UNDERSTANDING AND ADVICE GIVING IN AQUA

Alexander E. Quilici
Michael G. Dyer
Margot Flowers

April 1986
CSD-860086

Understanding and Advice Giving in AQUA[1]

Alexander E. Quilici
Michael G. Dyer
Margot Flowers

Artificial Intelligence Laboratory
Computer Science Department
University of California
Los Angeles, CA 90024

Abstract

This paper examines the process of problem understanding and advice giving. The problems we consider are typical planning problems that novice computer users encounter. We describe a representational system for user planning problems, show how advice can be generated using a taxonomy of planning problems and associated heuristics for advice formulation, present heuristics that can be used to repair failed plans and to create new plans by combining existing plans in novel ways, and suggest a memory organization for planning knowledge that allows for efficient retrieval of relevant planning experiences.

The theory discussed in this paper is implemented in a computer program called AQUA. AQUA takes natural language descriptions of problems users are having with the UNIX[2] operating system and provides natural language advice that explains their failures and suggests solutions. AQUA is also able to create solutions for problems that it has not been presented with before.

1. Introduction

When novice computer users have trouble performing a task, they usually seek help from an expert instead of trying to wade through often poorly written, hard-to-understand computer manuals. They explain their situation to the expert and describe their previous failed attempts. The expert then explains these failures and provides a solution for the user's problem. This paper describes the theory behind AQUA, a computer program that models the process of problem understanding and advice giving of a typical computer consultant.

Our theoretical goal in building AQUA is to gain computational insight into how human experts organize and acquire their planning knowledge and then use it in problem understanding and advice giving. Our more practical goal is the design of computer programs capable of understanding and giving advice for the typical plan-oriented problems of novice computer users.

1.1 An I/O Example

AQUA's input is a natural language description of a problem a user is having with using the UNIX operating system. Its output is an English language solution for the user's problem, along with an explanation for any previous, failed attempts by the user. Here is an example user problem description and the resulting advice from AQUA:

---

[1]The work reported here was supported in part by a grant from the Lockheed Software Technology Center, Austin, Texas, for building intelligent tutoring and advice-giving systems.

[2]UNIX is a registered Trademark of Bell Laboratories.

STUBBORN FILE

USER: I tried to remove a file with the "rm" command. The file was not removed, and the error message was permission denied. I checked and I own the file. What's wrong?

AQUA: To remove a file, you need directory write permission. To remove a file, you do not need to own it.

What is involved in understanding this user problem description and then generating appropriate advice? To understand that the user has a problem, AQUA must realize that the user has (1) a *failed goal* of removing a particular file, (2) a *failed plan* of using **rm**[3] to get rid of that file, (3) a *satisfied goal* of verifying the file's ownership, and (4) a *hypothesis* that owning a file is a precondition for removing it.

Once AQUA has made these inferences and created a model of the user's problem, it can give advice by comparing the user's knowledge with its own, pointing out and explaining any differences. In this example, the differences lie in their respective beliefs about what the precondition for file removal is (the user hypothesizes that it is file ownership; AQUA knows that it is write permission on the directory containing the file), so AQUA tells the user what it knows to be the precondition and corrects the user's incorrect hypothesis.

1.2 The Theoretical Issues

We view advice giving as a process of memory search, guided by heuristics for problem understanding, advice generation, and plan creation. Thus, there are several important theoretical questions we must address in building AQUA.

(1) How do we understand someone else's problems?

Before we can give reasonable advice, we must understand the problem. From examining STUBBORN FILE, we can see that AQUA must be able to build a model of the user's problem that includes a model of the user's knowledge state. To do so, AQUA needs to be able to recognize the user's goals and plans, to detect when a plan has failed or a goal has been achieved, to find the causes for any stated outcomes, and to infer the motivations behind the user's actions.

(2) How do we generate good advice?

Once we understand what someone's problem is, we try to remember similar past experiences and their solutions, which are then used to generate advice. After AQUA understands what the user's problem is and has created a model of the user's knowledge state, it compares past experiences it is reminded of with the user's to determine what advice to give. Different types of problems require different advice, so AQUA must be able to determine what is good advice for a given situation.

(3) What planning knowledge is worth remembering and how is this planning knowledge organized in memory?

An expert in any domain clearly possesses a large amount of domain dependent knowledge, which must be accessed in a reasonable and efficient manner. AQUA must be able to organize and retrieve knowledge about plans, such as their uses, their outcomes, and their failures, in an efficient manner, even though there may be hundreds of different plans and planning episodes stored in memory. To do so, AQUA must index its planning knowledge so that when a situation requiring planning occurs, features of

[3]Throughout this paper the names of UNIX commands will appear in boldface.

that situation will remind it of earlier, similar planning episodes.

(4) How is planning knowledge acquired?

We become more expert through experience. We try plans, observe their results, and then use these experiences to create new plans. While planning, we make use of both successful and failed planning experiences. AQUA must possess this ability to acquire new plans, and must be able to add them to memory in a way so that they are retrieved when situations arise where they might be useful.

1.3 The Domain

AQUA's computer consulting domain provides a clean framework for studying the process of problem understanding and advice giving. By limiting ourselves to typical user planning problems, we avoid dealing with the complexities that we would have to contend with in creating a more general, DEAR ABBY-like advice-giving program, and can concentrate instead on the representation, organization, and acquisition of planning knowledge. Among the complexities we are avoiding are interpersonal relationships, social situations, and planners that can consider multiple agents in their planning. On the practical side, since the majority of problems novice computers have with using computer systems are planning problems, AQUA has the potential to make substantial contributions to more usable computer systems. Additional arguments for using the computer consulting domain to study planning problems can be found in [Wilensky 1982].

1.4 Overview of AQUA's Design

AQUA consists of several interrelated components, each of which will be discussed in detail in later sections of this paper:

(1) an integrated, demon-based parser [Dyer 1983] that parses the English description of a user problem into an appropriate conceptual representation that is stored in working memory.

(2) a dynamic episodic memory [Schank 1982] that holds AQUA's planning knowledge (memories of plan uses and their results, along with generalizations that have been made from them).

(3) a planner that can use planning heuristics, analogical reasoning, and memory search to create new plans [Wilensky 1983, Sacerdoti 1974, Carbonell 1983].

(4) an advice constructing component that determines the type of problem the user is having, decides on the appropriate advice, and then generates it into English.

Figure 1 shows the interrelationships between these components.

During parsing, information from the user's problem description (such as the user's goal or an event's outcome) is used in searching episodic memory for related planning experiences, which are then used in generating advice. When no appropriate planning experience can be found, the planner attempts to create new plans by using heuristics that combine and modify plans found in episodic memory. The advice constructor compares the user model in working memory with the planning experiences it was reminded of during parsing, classifies the user's problem, and then uses heuristics associated with each problem class to generate appropriate advice.

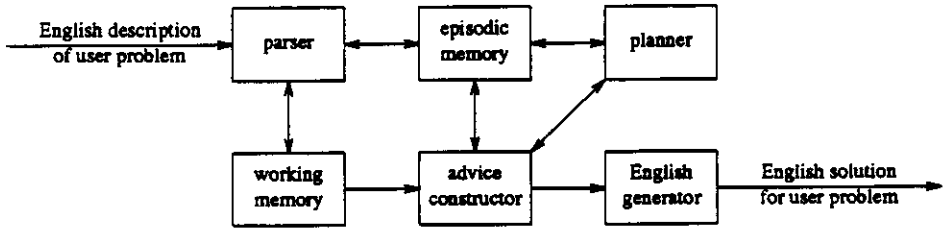

Figure 1: Diagram of AQUA's Components

1.5 Background and Related Work

AQUA can be thought of as a story understanding program, in which the stories involve the planning experiences of a single character trying to accomplish various tasks on a computer system. There is an ever-increasing body of research in the areas of natural language processing and story understanding. However, we give an overview only of those theories and programs that have had a significant effect on the theory and implementation of AQUA.

1.5.1 Semantics-Based Understanding

MARGIE [Schank 1975] was one of the earliest programs to emphasize semantics instead of syntax in understanding natural language text. MARGIE parsed English sentences into Conceptual Dependency (CD) [Schank 1972, Schank 1977] and then generated English paraphrases for each of the inferences (also represented in CD) that could be made from the conceptual representation.

CD represents knowledge about actions, consisting of several conceptual primitives and inference rules associated with each primitive. Each primitive has several slots for the character performing the act, the object modified by the act, the state before the act, the state after the act, and so on. As an example, INGEST is a CD primitive that represents placing substances inside the body, and is used to represent words like "eat," "smoke," and "drink." One rule associated with INGEST says that once something has been ingested, it is located inside the actor who ingested it.

MARGIE showed that using a small set of conceptual primitives has several benefits. First, sentences with similar meanings have similar representations, since they will be represented using the same primitive. Second, fewer inference rules are needed, since the inference rules can be associated with the primitive instead of with each of the words it represents. Finally, expectations associated with a primitive can be used to disambiguate words. In the phrase "John drank gin," it is the expectation associated with INGEST that selects the meaning of an alcoholic beverage, instead of a card game.

Unfortunately, MARGIE could not handle more than one sentence at a time. Because each CD has several possible inferences and there are inferences that can be made from each of these CD's, MARGIE was quickly overwhelmed by a combinatorial explosion of inferences when it processed texts containing multiple sentences. A partial solution to this problem was implemented in SAM [Cullingford 1978], a program that used scripts to limit inferences to those relevant to the given text. Scripts represent stereotypical sequences of events. The restaurant script, for example, contains the knowledge that you sit down, order, eat, pay, and then leave.

The advantage of scripts is that they allow us to infer events in the script that have not been specified, such as knowing that someone must have eaten, even if it has not been explicitly stated. In addition, scripts helped disambiguate sentences and resolve pronoun referents. When we read "he tipped him five dollars," we know that "he" refers to the diner and "him" to the waiter. The major disadvantage of scripts is that they lack intentionality; although the actions and their order are specified, the reasons behind

them are not. In the restaurant script, for example, there is no information as to why someone pays for their food, only that they do.

1.5.2 Plan Based Understanding

PAM [Wilensky 1978] was a program designed to understand novel (not script-based) situations. To do so, PAM used general knowledge about the goals and plans of the characters in the stories it read to infer the causal connections between their actions. PAM knew what the plans were for achieving each goal, what goals particular states gave rise to, and what the inference rules were that aided in connecting events to plans. To understand a story PAM tried to explain each action in the story in terms of the plan it is a part of, each goal in terms of the plan it was a subgoal of, and each plan in terms of the goal it achieved.

As an illustration of PAM's processing, consider the following story, taken from [Wilensky 1983]:

Impress Date

> John wanted to impress Mary. He asked Fred if he could borrow the Mercedes for the evening.

Asking about the Mercedes is explained as part of a plan for getting possession of a vehicle, which is in turn explained as being instrumental (a subgoal) to taking Mary out in an expensive car, which is finally explained as the plan for the goal of impressing Mary that is explicitly mentioned in the text.

PAM could also understand and reason about more complicated stories involving goal conflicts and goal competitions. While PAM could understand the intentions of characters, it could not detect bad planning on their part or use knowledge about bad planning to understand a character's actions. In addition, PAM could not detect the use of a novel plan to achieve a goal and did not attempt to create new plans.

1.5.3 Memory Organization

These early story understanding programs did not maintain any memory of the stories they read. The IPP [Lebowitz 1980] and CYRUS [Kolodner 1984] programs were attempts to model the way people remember episodes and make generalizations from them. IPP read stories about terrorism, storing the important features of each story in an episodic memory, and then using this memory as a basis for making generalizations. After reading several stories about kidnapping in Italy, for example, IPP was able to conclude that the victims of kidnappers in Italy are usually businessmen. Interestingly, IPP could use these generalizations to assist in the understanding process. Using the generalization that terrorist attacks in Britain are normally done by the IRA, IPP would understand that "Irish Guerrillas" referred to members of the IRA.

A major limitation of IPP was that it simply filled in slots for the terrorist activities that it knew about, with no notion of causality. As a result, IPP often made generalizations that seem silly to people. In addition, because IPP worked by fulfilling prior expectations and ignoring information that did not conform to these expectations, unexpected information was often missed and not remembered.

CYRUS read stories about diplomatic events in the life of former Secretary of State Cyrus Vance, remembered these events, and then answered questions about Vance's activities. Like IPP, CYRUS stored these events in an episodic memory and made generalizations from them. However, CYRUS could also dynamically reorganize its memory when necessary. CYRUS used MOPs (Memory Organization Packets) [Schank 1982] to organize events around their differences, as illustrated by Figure 2, which shows the memory organization of several different diplomatic meeting events.

