level action. Therefore we want to circumscribe that part of # which applies to non-top-level actions. This can be done by adding a predicate to KB2 which is true of all top-level action instances, and circumscribing # again. The predicate #basic added above must be allowed to vary in the circumscription.

Let Φ = { $\forall$ e . #toplevel(e) $\supset$ #(e,TopLevelAct) }

KB3 = Circumscribe(KB2 $\cup$ Φ, #, #basic)

As before, the process can percolate though many levels of the action decomposition hierarchy. Note that the concepts basic action and top-level action are not antonyms; for example, the type MakeFettuciniMarinara is basic (not specializable), yet any instance of it is also an instance of TopLevelAct.

Circumscription cannot be used to express the simplicity constraint. Instead, one must minimize the *cardinality* of the extension of #, after the observations are recorded. [Kau85] describes the cardinality-minimization operator, which is similar, but more powerful than, the circumscription operator.

7. Implementation Considerations

The formal theory described here has given a precise semantics to the plan recognition reasoning process by specifying a set of axioms from which all desired conclusions may be derived deductively. Although no universally-applicable methods are known for automating circumscription, by placing reasonable restrictions on the form of the action hierarchy axioms, we can devise a special-purpose algorithm for computing the circumscriptions. As a result, in theory we could simply run a general purpose theorem prover given the resulting axioms to prove any particular (valid) conclusion. In practice, since we often don't have a specific question to ask beyond "what is the agent's goal?" or "what will happen next?", it is considerably more useful to design a specialized forward chaining reasoning process that essentially embodies a particular inference strategy over these axioms.

We are in the process of constructing such a specialized reasoner. The algorithm divides into two components: the preprocessing stage and the forward-chaining stage. The preprocessing stage is done once for any given domain. The two completeness assumptions from in the previous section are realized by circumscribing the action hierarchy. The result of the circumscription can be viewed as an enormously long logical formula, but is quite compactly represented by a graph structure.

The forward-chaining stage begins when observations are received. This stage incorporates the assumption that as few top-level acts as possible are occurring. As each observation is received, the system chains up both the abstraction and decomposition hierarchies, until a top-level action is reached. The intermediate steps may include many disjunctive statements. The action hierarchy is used as a control graph to direct and limit this disjunctive reasoning. After more than one observation arrives, the system will have derived two or more (existentially instantiated) constants which refer to top-level actions. The simplicity assumption is applied, by adding a statement that some subsets of these constants must be equal. Exclusive-or reasoning now propagates down the hierarchy, deriving a more restrictive set of assertions about the top-level acts and their subacts. If an inconsistency is detected, then the number of top-level acts is incremented, and the system backtracks to the point at which the simplicity assumption was applied.

This description of the implementation is admittedly sketchy. Many more details, including how the temporal constraint propagation system integrates with the forward-chaining reasoner, will appear in a forthcoming technical report.

8. Future Work

Future work involves completing the theoretical foundation, and building a test implementation.

The theoretical work includes a formal specification of the form of the action taxonomy so that its circumscription can always be effectively computed. Theorems guaranteeing the consistency and intuitive correctness of the circumscription will be completed.

More complex temporal interactions between simultaneously occurring actions will be investigated. We will show how the framework handles more complicated examples involving step-sharing and observations received out of temporal order (e.g. mystery stories). It will probably be necessary to develop a slightly more sophisticated simplicity constraint. Rather than stating that as few top-level actions occur as possible, it is more realistic to state that as few top-level actions as possible are occurring at any one time. In addition, observations of *non*-occurrences of events (e.g. the agent did *not* boil water) are an important source of information in plan recognition. Non-occurrences integrate nicely into our framework.

Many of the subsystems that are used by the plan recognizer (such as a temporal reasoner [All83a], and a lisp-based theorem prover which handles equality [All84a]) have been developed in previous work at Rochester, and construction of the complete implementation is under way.

References

All83. James F. Allen, "Recognizing Intentions from Natural Language Utterances," in *Computational Models of Discourse*, ed. M. Brady, MITP, 1983.

All83a. James F. Allen, "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM*, no. 26, pp. 832-843, Nov 1983.

All84. James F. Allen, "Towards a General Theory of Action and Time," *Artificial Intelligence*, vol. 23, no. 2, pp. 123-154, July 1984.

Cha85. Eugene Charniak and Drew McDermott, *Introduction to Artificial Intelligence*, Addison Wesley, Reading, MA, 1985.

Cla78. K.L. Clark, "Negation as Failure," in *Logic and Databases*, ed. J. Minker, Plenum Press, New York, 1978.

Huf82. Karen Huff and Victor Lesser, "KNOWLEDGE-BASED COMMAND UNDERSTANDING: An Example for the Software Development Environment," Technical Report 82-6, Computer and Information Sciences University of Massachusetts at Amherst, Amherst, MA, 1982.

Kau85. Henry A. Kautz, "Toward a Theory of Plan Recognition," TR162, Department of Computer Science, University of Rochester, July, 1985.

McC85. John McCarthy, "Applications of Circumscription to Formalizing Common Sense Knowledge," in *Proceedings from the Non-Monotonic Reasoning Workshop*, AAAI, Oct 1985.

Pol84. Martha E. Pollack, "Generating Expert Answers Through Goal Inference," PhD Thesis Proposal, Department of Computer Science, University of Pennsylvania, August 1983, January 1984. DRAFT

Sid81. Candace L. Sidner and David J. Israel, "Recognizing Intended Meaning and Speakers' Plans," *IJCAI*, 1981.

Ten86. Josh D. Tenenburg, "Reasoning Using Exclusion: An Extension of Clausal Form," TR 147, Department of Computer Science, University of Rochester, Jan 1986.

Wil83. Robert Wilensky, *Planning and Understanding*, Addison-Wesley, Reading, MA, 1983.

All84a. James F. Allen, Mark Giuliano, and Alan M. Frisch, "The HORNE Reasoning System," TR 126 Revised, Computer Science Department, University of Rochester, Sept 1984.