Each MOP consists of a content frame, which describes the normal features of the events it indexes, and indices to the events it organizes, which are specified to retrieve the event. For example, in Figure 2, the content frame of MOP1 describes the normal features of a diplomatic meeting, and the events organized by this MOP are indexed by their participants or their topic. Thus, to retrieve one of these events, the

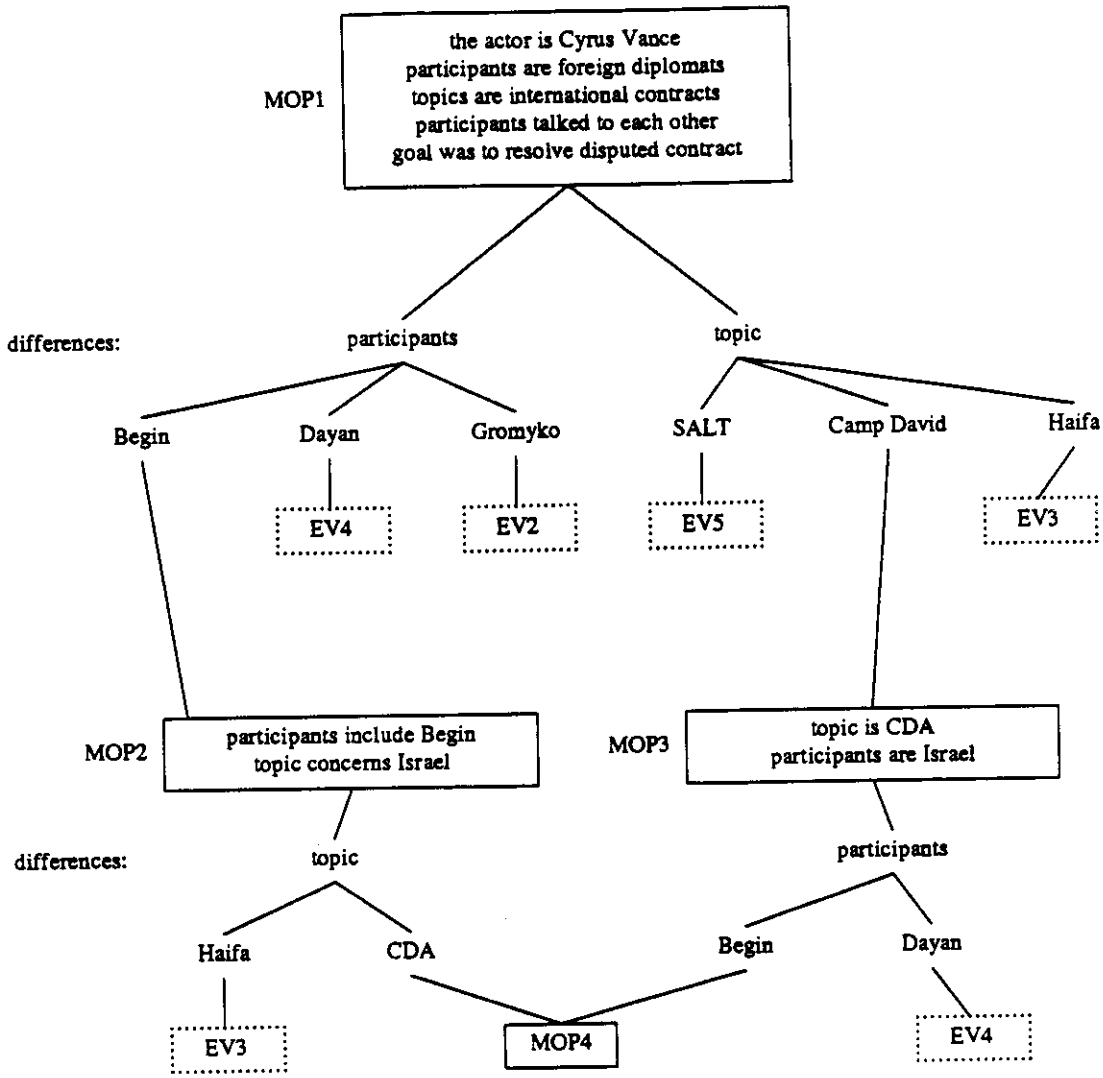

Figure 2: Memory Organization in CYRUS

meeting's participants or the meeting's topic must be specified — other features, such as the meeting's location, do not index these events. To retrieve an event when its indices were not specified explicitly, CYRUS would try to infer potential indices from the features it was given, using various retrieval heuristics, temporal knowledge, and reasoning about its own memory organization.

CYRUS showed how human episodic memory can be organized in an efficient manner by indexing events in terms of their differences. A limitation of CYRUS, as with IPP, was its lack of causal knowledge about its domain. In addition, unlike IPP, CYRUS did not use its generalizations to aid in parsing and disambiguating the questions it was posed. Importantly, neither system tried to organize planning knowledge or to use its memory events to plan for future events.

1.5.4 Integrated Understanding

Except for IPP, few of the story understanding programs integrated parsing and the process of memory access and search. In BORIS [Dyer 1983], rather than waiting for each input sentence to completely processed before integrating it into memory, memory accesses and searches take place as each input sentence is read. Thus, the episodic knowledge contained in the story BORIS was reading aided BORIS in parsing the story. For example, role information remembered from reading that George was a teacher allowed BORIS to understand that the phrase 'George examined Fred'' meant that George gave Fred a test. Alternatively, had BORIS read that George was a doctor, it would have understood the meaning of a physical examination.

BORIS was able to understand stories that involved novel planning failures, using TAUs (Thematic Abstraction Units) to represent abstract situations in which planning failures occurred. For example, TAU-TOO-COSTLY, which is characterized by the adage ''killing a fly with an elephant gun,'' captures the planning failure inherent in choosing the higher costing plan of otherwise equally efficacious plans. TAUs are important because they provide generalized planning advice across domains, rather than specific planning rules for each situation, and because they carry expectations about the affective reactions of narrative characters.

TAUs are also often the basis for cross-contextual remindings, since many stories seem to be indexed in memory under various TAUs. For example, even though there are no surface similarities, the story:

STUPID USER

I needed to get rid of one of my files, so I removed all of them with the command ''rm *''.

reminded one person of the story:

STUPID BOY

John had a problem with mice in his apartment. He sprayed his apartment with pesticide and ended up killing his cat.

Both of these stories are indexed under TAU-TOO-COSTLY because a high cost plan was used when a lower cost plan was available. In STUPID USER, **rm** could be used to remove a single file. In STUPID BOY, a mousetrap could have been used to kill the mice.

BORIS used intentional and causal links (i-links) to represent the motivations and intentions of characters. Dyer's i-links are shown in Figure 3. In this representation, a goal G intends a plan P, which is executed by an event E that achieves G. The i-links are grouped in pairs, so if if there is a link ''intends'' from a goal G to a plan P, there is also a corresponding link ''intended-by'' from P to G.

	G enables P	P enabled-by G
	G intends P	P intended-by G
E motivates G	G motivated-by E	
E thwarts G	G thwarted-by E	
E suspends G	G suspended-by E	
E achieves G	G achieved-by E	
E realized-by P		P realizes E
E blocks P		P blocked-by E
E forces E		

Figure 3: Dyer's intentional links

We illustrate this framework with the following story:

Poor Child

> John needed money. He decided to rob a bank. He changed his mind and asked his father for money. His father gave him twenty dollars.

John has a goal of possessing money, which *intends* a plan of stealing it from a financial institution, which is never *realized*. Instead, John *intends* another plan, asking someone for money, which is then *realized* with an event. This event *motivates* a goal of his father; namely, that John possess some money. This goal *intends* his father's plan to give John some money, which is *realized* by an event, which *achieves* John's goal.

I-links declaratively represent the relationship between the goals, plans, and actions of characters, and are used to guide search and retrieval during question answering. In contrast, PAM used procedural inference rules for the same purpose. While BORIS used I-links and TAUs in understanding stories about planning, it did not use them in creating new plans or generating advice.

1.5.5 Advice Giving

Despite its many contributions, BORIS did not have expert knowledge in any particular domain, and did not provide advice for the characters in the stories it read about. UC [Wilensky 1982, Wilensky 1984] is a program designed to give expert advice in using the UNIX system. UC accepts questions as input, parses them into a conceptual representation, and searches its memory for the answer associated with a given question. Figure 4 illustrates UC's capabilities, showing some questions processed by UC and the answers generated for them. These are taken from [Wilensky 1984].

```
User:   How do I delete a file?
UC:     Typing "rm filename" will remove the file with name filename
        from your current directory.

User:   A directory?
UC:     Typing "rm -r directory name" will recursively delete the
        directory with the name directoryname and all the files and
        directories contained in the directory.

User:   How can I find out how full the disk is?
UC:     Typing "df" will tell you how full the disk is.

User:   Do you know how to read a file?
UC:     Typing "pr filename" will print the file with name filename
        on your terminal.
```

Figure 4: Some examples of UC's question-answering behavior

When UC does not know the answer to a question, it uses the PANDORA program [Faletti 1982] to attempt to create specific "novel" plans by instantiating more general plans. For example, UC can answer a question like "How can I get more disk space?" by recognizing that the user has a goal of acquiring more of a resource, and then suggesting the appropriate plans for the goal: use less of the resource (remove unneeded files) or request more of the resource (ask the system administrator for more space).

Since UC has a question-answering orientation and is not concerned with understanding stories that involve planning failures, such as STUBBORN FILE, it does not try to build a model of the user or remember previous planning failures. Therefore its planner does not attempt to use pieces of existing plans or make use of known planning failures during the planning process.

1.6 Our Approach

How does AQUA build on and differ from these previous AI programs? AQUA, like BORIS, takes an integrated approach to parsing and understanding. AQUA also uses a modified set of i-links to represent the relationships between the user's goals, plans, and events. To aid in understanding user problem descriptions, AQUA has a CYRUS-like memory that organizes planning knowledge in the form of planning experiences and generalizations made from them. AQUA's memory is modified as user stories provide new information.

Like UC, AQUA is an advice giving program whose domain is UNIX consulting. Unlike UC, however, AQUA attempts to understand stories involving planning failures, and builds an explicit model of the user's problems and expectations. In addition, AQUA's planner tries to make use of previous planning failures when it encounters a new situation. Finally, AQUA can create new plans both by instantiating high-level plans, as does UC, but also by modifying existing plans to make them work better, repairing earlier failed plans and combining existing plans in novel ways.

2. The User Model

We must understand someone's problem before we can provide them with advice. AQUA understands a user's problem by parsing its English description into a conceptual representation that captures the user's goals, the plans used towards achieving those goals, the results of their execution, and importantly, the inferences that the user has made. That is, AQUA tries to understand not only the user's actions but also the user's current knowledge state. AQUA tries to infer, for example, what the user believes the enablement conditions are for any failed plans tried by the user. Once AQUA has built its user model, it generates advice by comparing the user's actions and expectations with its own, correcting user misconceptions and filling in gaps in the user's knowledge. In this section we show how AQUA represents user problems and discuss some of the rules needed to build this representation.

2.1 Representing User Problem Descriptions

Johnson and Solloway [Johnson 1984] have argued that intentional modeling is necessary for accurate problem diagnosis. AQUA uses intentional and causal links similar to Dyer's [Dyer 1983] i-links to represent the relationship between the various goals, plans, events, and states [Schank 1975, Schank 1977] mentioned in and inferred from the user's problem description. AQUA also uses i-links to represent hypotheses made by the user. We illustrate how AQUA uses i-links to represent user problem descriptions with the story STUBBORN FILE, repeated below for convenient reference.

STUBBORN FILE

USER: I tried to remove a file with the "rm" command. The file was not removed, and the error message was permission denied. I checked and I own the file. What's wrong?

AQUA: To remove a file, you need directory write permission. To remove a file, you do not need to own it.

AQUA's representation for STUBBORN FILE is shown in Figure 5. The user has a goal of removing a particular file, which *intends* the plan of using **rm**. When this plan is *realized* by an event, the event *results-in* an error message but does not *achieve* the user's goal. A state of insufficient permission is *required* for this error message to occur, and is the state that *disables* the user's plan. The user *refines* this state as the user not possessing the file, which *motivates* a user goal to verify the file's ownership, which is *achieved* by some unknown event.

Figure 5 illustrates how i-links capture the user's failed goal of removing a file, the user's failed plan of using **rm**, and the user's satisfied goal of checking the file's ownership. Importantly, they also capture the user's assumption that owning a file is a precondition to removing it. In the next section, we describe

in detail the i-links AQUA uses to represent user problems, and discuss how they differ from Dyer's.

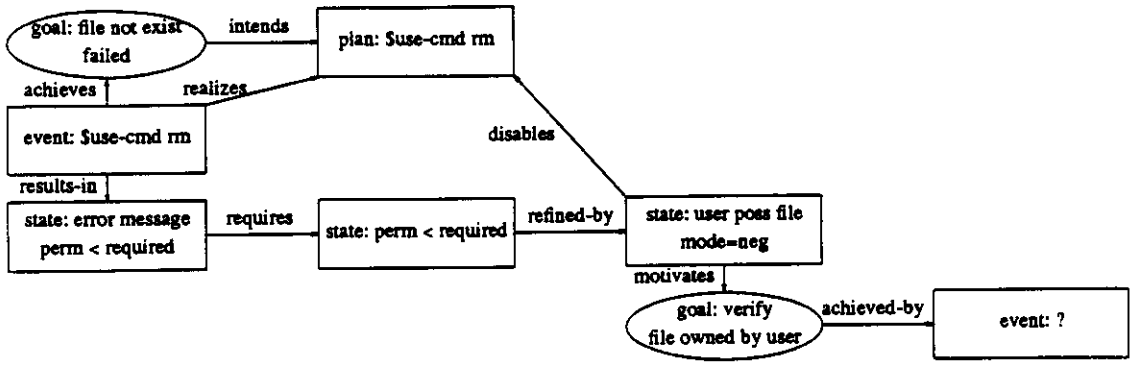

Figure 5: AQUA's representation of STUBBORN FILE

2.2 AQUA's I-links

The purpose of i-links is to capture the intentions and motivations of narrative characters. The i-links AQUA uses to represent user problem descriptions are listed in Figure 6.

	G achieved-by E	E achieves G	
	G motivated-by E	E motivates G	
S results-from E		E results-in S	
E realizes P	P realized-by E		
S enables P			P enabled-by S
S disables P			P disabled-by S
	G intends P		P intended-by G
S refines S			
S requires S			

Figure 6: AQUA's intentional and causal links

While AQUA's i-links are closely related to Dyer's, they differ in their treatment of states. In BORIS, states were indexed under the events they enable or disable and were not connected by intentional links to events, goals, or plans. In AQUA, however, there are intentional connections between states and other knowledge structures. We now describe the five differences between AQUA's i-links and BORIS's, and present the rationale behind our alterations.

(1) Goals are *thwarted* and plans are *disabled* or *enabled* by states, rather than events.

In BORIS, only events could thwart goal achievement or block plan execution. In STUBBORN FILE, however, the user's plan of using **rm** to remove his file is *disabled* by a state: insufficient write permission on the directory containing the file. This state also *thwarts* his goal of removing the file. The event, if any, that caused this state is unimportant and does not need to be represented.

As another example, this time from outside AQUA's domain, consider the following story:

ALL WET

> John was sitting on his front porch when it began raining. He tried to go back inside the house but found that the door was locked and he had forgotten his key. He got very wet.

John's plan of going inside the house is *disabled* because the door is locked and because he does not have a key, both states rather than events. His goal of staying dry is *thwarted* because it is raining and he cannot get inside. These stories point out the necessity for using intentional links to explicitly connect states with the goals they thwart and the plans they enable or disable.

(2) Events *result-in* states and states *result-from* events.

In AQUA, it is the states resulting from an event that thwart goals and disable or enable plans, not the event itself. Thus, when an event occurs, the states resulting from it are explicitly added to the representation using a *results-in/results-from* link. In STUBBORN FILE, for example, the error message *results-from* the user's use of rm. As another example, in ALL WET, the raining event *results-in* John's getting wet.

results-in/results-from is actually more of a causal link than an intentional link, but is included in the set of i-links because the states it links to events are linked to goals and plans with intentional links. The difference between *results-in* and Dyer's *forces* is that *results-in* connects an event to the states that result from it, while *forces* connects two events in which one event directly causes the other event to occur, such as dialing a phone number *forcing* that phone to ring.

(3) States can *require* or be *required-by* other states.

The *requires* link represents situations where one state exists because of another state. For example, in STUBBORN FILE, an error message results from the user's attempt to remove his file because the user does not have sufficient permission to do so. We can represent this situation using a combination of *requires* and *results-from* links. The existence of an error message *results-from* the use of rm, and *requires* a state of insufficient permission.

Notice that like *results-in*, *requires* differs from Dyer's *forces*. Although insufficient permission is a necessary condition for the error message, it is not sufficient — an event must occur to bring about error message's existence. In contrast, *forces* represents one event directly resulting in the occurrence of another event.

(4) States can *refine* and be *refined-by* other states; the *refines* link can also connect goals.

The *refines/refined-by* link represents the hypotheses and discoveries characters make about their current world state. In STUBBORN FILE, for example, the user hypothesizes that "insufficient permission to remove a file" means that "the file is not owned by the user." That is, the user specializes — or *refines* — the concept "insufficient permission" into the concept "file not owned by user."

Characters refine their goals, as well as refining their views about states, as illustrated by the following story:

TOUGH REMOVE

> I tried to remove my file "foo" with rm. I got an error message that said "foo" is a directory. So I tried to use "rmdir" to remove it. But I got an error message that said "foo" was not empty. So I used "rm" to remove the files it contained, and then used "rmdir" to remove it.

In TOUGH REMOVE the user first specializes his goal of removing a file into a goal of removing a directory, and then further refines this goal into a goal of removing a non-empty directory.

(5) The links *blocks*, *forces*, and *suspends* are not used in AQUA.

The *blocked/blocked-by* pair, which connected events with plans in BORIS, is no longer needed since only the states resulting from an event disable plans, not the events itself. Because *forces* and *suspends* are not required to represent any of the stories AQUA currently processes, they are not implemented. No theoretical claims are being made about this omission.

2.3 Building the User Model

We have shown how AQUA represents user problem descriptions. To build its representation for user problem descriptions, AQUA must perform three important tasks: recognizing user hypotheses, inferring user goals, and tracking the status of user goals. In this section we describe inferences AQUA must make to parse a user's English problem description.

2.3.1 Recognizing User Hypotheses

Users often try to figure out the causes of their planning failures. To do so, they hypothesize and test potential reasons for these failures. It is therefore important for AQUA to recognize and understand the assumptions and inferences that users make when their plans fail. In STUBBORN FILE the user makes the inference that owning a file is an enablement condition for its removal. AQUA can infer the user's hypotheses using the following rule:

```
IF   a plan P is disabled by a state S1 AND
     an attempt was made to verify a state S2 AND
     S2 can be a specialization of S1
THEN assume the user believes P is disabled by S2 AND
     assume the user hypothesizes S2 refines S1
```

That is, when a plan fails because of some generalized state (such as insufficient permission), one thing user's often do is hypothesize a specialization for the state (such as not owning the file) and then try to verify that it is indeed the problem.

Notice that the user had to determine that insufficient permission was the problem before he could specialize it into not owning the file. AQUA and the user make this initial hypothesis using the following rule:

```
IF   an error message occurs during plan execution
THEN assume that the error message describes the plan's disablement
```

In STUBBORN FILE, for example, the error message was "permission denied", so the inference can easily be made that the user does not possess the necessary access permissions and that this is causing the plan failure.

This rule is obviously domain dependent, since error messages are not always provided when plans fail. However, there are similar rules for determining why a plan has failed that are appropriate for other domains. For example, if a plan of driving somewhere fails because the car's engine stops running, it is a good idea to examine the dashboard indicators. In a sense they provide an error message, which points to the cause of the failure, indicating an empty gas tank, low oil pressure, an overheated motor, and so on. Further hypotheses must then be made to specialize the error into the failure's cause so it can be fixed.

2.3.2 Inferring User Goals

Users rarely state their goals explicitly, so they must be inferred. For example, in STUBBORN FILE the user starts describing his problem with the sentence "I tried to remove a file with rm." AQUA must infer that the user's goal is to have the file cease to exist, and does so by applying the following rule:

```
IF   an actor makes an attempt to perform an action X
THEN infer that the actor's goal is to achieve the
     state(s) that normally result from performing X
```

Here, because the user states that he is trying to remove a file and the normal result of removing an object is for the object to cease to exist, AQUA infers the user's goal.

Users often simply mention an unusual or undesirable state, expecting the advice giver to infer that their goal is to get out of their particular situation. As an example, consider the following story:

CONFUSED TERMINAL

USER: My terminal is in a strange mode and typing "stty" did not fix it. I checked and my terminal type is set correctly.

AQUA: Use the command "stty nohang", and then turn your terminal off and then on again to return your terminal to its normal mode.

Here, AQUA should infer that the user's goal is for the terminal to be returned to its normal mode, which can be done using the following rule:

```
IF   an object X is an unusual state S, or the user
     mentions that S is undesirable
THEN infer that the user's goal is for X to
     be returned to its normal state.
```

2.3.3 Tracking the Status of User Goals

Just as the user's goals may have to be inferred, the status of a goal may also have to be inferred. In STUBBORN FILE, for example, when the user says that "the file is still there," AQUA must infer that the user has made a failed attempt to achieve his goal. This is accomplished by applying the following rule:

```
IF   a goal is to transform an object X, and it can be
     inferred that no transformation has been achieved
THEN the goal has not been achieved
```

In this case, because the user's goal is to change an object's location and the user explicitly states that the object's location hasn't changed, AQUA can infer the user's goal failure. We can see that whenever a state is mentioned, AQUA must determine whether it gives rise to a goal or if it indicates the success or failure of a goal.

3. Providing Advice

Once we understand what someone's problem is, we can try to provide advice. While they describe their situation to us, we are often reminded of similar experiences. To give advice, we compare their situation to the experiences we have been reminded of, explain any differences, and provide the solutions we discovered and used in our earlier experiences.

AQUA models this behavior. After AQUA has built a user model from the user's problem description and has understood what the user's problem is, it tries to provide the user with advice. To do so, AQUA compares the user's experience with any experiences it was reminded of during the understanding process, using the differences to classify the user's problem. AQUA then uses heuristics associated with each problem class to determine what to generate as advice. In this section we discuss the different classes of user problems we have identified and show how advice heuristics associated with each of these classes can be used to provide advice.

3.1 User Problem Classes

We examined approximately 50 user problem descriptions and their associated expert advice.[4] We have recognized 9 classes of user planning problems: (1) Unknown plan for goal, (2) Wrong plan for goal, (3) Incorrect goal refinement, (4) Unknown disablement, (5) Wrong disablement, (6) Unknown enablement, (7) Incorrect enablement, (8) Better plan exists, and (9) Plan has bad side effects. AQUA has processed at least one story from each of these classes.[5] In this section we describe each class, the rules for recognizing it, the advice heuristics associated with it, and sample user problems belonging to it.

3.1.1 UNKNOWN PLAN FOR GOAL

User's often have no idea how to accomplish their goals. Consider the following story:

OOPS

USER: I accidentally typed "rm *" and removed all my files. What do I do now?
AQUA: Your files can be recovered from tape by sending mail to "request".

Here the user's problem is that he has a goal of accessing the removed files, but has no plan for achieving this goal. AQUA's advice provides a plan (having the files reloaded from tape) that achieves this goal. Figure 7 shows how we can abstractly characterize this problem class using i-links and shows the appropriate advice heuristic for this class.

Questions beginning with "How can I do *X*?" or "Is there any way to do *X*?" usually indicate a user problem that falls into this class. The following story is an example:

WHERE IS MY FRIEND?

USER: Is there any way to list the users who are currently logged on?
AQUA: Use the "who" command to list the users who are currently logged on.

---

[4] At UCLA users can electronically mail problem descriptions to "help". "help" is several UNIX experts who solve the user's problem and save copies of both the user problem description and the solution they provided. Accurate behavior samples are therefore easy to obtain.

[5] The stories shown as examples are not necessarily the same stories processed by AQUA. However, we have represented all of these stories using the i-links described in section 2, and we are currently implementing the lexical knowledge, inference rules, and planning experiences necessary to process them.

USER's experience:	G1 intends ?, G1 unachieved
AQUA's knowledge:	G1 intends P1, P1 realized-by E1, G1 succeeds
Advice heuristic:	Tell the user the correct plan for his goal (G1 intends P1)

Figure 7: UNKNOWN PLAN FOR GOAL

Occasionally users try the right plan for their goal, the plan fails, and examining the plan's results causes them to infer that they have a different goal. The following story is an example:

COMPLICATED REMOVE

USER: I want to remove one of my directories. When I tried "rmdir", the error message was "directory not empty". What should I do?

AQUA: Use the "rm" command to remove the files in the directory. Then use the "rmdir" command to remove the empty directory.

Here, the error message the user gets when he tries to remove the directory allows him to refine his goal to removing a nonempty directory. However, he does not have a plan for this refined goal, so AQUA's advice provides this plan.

3.1.2 WRONG PLAN FOR GOAL

Users often try to achieve a goal with an inappropriate plan. This mistake usually occurs because they have achieved similar goals with the plan they tried, but the plan has some flaw that prevents it from satisfying their current goal. This class of problems is exemplified by following story:

WRONG REMOVE

USER: I tried to remove a directory with "rm" but the directory was not removed. What should I do?

AQUA: Use the "rmdir" command to remove a directory. The "rm" command can only be used to remove files.

In WRONG REMOVE, the user's problem is that her plan of using **rm** has failed to achieve her goal of removing a directory. AQUA's advice is to provide a plan (using **rmdir**) that achieves the user's goal, and to specify that the user's plan is used to achieve a different goal (removing a file). Figure 8 shows the characterization of this problem class and its advice heuristic.

USER's experience:	G1 intends P1, P1 realized by E1, G1 not achieved by E1
AQUA's knowledge:	G1 intends P2, P2 realized by E2, G1 achieved by E2, P1 intended-by G2
Advice heuristic:	Tell the user the correct plan for his goal (G1 intends P2), Tell the user the normal use of the plan he tried (P1 intended-by G2)

Figure 8: WRONG PLAN FOR GOAL

A user may have a plan that achieves their goal that is different than AQUA's, so to recognize problems in this class AQUA must do more than simply detect that the user's plan for a particular goal differs from its own. Consider the following story:

DISASTER STRIKES

USER: I accidentally used the command "rm -r" to remove a directory and the files it contained. How can I recover them?

AQUA: Your files can be recovered from tape by sending mail to "request".

The user does not know how to recover the removed files, so DISASTER STRIKES belongs in the problem class UNKNOWN PLAN FOR GOAL, not WRONG PLAN FOR GOAL.

As a side effect of processing DISASTER STRIKES, AQUA should now know that the goal of removing a directory and the file it contains can be achieved using the plan **rm -r**, even if it did not have this planning knowledge previously. Therefore, when a user presents a plan that AQUA is not familiar with, AQUA must infer that plan's results to decide whether the user's plan succeeds or fails. If the user's plan succeeds, AQUA must compare it to AQUA's own plan, and if the user's plan appears to be better, AQUA should use the user's plan in the future. In this way learning new plans can take place as a side effect of advice giving.

3.1.3 INCORRECT GOAL REFINEMENT

Not only do users choose the wrong plans for their goals, but they often do not even know precisely what it is that they are trying to accomplish. That is, users often have a different goal that they think they do. Here is an example:

STUBBORN FOO

USER: I tried to remove the file "foo" with the "rm" command, and the error message was "file is a directory." Help me.

AQUA: "foo" is a directory, not a file. To remove a directory, you need to use the "rmdir" command.

The user's stated goal is the file **foo**'s removal. The user's plan, **rm**, fails to achieve this goal, and has an outcome that contradicts a feature in the user's goal; namely, that **foo** is a directory and not a file. Thus, AQUA infers that the user's goal is really to remove a directory, and AQUA's advice points this mistake out and gives the correct plan for removing a directory. The characterization and advice heuristic for this class is shown in Figure 9.

USER's description:	G1 intends P1, E1 realizes P1, G1 failed, P1 results-in S1, S1 contradicts a feature of G1
AQUA's knowledge:	G2 is G1 without contradiction specified in S1, G2 intends P2, E2 realizes P2, G2 succeeds
Advice heuristic:	Tell the user the feature specified by S1 that contradicts a feature of G1 Tell the user the plan for G2 (G2 intends P2)

Figure 9: INCORRECT GOAL REFINEMENT

AQUA must rely on information from the plan's outcome or from the user's description of the world state to determine the user's real goal. Notice that in STUBBORN FOO, if the user does not mention the error message, AQUA cannot determine what the user's problem is.

3.1.4 UNKNOWN DISABLEMENT

Clever users often try the correct plan for their goal, but find themselves unpleasantly surprised when the plan does not work and they have no idea why. The following story is an example:

THESIS WONT PRINT

USER: I need to print my thesis but every time I try to print it, I get the error message "cannot access device /dev/lpr." What is going on here?

AQUA: You are not able to print your thesis because the printer is turned off.

THESIS WONT PRINT is similar to STUBBORN FILE, except that the user does not hypothesize a reason for the plan failure. This problem class is shown in Figure 10.

USER's description:	G1 intends P1, P1 realized-by E1, E1 does not achieve G1, ? disables P1
AQUA's knowledge:	G1 intends P1, P1 realized-by E1, E1 does not achieve G1 S1 disables P1, S1 undone, P1 realized-by E2, G1 achieved
Advice heuristic:	Tell user G1 cannot be achieved because S1 needs to be fulfilled

Figure 10: UNKNOWN DISABLEMENT

Many problems that fall into this class occur because of unfulfilled preconditions, usually preconditions that the user is unaware of. Turning on the printer and making sure it has enough paper are preconditions to printing anything, but users are rarely responsible for fulfilling them in most computing environments.

Plans usually have more than one condition that can disable them. When such plans fail, users often hypothesize and fix several disablement conditions, but will not know about or forget an important one. Here is an example:

MISSING PROGRAM

USER: I am trying to create my own command directory. I placed a new command in ".bin", but I get "command not found" when I try to execute the program. The program is executable. I also made ".bin"

AQUA: The "csh" does not know about your new program. Use the command "rehash" to inform the "csh" about your new command.

In this example, the user has ensured that two potential disablement conditions do not exist, but has failed to correct the third. AQUA mentions the third disablement condition and a potential correction for it.

AQUA should suggest as potential disablement conditions any states likely to disable a plan that the user has not mentioned in the problem description, as exemplified by the following version of MISSING PROGRAM:

MISSING PROGRAM (AGAIN)

USER: I am trying to create my own command directory. I placed a new command in ".bin", but I get "command not found" when I try to execute the program.

AQUA: The command might not be executable. Use "chmod" to make it executable. ".bin" might not appear in your path. Use "set" to modify your path. The "csh" might not know about your new program. Use the command "rehash" to inform the "csh" about your new command.

Notice that since AQUA cannot determine which of the potential disablements is the true cause of the

failure, its advice points out th

3.1.5 INCORRECT DISAB

When a plan fails, user
(which we are by now famil
owning a file is preventing h
write into the directory cont
terizes this situation.

USER's description:	G1 int
AQUA's knowledge:	G1 int S2 dis
Advice heuristic:	Tell t

In STUBBORN FIL
rm. However, they dif
STUBBORN FILE wo
AQUA must be able to
Advice regarding
If the state is impossi
the following story:

USER: I am
const

AQUA: The
kno
VA

The situatio
tional constraint
possible to undo
al rule:

AQUA realize
er) is disable
ing troff to p

[6]The troff
an Apollo

3.1.6 UNKNOWN ENABLEMENT

In addition to asking why plans fail, users ask what they have to do to get the plan to succeed. That is, they explicitly ask what the preconditions of plans are, as in the following story:[7]

DISK USAGE PROBLEM

USER: Is there anything special I have to do to get "df" to compute the disk usage of my home file system?

AQUA: You must execute "df" on the site of the file system where you need the disk usage computed.

Here, the user has the correct plan for her goal, and is simply requesting information about the enablement conditions for the plan. AQUA describes the enablement condition to her. The abstract characterization for this problem class is shown in Figure 12.

USER's description:	G1 intends P1, ? enables P1
AQUA's knowledge:	G1 intends P1, S1 enables P1
Advice heuristic:	Tell user that S1 enables P1

Figure 12: UNKNOWN ENABLEMENT

The difference between the UNKNOWN DISABLEMENT and UNKNOWN ENABLEMENT is that in the former the user mentions an event that failed, but in the latter the user is simply requesting the plan's preconditions. As with UNKNOWN DISABLEMENT, all of the plan's enablement conditions should be specified in AQUA's advice. The following story provides an example:

NO MORE OPEN MODE

USER: When I start "vi", it begins in "open mode". What do I have to do to get it to start up in "visual mode"?

AQUA: You must be on a terminal with an addressable cursor, and you must set your terminal type before starting "vi".

3.1.7 INCORRECT ENABLEMENT

Users often manage to get the preconditions for plans wrong, and then ask why the plan did not work. The following story is an example:

READ MAIL

USER: What do I need to do to be able to use "rdm" to read my mail on "diana"? Trying "on diana rdm" did not work.

AQUA: You do not need to run "rdm" on "diana". You must make sure "diana" is up.

Here the user believes that running **rdm** on a particular machine enables using **rdm** to read his mail. AQUA knows that the enablement is simply having that machine be up and that there is no need to run **rdm** on that machine. The advice provides the actual enablement condition and corrects the mistaken enablement.

[7]Because the UCLA version of UNIX is distributed, some of these user problem descriptions are unique to UCLA users and will be unfamiliar to the reader with previous UNIX experience.

USER's description:	G1 intends P1, S1 enables P1
AQUA's knowledge:	G1 intends P1, S2 enables P1, S1 does not enable P1
Advice heuristic:	Tell user that S1 does not enable P1, Tell user that S2 enables P1

Figure 13: INCORRECT ENABLEMENT

3.1.8 BETTER PLANS EXIST

Users frequently use a plan that achieves their goal but has some flaw (inefficiency, costly, and so on) that makes it less than optimal. For example:

SLOW REPLACEMENT

USER: I just edited a file, changing each beer to BEER, one by one. Is there an easier way?

AQUA: Use "1,$s/old-text/new-text/" to replace all instances of a pattern throughout a file. It is faster.

Here, AQUA's plan for doing global text substitution is a single command, instead of the sequence of commands the user is currently using. AQUA provides its plan as advice and provides the reason why it is better.

USER's knowledge:	G1 intends P1, P1 realized-by E1, G1 achieved
AQUA's experience:	G1 intends P1, P1 realized-by E1, G1 achieved, G1 intends P2, P2 realized-by E2, G1 achieved, P2 better than P1 because of R1
Advice heuristic:	Tell the user that G1 intends P2, Tell the user that P2 is R1

Figure 14: BETTER PLANS EXIST

There are many ways for one plan to be better than another. AQUA uses the *planning metrics* (suggested in [Dyer 1983]) to compare plans, and has rules that aid in computing their relative values. For example, the rule used to determine the better of the two plans in SLOW REPLACEMENT is:

```
IF   plan P1 requires a repeated action AND
     another plan P2 has another action done once
THEN P2 is a lower cost plan than P1
```

Users do not always explicitly ask if there is a better plan, as does the user in SLOW REPLACEMENT, as illustrated by the following variation of SLOW REPLACEMENT:

UNDO REPLACEMENT

USER: I just edited a file, changing each alex to ALEX, one by one. How can I undo these changes?

AQUA: Use "u" to undo your last change. Use "1,$s/old-text/new-text/" to replace all instances of a pattern throughout a file. It is faster.

Thus, if AQUA's plan compares favorably with the user's, it should be suggested as an alternative, even if the user did not explicitly request a better plan. AQUA performs this comparison whenever the user's plan for a goal differs from its own.

3.1.9 PLAN HAS BAD SIDE-EFFECTS

Users often select a plan to achieve their goal that violates other goals. This occurs either because users do not realize or forget the effects of executing that plan. Here is an example.

LOST FILE

USER: I removed my file from the line printer queue using "rm". How can I recover it?

AQUA: Your file can be recovered from tape by sending mail to "request". Use "iprm" to remove files from the line printer queue. "iprm" does not remove the file.

Here the user's goal is to prevent the file from printing. Using rm achieves this goal, but has the undesirable side-effect of removing the file, which presumably violates the users goal of accessing its contents. We can represent this problem class as shown in Figure 15.

USER's description:	G1 intends P1, P1 realized-by E1, E1 results-in S1, G1 achieved, G2 failed
AQUA's knowledge:	G1 intends P2, P2 realized-by E2, G1 achieved, G2 not failed
Advice heuristic:	Tell the user that G1 intends P2, Tell the user why P2 instead of P1

Figure 15: PLAN HAS BAD SIDE EFFECTS

3.2 Multiple Problems Within a Single Description

LOST FILE actually contains two user problems: PLAN HAS BAD SIDE EFFECTS and UNKNOWN PLAN. Not only is there a better plan than the user's for preventing a file from printing, but the user is not aware of any plan for achieving his goal of recovering the accidentally removed file. In fact, many user problems have sub-problems in several different classes. The following story is another example.

CREATE FILE

USER: I tried to create an empty file named "database" by copying another empty file named "empty" that I had created with the editor. The copy failed with an error message about permission denied for "empty". What do I do?

AQUA: You need to be able to read a file to copy it. Using "touch" is an easier way to create an empty file.

AQUA must notice that this one problem contains both UNKNOWN DISABLEMENT and BETTER PLANS EXIST in it and provide appropriate advice for each type of problem. Here, AQUA realizes this because it is reminded of two planning experiences; the first involves creating a file (AQUA used touch) and the second involves copying files and getting a permission denied message. In comparing the file creation episode with the user's, AQUA can infer BETTER PLANS EXIST. In comparing the failed file copying episode with the user's, AQUA can infer UNKNOWN DISABLEMENT.

Since there can be more than one user goal in a single problem description and one potential planning failure for each attempt at achieving a goal, AQUA must be prepared to recognize more than one user planning problem in a user's problem description. To do so, AQUA examines each planning attempt for each goal to determine the type of planning error made, and provides appropriate advice.

3.3 Preventive Advice

So far, we have seen two general classes of advice: achievement and recovery. Achievement advice specifies a plan for achieving a goal, and is exemplified by the stories THESIS PRINTING PROBLEM and WRONG REMOVE. Recovery advice explains why a goal has failed and provides a plan for recovering from the failure. The stories STUBBORN FILE and LOST FILE are examples of this type of advice. There is one more important class of advice: preventive. Preventive advice specifies how to prevent future goal failures.

The following story, a new version of our earlier story OOPS, is an example of preventive advice:

OOPS II

USER: I accidentally removed all of my files with "rm *". Is there any way I can recover them?

AQUA: Send mail to "request" asking that your files be recovered from tape. Use the "rm -i" command to be asked before the file is removed.

Here, the preventive advice is a new plan for removing files that makes it less likely that the user will accidentally remove files.

When should preventive advice be given? Since the point of preventive advice is to protect against future high cost recoveries from goal failures, AQUA considers giving preventive advice whenever it detects a situation where there was a goal failure with a high recovery cost. Whenever a recovery plan is suggested, AQUA examines the cost of the recovery plan to decide whether or not preventive advice should be given. In OOPS II, the goal failure is the user's desire to access the removed files. Because it can take several days to get the files back, there is a high cost recovery plan, so AQUA provides an alternative plan. Preventive advice is especially important when there is no recovery plan, as in the following story.

NOT SAVED

USER: I was editing a file when the system crashed. How do I get it back?

AQUA: You cannot recover your file. You need to save your file often during editing.

Importantly, a potential goal failure with a high cost recovery plan should trigger preventive advice. Consider the following story:

DEADLY REMOVE

USER: Is there an easy way to remove all of the files in a directory?

AQUA: Use the "rm *" command to remove all of the files in a directory. Use the "pwd" command to verify that you are in the correct directory before using "rm *".

People often use "rm *" in the wrong directory, resulting in the accidental removal of a multitude of files. As we have pointed out earlier, file recovery is a high cost plan. Thus, because the plan used is a low cost, high risk plan that often results in goal failures, the advice specifies a method to lessen the likelihood of a goal failure.

4. Planning

People can understand and provide advice for novel situations. To do so, we need the ability to create plans for goals that we have not had before. We create plans by making use of our previous planning experiences, combining, modifying, and correcting existing plans. AQUA must have the same capability, since we do not want to limit AQUA's advice giving to situations where it knows the solution in advance.

AQUA's planner models the planning process that we have observed people follow in our informal protocols. The planner is called by the advice-giving component when it finds that there is no stored plan for the user's goal, and uses several situation-based, high-level planning strategies to create new plans from existing ones. Once a new plan has been formulated, it is indexed under the user's goal and no new planning takes place the next time an identical situation is encountered. In this section we discuss some planning heuristics people apper to use to create novel plans.

4.1 A Planning Example

How are new plans created? We illustrate the process people go through with the following story, presented earlier as an example of INSUFFICIENT GOAL REFINEMENT.

COMPLICATED REMOVE

USER: I want to remove one of my directories. When I tried "rmdir", the error message was "directory not empty". What should I do?

AQUA: Use the "rm" command to remove the files in the directory. Then use the "rmdir" command to remove the empty directory.

Here, the disablement condition refines the user's goal from removing a directory to removing a nonempty directory. Thus, AQUA's advice should simply be a plan for this more specialized goal. AQUA's representation for the situation is shown in Figure 16.

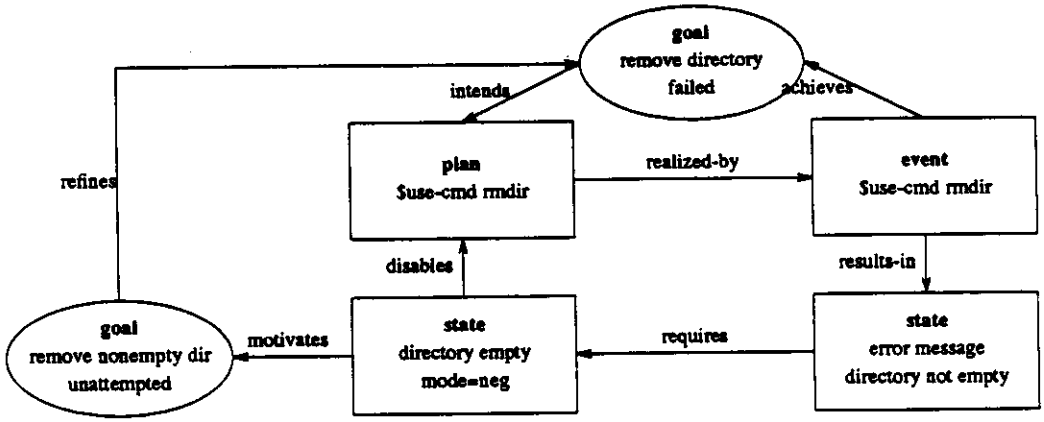

Figure 16: Representation of COMPLICATED REMOVE

If AQUA already has a plan for the goal of removing a nonempty directory, this example is not particularly interesting. However, suppose AQUA knows only that **rm** is used to remove a file, that **rmdir** is used to remove a directory, and that, in general, something can be made empty by figuring out its contents

and then removing them.[8] AQUA should be able to put this information together to create a plan for removing nonempty directories.

How is a plan that will remove nonempty directories formulated from this knowledge? Here is a protocol taken from a novice UNIX user given this problem.

Protocol-1

> Well, let's see. The problem is that **rmdir** won't remove a directory that isn't empty. But I can still use **rmdir** if I make the directory empty. How can I make it empty? Well, the directory contains files, and I have a way of getting rid of files. So I can use **rm** to remove all of the files in the directory. Then I can use **rmdir** to remove the directory.

We can see that the user first realizes that his problem is that the plan used (**rmdir**) won't work if the directory being removed isn't empty. Since he knows that it will work if the directory is empty, he figures out that if he can make the directory empty, he can use **rmdir** to delete it. To empty the directory, the user realizes he has to remove any files it contains, which he knows he can do with **rm**. A diagram of the user's planning process is shown in Figure 17.

4.2 Planning Strategies

In COMPLICATED REMOVE, the user creates a new plan both by combining specific existing plans (**rm** and **rmdir**) and instantiating a general plan (for emptying an object) in a novel way. In this section we describe the planning strategies that are guiding the user's planning process. A planning strategy is a heuristic guideline for the planner that consists of a situation and a suggestion for how to plan in that situation.

4.2.1 UNDO-DISABLEMENT

UNDO-DISABLEMENT is a planning strategy that provides a method for handling planning failures with a known disablement condition, and is shown in Figure 18. The planning situation is that a plan P1 was executed in a failed attempt to achieve a goal G1. P1's execution fails because G1 has some feature X that another goal G2, which P1 successfully achieves, does not have. The suggestion is to get rid of the disabling feature and then use the old plan.

Situations in which this strategy is applicable occur frequently. Here is another example:

CAN'T MAKE DIRECTORY

USER: I want to create a directory called "foo" but when I type "mkdir foo" I get the error that "foo" already exists.

AQUA: "foo" is a file, not a directory. The "mkdir" command can not create a directory when a file exists with the directory's name. Use the "rm" command to remove "foo". Then use the "mkdir" command to create the directory.

In this case, the goal is to create a directory named **foo**. However, the normal plan of using **mkdir** fails, and the disabling feature is that **foo** already exists as a file. UNDO-DISABLEMENT suggests that undoing the disabling condition (**foo**'s existence) will solve the user's planning problem. Undoing this condition involves getting rid of **foo**.

---

[8] Of course, AQUA may know other things at this point, but we care only about the knowledge that will be relevant to the planning process.

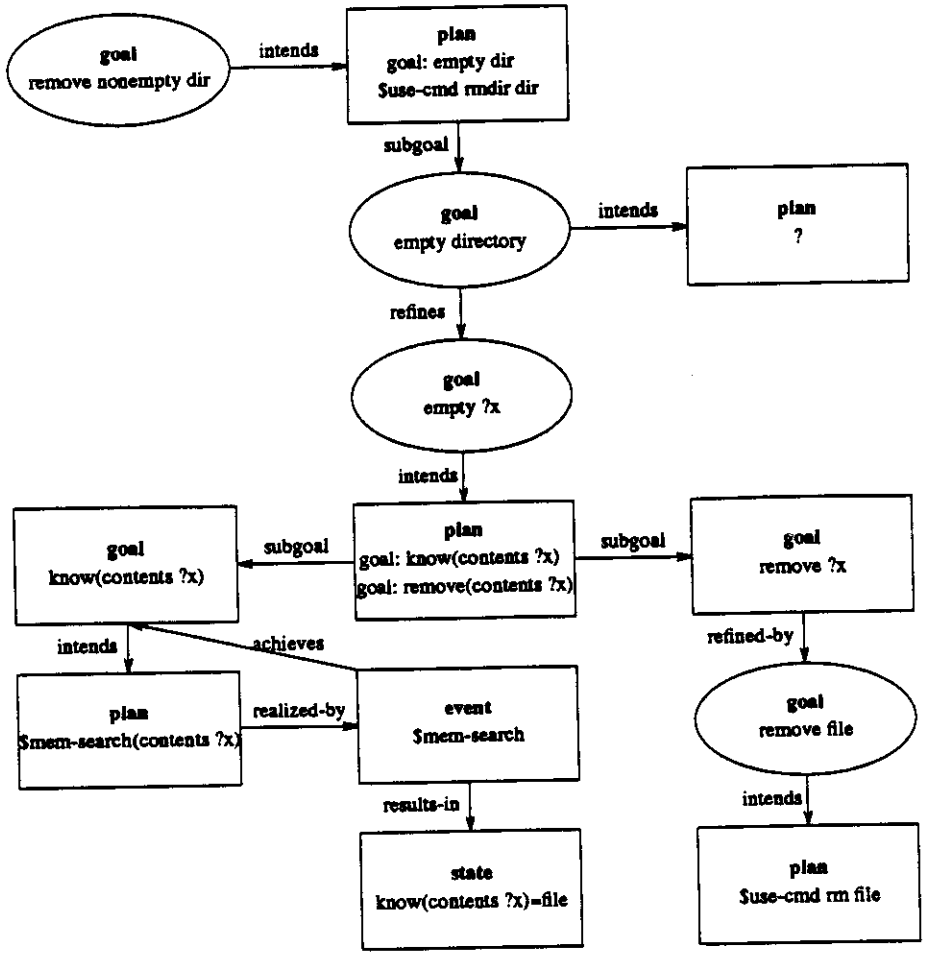

Figure 17: Planning process for COMPLICATED REMOVE

Situation:	G1 intends P1, P1 realized-by E1, E1 disabled-by S1 (G1 failed) G2 intends P1 (G2 achieved), S1 is (G1 has feature X), G2 is (G1 without feature X)
Suggestion:	Create a new plan with steps: (1) UNDO feature x (2) Use plan P1

Figure 18: UNDO-DISABLEMENT

4.2.2 GENERALIZE GOAL FEATURE

The second strategy used in COMPLICATED REMOVE is GENERALIZE-GOAL-FEATURE, shown in Figure 19, which provides advice on searching memory for a reasonable plan. This strategy says if there is no plan for the goal we are trying to achieve, try generalizing features of the goal to determine whether a useful generalized plan exists. In COMPLICATED REMOVE, the user does not know how to empty a directory, but does have a general plan for emptying an object: find out the object's contents and then use the appropriate plan for removing them. The user finds an appropriate plan by generalizing his

Situation:	G1 does not intend a plan
Suggestion:	Select a feature of G1 to generalize Generalize the selected feature of G1 Search memory for a plan for the generalized goal Instantiate that plan with the feature of the G1

Figure 19: GENERALIZE-GOAL-FEATURE

goal to "emptying an object," searching for a plan for the generalized goal, and then instantiating the plan it finds with an object that is a directory.

How can AQUA intelligently select the goal feature to generalize? AQUA could randomly select features to generalize, in the hope that eventually it would come up with a goal it has a plan for. However, when there are many goal features, where only a few are relevant, this is not an acceptable approach. Therefore, instead of blind generalizations, AQUA uses generalization strategies indexed under GENERALIZE-GOAL-FEATURE that select the feature to generalize based on the current goal situation. In COMPLICATED REMOVE the particular generalization strategy used is GENERALIZE-GOAL-OBJECT, shown in Figure 20.

Situation:	G1 involves making a change to a specific type of physical object
Suggestion:	Generalize the object G1 to physical object

Figure 20: GENERALIZE-GOAL-OBJECT

The user, after unsuccessfully searching for a plan to empty a directory, applies GENERALIZE-GOAL-OBJECT and tries searching for a plan to empty a physical object. This search is successful. Notice that there are other goal features that could have led to a less useful generalization. For example, the user could have generalized a goal to making some change to a directory. However, this doesn't lead to any useful plans.

4.3 A More Complex Planning Example

We will use the following story and several protocols of users providing advice to further illustrate the use of UNDO-DISABLEMENT and GENERALIZE-GOAL-FEATURE and to introduce several other strategies.

WANT DIRECTORY NAMES

USER: How can I print my directory names?

AQUA: Use the command "ls -l | grep ^d" to list your directory names.

Before we examine the process by which this plan of AQUA's was formulated, we will briefly explain how it works. ls is used to list the names of the files in a directory. ls -l provides additional information such as the each file's type (directory, file), size, and so on. Output lines beginning with a d indicate a directory. grep is a program that finds input lines that match a pattern; in this case the pattern is ^d, which specifies lines beginning with the letter d. The | makes ls's output grep's input. Thus, this command runs ls and uses grep to select the directory names from its output.

4.3.1 UNDO-EXTRA-EFFECT

Here is a protocol of a user who never had this goal before, although he did know **ls** and its options and **grep** and its patterns.

Protocol-2

> **ls** prints the names of all files and all directories. But we only want directory names, so we have to filter out file names. If I can write a pattern that indicates a directory, we can use **grep**. So, how can we mark directory names? **ls -l** prints a **d** at the beginning of a line for directories and a **-** for ordinary lines. Can I write a **grep** pattern to do that? Let's see. ^ is the beginning of a line, so I just have to follow it with a **d**.

The user starts with the plan of using **ls**. Unfortunately, **ls** prints both directory names and file names. Since there is an extra, undesirable effect of the plan, UNDO-EXTRA-EFFECT is a useful strategy. UNDO-EXTRA-EFFECT is shown in Figure 21.

Situation:	G1 intends P1, P1 realized by E1, E1 results-in S1,S2, S1 achieves G1, S2 is undesirable
Suggestion:	Make the plan P1 plus an extra step that undoes S2

Figure 21: UNDO-EXTRA-EFFECT

The strategy simply states that if a plan is producing an undesirable side effect, add a step to the plan that undoes the undesirable effect. Here it suggests a goal of getting rid of the file names that are output by **ls**.

Unfortunately, the user has no existing plan for achieving this goal, so GENERALIZE-GOAL-FEATURE is applied, resulting in the goal of removing undesirable command output. The user has a general plan for this goal: modify the command to generate an indication for the desirable output, write a pattern for grep that can recognize the indication, and put these two commands together for the final, workable plan. At this point, no new strategies are needed, since the user has known plans for the subgoals. The user's complete planning process is shown in Figure 22.

4.3.2 PREVENT-EXTRA-EFFECT

Consider the following protocol taken from a different user for the same problem.

Protocol-3

> Is there a command that lists directory names? **ls** lists all names. Is there an option for directory names? Not that I remember. Is there an option that does anything special with directory names? Oh, **ls -F** puts a / at the end of each directory name, so I can filter them out with **grep**. But there are multiple names per line and some might not be directories. Hmmm. How can I get one per line? **ls -1** does that. So **ls -1F** will have one directory name per line ending in a /. That's easy to recognize with **grep**, it's just **/$**. **ls -1F | grep /$** should work.

In this example, the user starts off with a different strategy, attempting to modify **ls** to print only directory names. This strategy, PREVENT-EXTRA-EFFECT, is shown in Figure 23. PREVENT-EXTRA-EFFECT is useful in the same situations as UNDO-EXTRA-EFFECT; however, it suggests making changes to the plan instead of trying to undo the undesirable effects. Here, the user doesn't think up any workable modifications to the plan and PREVENT-EXTRA-EFFECT is abandoned.

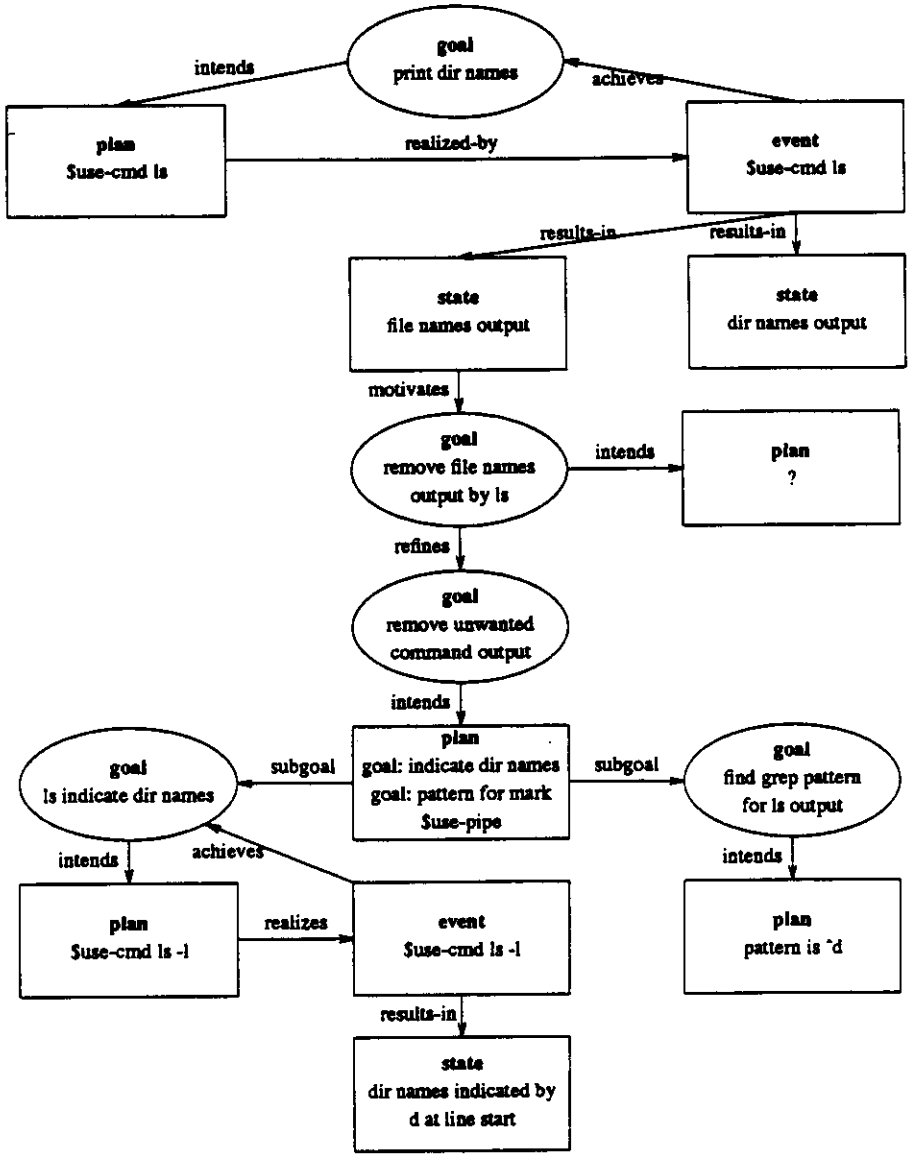

Figure 22: Planning Using UNDO-EXTRA-EFFECT

Situation:	G1 intends P1, P1 realized by E1, E1 results-in S1,S2, S1 achieves G1, S2 is undesirable
Suggestion:	Modify P1 to not produce the undesirable state

Figure 23: PREVENT-EXTRA-EFFECT

The user's planning then proceeds with the same strategies used by the previous user. However, this user chooses a different plan, ls -F, which indicates a directory by appending a / to its name. Once this plan is selected, the user tries to write a grep pattern to recognize directory names. At this point the user realizes that no pattern will work because there is more than one name placed on each line. PREVENT-EXTRA-EFFECT is selected as a planning strategy, generating a goal of modifying ls -F to print one name per line. The user knows that the -1 option will do this. A diagram of this user's planning process is shown in Figure 24.

A third user planned for this problem in a similar way, but when she discovered that ls -F wrote more than one name on a line, she used UNDO-EXTRA-EFFECT instead of PREVENT-EXTRA-EFFECT. Her plan was to use another command to transform ls -F's output into one name per line input for grep. Thus, different high-level strategies can result in different low-level plan steps.

4.4 AQUA's Planner

AQUA's planner is called from the advice-giving component when there is no previously successful plan indexed by the user's current situation. As with most planners, its input includes the goals it is planning for and a description of the current world state. In addition, the advice-giving component provides the planner with a conceptual representation of the user's attempts toward achieving these goals and their results. The planner's output is the sequence of actions the user must perform to achieve his goal.

The planner's control structure is straightforward and summarized in Figure 25. First, features of the situation it is given are used as indices to plans that are likely to be applicable in that situation.[9] When no workable plan is found the planner is forced to create a new plan. To do so, abstract characteristics of the planner's current situation are used to index the planning strategies discussed in the previous section. The strategy indexed is then applied to come up with a potential plan. The planner can then plan for the subgoals of this plan. After each plan is found, the planner adds the expected results of the plans execution to its representation of the situation. The situation is then examined and if problems are detected with the plan, alternate strategies are attempted.

4.5 Planning – Related Work

AQUA's planning strategies are similar to Wilensky's meta-plans [Wilensky 1983, Faletti 1982] and Dyer's TAUs [Dyer 1983] providing advice and suggesting ways to select, use, and create appropriate plans. Meta-plans are plans for achieving goals of the planning process (meta-goals) and are designed to handle goal interactions. For example, REPLAN and CHANGE-CIRCUMSTANCE are meta-plans that achieve the meta-goal RESOLVE-GOAL-CONFLICT. REPLAN tells the planner to find a plan that does not lead to the goal, either by using a plan that specifically resolves the conflict (USE-NORMAL-PLAN) or by finding another plan that does not have the action that leads to the conflict (TRY-ALTERNATIVE-PLAN). CHANGE-CIRCUMSTANCE tells the planner to remove the state that is leading to the goal conflict. For example, if there is a conflict because of a lack of a resource, it is a good idea to obtain more of the resource.

AQUA's planning strategies differ from Wilensky's meta-plans in several ways. They provide more specific planning suggestions than Wilensky's meta-plans. The strategy PREVENT-EXTRA-EFFECT, for example, is more detailed than the meta-plans TRY-ALTERNATE-PLAN or CHANGE-CIRCUMSTANCE, as it suggests a particular method of recovery for a specific type of plan failure: modifying the failed plan so that it does not produce the undesirable side effect. Unlike meta-plans, AQUA's planning strategies suggest ways to create new plans from existing ones and to repair failed plans. UNDO-DISABLEMENT is another instance of REPLAN that suggests a method of combining two existing plans to form a novel plan, after one of the existing plans has failed.

---

[9]Indexing and the organization of planning experiences is discussed in the next section.

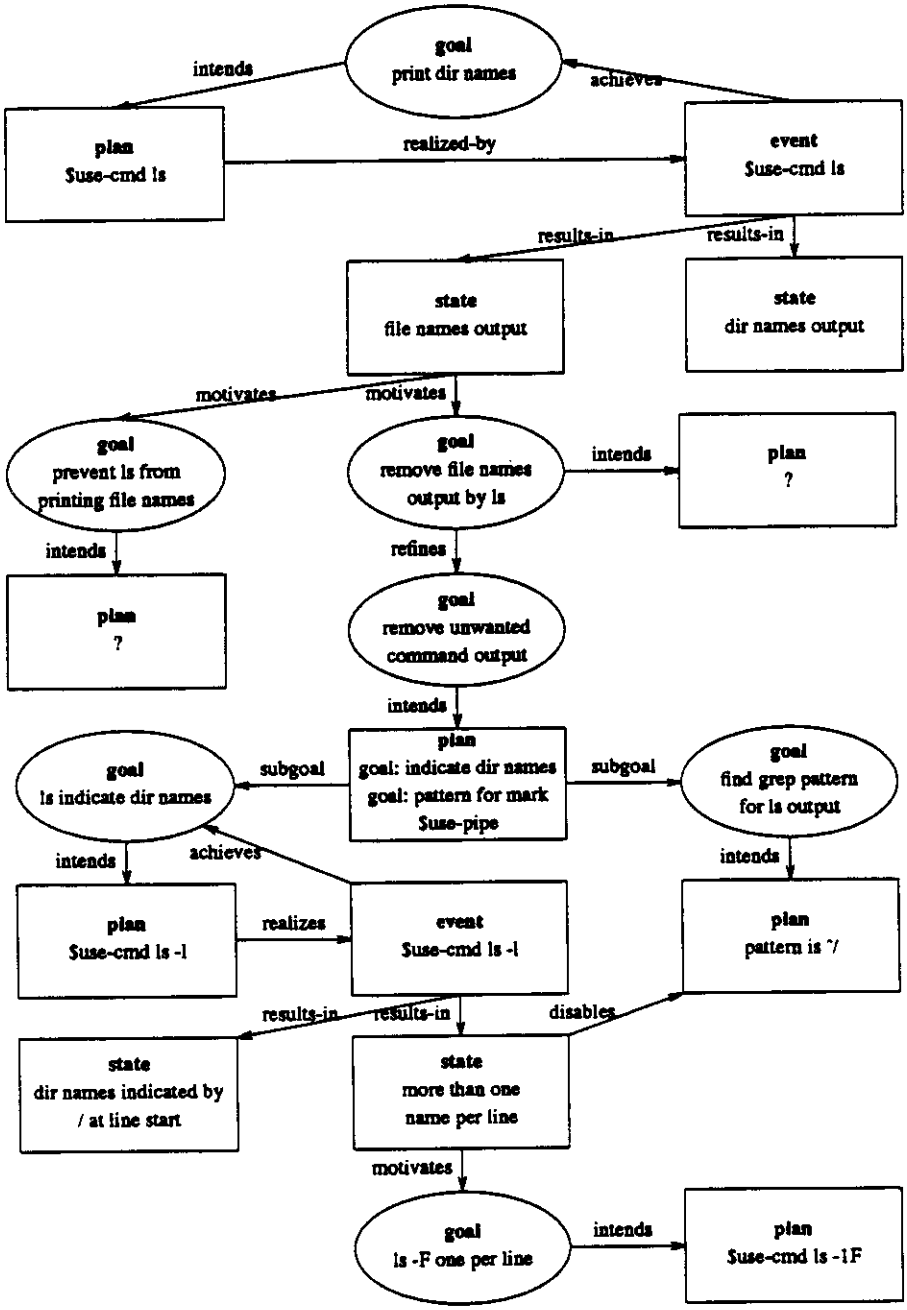

Figure 24: Planning Using PREVENT-EXTRA-EFFECT

(1) Find a candidate plan indexed by the current planning situation.
(2) If no plan is found, find a planning strategy appropriate for the current situation and use it to compute a candidate plan.
(3) Recursively plan for any subgoals in the candidate plan.
(4) Compute the plan's results to see if it works. If it fails, go to step 2, trying a different strategy.

Figure 25: The planner's control structure

Dyer's TAUs provide useful advice to a planner when it is in a situation where more than one plan is available. For example, TAU-TOO-COSTLY ("Don't kill a fly with an elephant gun") tells the planner to choose the lower cost of two equally efficacious plans. However, they do not suggest ways characters can create new plans or provide novel ways to recover from plan failures, the purpose of AQUAs planning strategies.

WOK [Hammond 1983] is also similar in spirit to AQUA, using planning strategies to guide the plan creation process. However, WOK's planning strategies are specific to goal interactions in its recipe-creating domain, rather than being general aids to the planner. One strategy used by WOK, for example, suggests that if the current situation is such that the taste of one ingredient dominates the taste of another, that object should be replaced with another, different taste. It appears as though AQUA's planning strategies serve as higher level suggestions than WOK's, and are therefore more general.

PLEXUS [Alterman 1985] is a planner than can refit existing plans to novel situations, using the background knowledge associated with the existing plan to select the steps to modify and the types of modifications to make. PLEXUS plans by trying to apply each step of the old plan, and if that fails, abstracting in a planning hierarchy associated with that step. Once an abstracted step works, it is specialized to fit the current situation, and the planner proceeds to the next step. In a sense, PLEXUS implements the GENERALIZE-GOAL-FEATURE planning strategy. Unlike AQUA, however, PLEXUS does not have any other strategies for reusing plans, and does not attempt to make use of information from planning failures to modify existing plans.

5. Memory Organization

Experiential (or episodic) memory plays an important role in advice giving, both in problem understanding and problem solving. When we are presented with a planning problem, remindings of similar planning experiences can provide potential solutions for it. During planning, we frequently create new plans by combining and modifying plans we have used previously. Because experts possess large numbers of planning experiences, it is important that planning knowledge be organized and retrieved in a reasonable manner. In this section we discuss the organization of planning experiences in episodic memory.

5.1 Planning Experiences

We remember our planning experiences: the goal we wanted to achieve, the plan we tried to use to achieve it, the results of using the plan, the plan's success or failure, and the causes we discovered for any failure. We can remember, for example, that **rm** is used to remove a file, that we get an error message when we try to use it to remove a directory, that trying to use it to remove a file named * can cause disastrous results, and that getting the error message "permission denied" means that we could not write into the directory containing the file. We also remember information we have generalized from these planning experiences. For instance, after using **rm** to try to remove mail messages and directories and having it fail, we are likely to generalize that **rm** is useful only for removing files.

We are reminded of these planning experiences when a similar experience occurs, and we use this information to predict the plan's results, as well as to explain any failure. For example, if we happen to get the error message "permission denied" when we use **rm**, and it has failed that way before, we will be

reminded of the previous failed experience. We can then use this information to guide us in recovering from the failure; here, changing the directory's write permission before we try to remove the file with rm.

5.2 Representing a Planning Experience

How do we represent a planning experience? Planning experiences are represented in memory using the same representation that we used for user problem statements. Thus, a planning experience contains information about a particular use of a plan, including the goal it was used for, its results, the events instantiating the plan's steps, and any disabling conditions. For example, consider the following planning experience:

BUSY FILE

> I wanted to remove one of my files, so I used **rm**. But it didn't work, telling me the file was busy. I assumed that someone was reading or writing the file. But I asked around and found out that I couldn't remove the file because the file was a running program.

AQUA's representation in memory for this experience is shown in Figure 26.

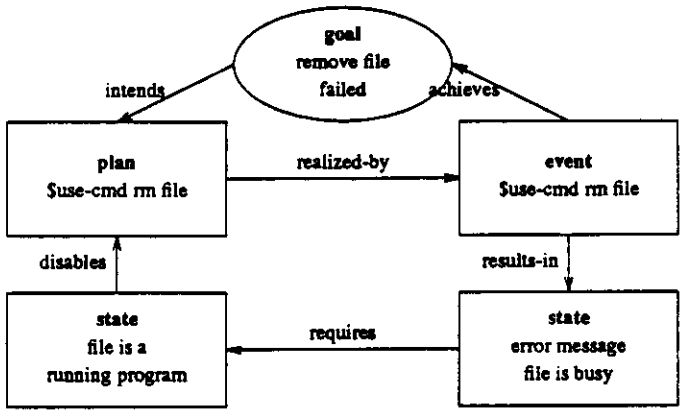

Figure 26: Memory Representation of BUSY FILE

AQUA remembers the results to the experience, as well as the cause of any failures. Thus, if the same planning failure occurs later, AQUA will be reminded of the earlier failure and know its cause, allowing corrective action to be taken.

5.3 Organization of Planning Experiences

Our organization for planning experiences is similar to that used by Kolodner to index events in CYRUS [Kolodner 1984]. However, AQUA's MOPs are generalized planning experiences, with a content frame consisting of the goal the plan is being used for and the results of the plan's use. The events are the individual planning experiences, and are indexed by differences from the normal or ideal experience with the plan. The indices are goal features, plan outcomes, and features of the world state when the plan was invoked.

As an example, Figure 27 shows a piece of AQUA's memory organization, organizing different experiences with **rm**. AQUA's MOPs are called *pMOPs*, since they organize planning experiences. The pMOP for the plan "$USE-CMD **rm**" describes the normal use of that plan: the plan is used to try and remove a file, the plan is executed by typing **rm**, and the plan results in a file that no longer exists. The

events indexed under this pMOP are specific experiences with using this plan. For example, one experience indexed under this pMOP is rm-exp.1, an experience where rm was used to try and remove a file, and the result was a permission denied error message. Other experiences indexed under this pMOP include trying to use rm to remove a directory and trying to remove a running program.

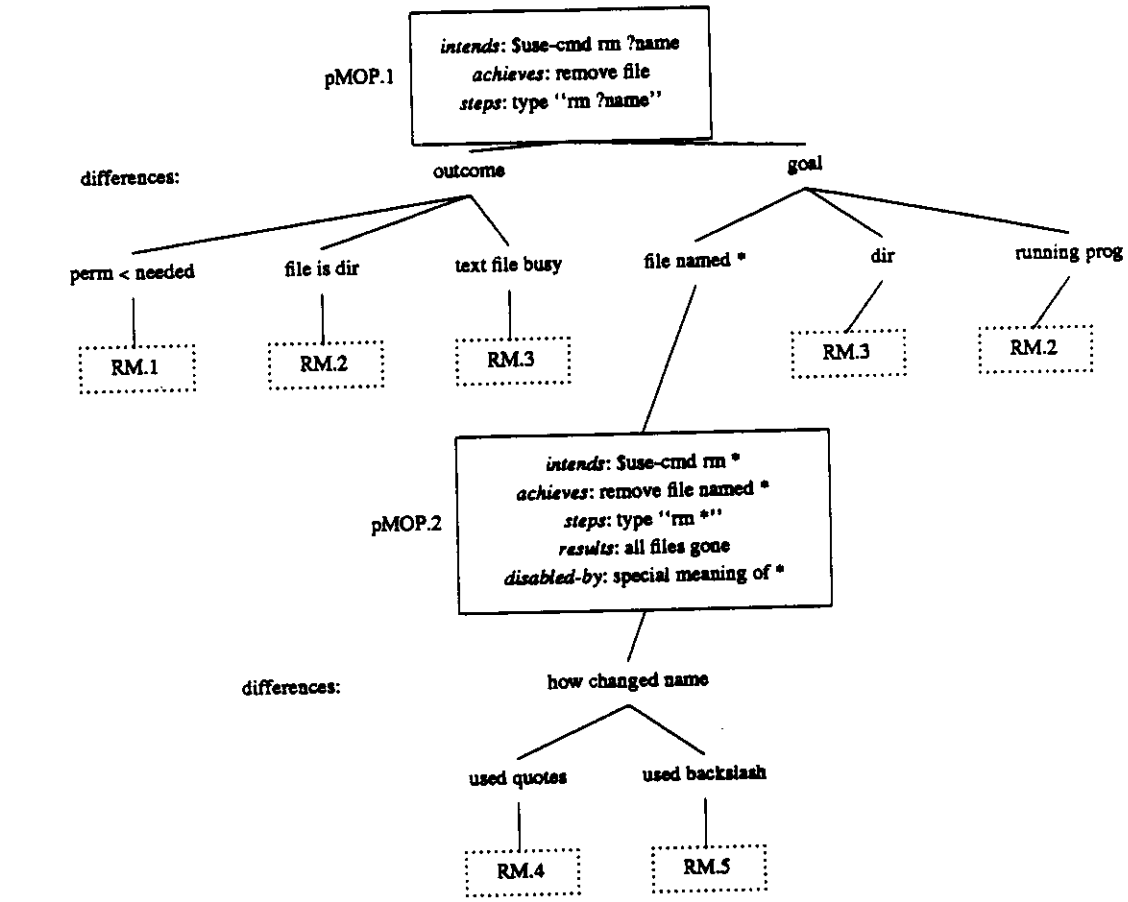

Figure 27: Organization of rm planning experiences

5.3.1 The Indices to Planning Experiences

Given a planning experience, AQUA uses features of it as indices to try and recall a similar planning experience. The indices to a planning experience are features of the plan's use (the goal it was trying to achieve) and features of the plan's outcome (the results of the plan's use). For example, when AQUA is presented with an experience of rm being used to try and remove a directory, it will use this unusual use as an index and retrieve its own similar experience (rm-exp.2). This similar experience will allow it to predict that the plan will fail. Similarly, an experience of rm producing a permission denied message will remind AQUA of rm-exp.1, providing the knowledge that directory write permission is required to remove the file.

Experts cannot easily list all of the ways a plan can fail. Very often they fail to remember more than one or two common failures. However, when they are given a particular description of a failure, if they have had a similar experience they can quickly determine its cause. The above organization accounts for this behavior. With this organization, given some information about a specific plan use, AQUA can be reminded of its earlier experiences and use this information to predict its outcome and to infer the cause of its failure.

5.4 Goal Organization

An expert in any domain will have had hundreds of different goals, each with various plans and planning experiences associated with it. Because plan retrieval should not slow down as the expert learns more plans, we need an efficient method for indexing these plans. To facilitate efficient retrieval, AQUA organizes the goals it remembers in a specialization hierarchy. Each goal has the pMOP describing its normal plan attached to it.

A portion of the hierarchy dealing with removal goals is shown in Figure 28. Given a goal, we find the plan appropriate for it by using features of the goal to index through the hierarchy. Thus, if we have a goal to remove a nonempty directory, to find the appropriate pMOP, we first use the feature that the object of the remove is a directory and then use the specialization that its contents are nonempty.

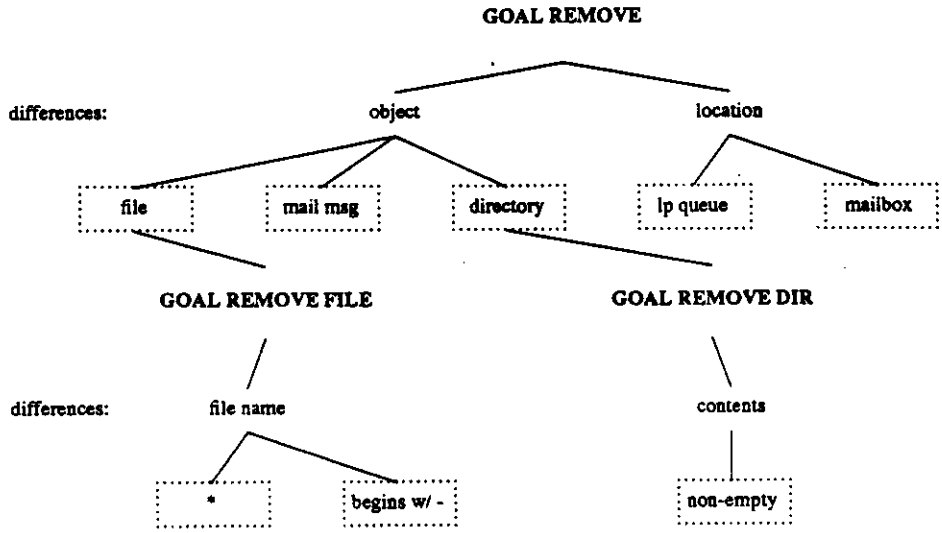

Figure 28: A portion of removal GOAL hierarchy

Notice that the hierarchy will be traversed as far as there is available information. If it is not known that a directory is empty, the pMOP found will be the one for removing a directory, and not the more specialized pMOP for removing a nonempty directory. Similarly, if there is no entry in the hierarchy for a particular feature, that feature will be ignored. This model accounts for the common mistake people make of using rm to remove a file named *. Until they observe disastrous results they do not realize certain file names are important and require the use of an alternate plan.

5.5 Indexing New Plans

When a new plan is formed for a new goal, it should be remembered so that no future planning is necessary for subsequent occurrences of the goal. In COMPLICATED REMOVE, for example, AQUA creates new plans that empty a directory and removing a nonempty directory. The next time either of these goals occur, there should be no need for planning.

AQUA indexes new plans under the goals they were created for by creating a pMOP describing the plan and then linking it to the goal. The hard part of indexing new plans is deciding where the goal they are linked to belongs in the goal hierarchy. Currently, to index a new goal, AQUA traverses the hierarchy until it has gone down as far as it can with the information it has. The new goal is then indexed by the features of it that differ from the goal it is indexed under in the hierarchy. This indexing scheme corresponds with the model of increasing expertise proposed in [Kay 1985]. Figure 29 shows AQUA's goal hierarchy before processing COMPLICATED REMOVE; Figure 28 shows it after the story has been indexed.

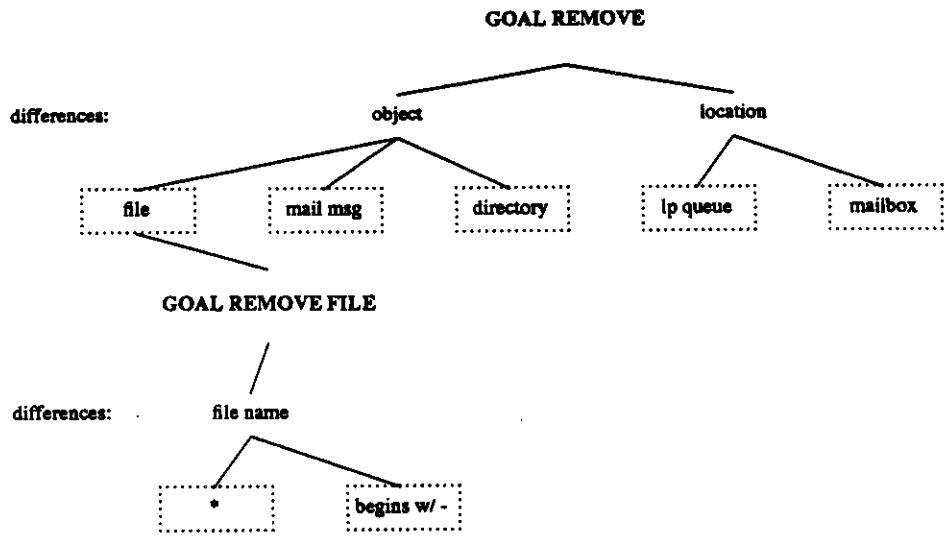

Figure 29: Goal Hierarchy before COMPLICATED REMOVE

6. Implementation Details and Trace

We now present a more detailed overview of AQUA, discussing the processes of parsing problem descriptions and generating English advice — issues we ignored earlier. We also provide a complete example of AQUA understanding a user problem description and providing advice, as well as a trace of the planner creating a novel solution for a planning problem.

6.1 Implementation Overview

AQUA is implemented in T [Rees 1984], a lexically scoped dialect of LISP, and runs on an Apollo workstation. AQUA uses features of RHAPSODY [Quilici 1985], a graphical AI environment, implemented in T, that provides tools for representing concepts and displaying them graphically, performing pattern matching, defining demons, and creating and using discrimination nets [Charniak 1980]. The current version of AQUA has understood and provided advice for several different stories. Each story takes between 2 and 5 minutes to process.

6.1.1 The Parser

When AQUA is presented with a user problem description, its first task is to build a conceptual representation of the user's problem. AQUA's parser is a demon-based parser similar to McDypar [Dyer 1983]. Demons are a form of delayed procedure, consisting of a test and an action. Whenever the test condition is true, the demon is said to "fire" and its action is executed. A demon can "spawn" other demons, as well as decide when to kill itself. These demons perform tasks such as word disambiguation, memory search, concept explanation and applying inference rules.

Each word in AQUA's lexicon has a concept associated with it — possibly several, if the word is ambiguous — along with demons that determine the correct meaning from context and determine how this word's concept fits in with other concepts in the current story. We illustrate demon-based parsing in more detail later in the section with a trace of AQUA's problem understanding ability.

6.1.2 The Generator

After AQUA has classified the user problem, it builds a conceptual representation of its advice. This representation is turned into English by RHAP [Reeves 1985], a recursive decent generator. Here is an example of a concept and RHAP's output:

```
Concept:    (AP &AP.2
               GOAL &REMOVE-FILE-GOAL
               PLAN &USE-CMD-RM
               LINK INTENDS)

Output:     Use the "rm" command to remove a file.
```

Here is a simplified explanation of RHAP's generation process. Each concept class has an associated template that provides information on how to generate the English that describes the concept. For the above concept, the template is `Use <PLAN> to <GOAL>`. The concept this template represents specifies that the goal of removing a file *intends* the plan of using **rm**. The template tells RHAP to output the word "Use", generate the concept's PLAN slot, output the word "to", and finally generate the concept's GOAL slot. Similar templates are associated with the concepts found in the goal and plan slots of the above concept.

6.2 Trace of STUBBORN FILE

Here is an annotated trace of AQUA understanding and giving advice for the story STUBBORN FILE. As a complete, unannotated trace of AQUA's processing is over 30 pages long, we have been forced to edit the verbatim trace to keep this paper to a reasonable length. We have made the following changes:

1. Although the trace starts out in great detail, showing each demon that is spawned, fired, and killed, toward the end we simply show the highlights, letting the reader fill in the missing information.

2. Every time a demon is spawned, it prints a message that describes the task it is to perform. We only show this message for the first instance of each demon class. We have also eliminated the extra information some demons output about their parameters and internal state.

3. Finally, we have made some formatting changes to increase readability. For example, we have changed the indentation and eliminated extra white space.

6.2.1 The First Sentence

```
>  (story 'STUBBORN-FILE)

PROCESSING SENTENCE: I TRIED TO REMOVE A FILE WITH RM *PERIOD*

I ==> Adding to *wm*: #(WMN.1)
    Created concept:
        (HUMAN &HUMAN.2)

    Spawning demon: (MERGE-THING.1 #(WMN.1) #(Procedure 246))
        Search for a matching concept
            (same class, no contradictory slots)
        and merge the concepts, replacing OUR-NODE's concept
    Executing -act: (MERGE-THING.1)
```

Lexical Entries – As AQUA reads each word or phrase it looks up its meaning in the lexicon. If a word is unambiguous, its concept is attached to the word's working memory node and any demons associated with this concept are spawned.

```
TRIED ==> Adding to *wm*: #(WMN.2)
    Spawning demon: (DISAMBIGUATE-TRIED.1 #(WMN.2))
        Examine the concept following TRIED.  If it's an action,
        create an EVENT whose goal is that action's normal result.
        If it's an object, create an EVENT instantiating the action
        that's the object's normal use.  Ignore the legal meaning.
```

Bottom-up Disambiguation – There are several possible meanings for "tried", illustrated by the following sentences:

(1) I tried to remove a file with rm.
(2) I tried Michelob.
(3) I was tried for stealing computer time.

In (1) "tried" means an event occurred toward achieving a goal. In (2) it means that an object was used for its normal function. For BEER, this action is an INGEST. In (3) "tried" indicates a legal action.

AQUA must infer which of these meanings of "tried" is meant. AQUA does so by spawning a demon that examines the concept that follows "tried". If it is an ACTION, then meaning (1) is selected. If it is an OBJECT, meaning (2) is selected.

```
TO REMOVE ==> Adding to *wm*: #(WMN.3)
    Created concept:
        (ACTION &ACTION.2
            TYPE    REMOVE
            ACTOR   ?*
            OBJECT  ?OBJ
            OUTCOME &STATE.21)
```

```
Spawning demon: (EXPECT.1 #(WMN.3) #(ACTION.2) (OBJECT) PHYS-OBJ AFT)
     IF    a CONCEPT with one of the given CLASSES is found when
           searching in the given direction
     THEN  the CONCEPT is bound to the given GAP
Spawning demon: (EXPECT.2 #(WMN.3) #(ACTION.2) (ACTOR) HUMAN BEF)
Executing +act: (EXPECT.2)
     Slot "(ACTOR)" in #(ACTION.2) <-- #(HUMAN.2)
```

Slot Filling – A concept such as REMOVE has various slots in it – the actor of the REMOVE, the object that was REMOVED, and the object's location – along with an expectation for the class of the concept that fills each slot. For example, we expect the actor of a REMOVE action to be a HUMAN. The EXPECT demon fills a slot by searching through working memory for a concept with the expected class. Word order restrictions are taken care of by the direction of the search. For REMOVE, the actor is expected *before* the concept and the object is expected *after* the concept. EXPECT demons run at a higher priority that other demons, since we want slots to be filled before memory search takes place.

```
Executing +act: (DISAMBIGUATE-TRIED.1 #(WMN.2))
Created concept:
     (EVENT &EVENT.13
          ACTOR    ?ACTOR
          ACHIEVES ?GOAL
          REALIZES ?PLAN)

     Inferring user's goal is #(GOAL.15), the result of #(ACTION.2)
     Slot "(ACHIEVES)" in #(EVENT.13) <-- #(GOAL.15)
Spawning demon: (INFER-INTENDS-LINK.1 #(WMN.2))
     IF   filled both the REALIZES and ACHIEVES slots of an event
     THEN infer an INTENDS link between the GOAL the EVENT achieves
          and the PLAN it realizes
Spawning demon: (EXPECT.3 #(WMN.2) #(EVENT.13) (REALIZES) (PLAN) AFT)
Spawning demon: (EXPECT.4 #(WMN.2) #(EVENT.13) (ACTOR) (HUMAN) BEF)
Spawning demon: (NOTE-ITEM.1 #(WMN.2) EVENT)
     Record that a GOAL, PLAN, EVENT, or STATE was instantiated.
     Place it on the to be explained list.
Executing +act: (EXPECT.4)
     Slot "(ACTOR)" in #(EVENT.13) <-- #(HUMAN.2)
Executing +act: (NOTE-ITEM.1)
Adding EVENT #(EVENT.13) to explain list
Spawning demon: (NOTE-ITEM.2 #(WMN.2) GOAL #(Procedure 251))
Executing +act: (NOTE-ITEM.2)
Adding GOAL #(GOAL.15) to explain list
```

Inferring user goals and intentions – Once AQUA processes the word "remove", it has disambiguated "tried" as an event achieving a goal. The normal result of the attempted action is assumed to be the user's goal. AQUA must also infer that the plan *realized-by* this event is *intended-by* the goal *achieved-by* this event. This task is done by the demon INFER-INTENDS-LINK, which waits until the goal and plan slots of the event have been filled in, and then makes this inference.

Other inferences are made at the end of each sentence, when AQUA verifies that it has explained each goal, plan, event, or state mentioned in the problem description. A concept is explained if it connected to the rest of the representation by appropriate i-links. The explanation process is examined in more detail shortly.

```
A ==> Adding to *wm*: #(WMN.4)
     Spawning demon: (IGNOR.1 #(WMN.4))
          Mark OUR-NODE as processed.
     Executing +act: (IGNOR.1)
```

```
FILE ==> Adding to *wm*: #(WMN.5)
    Created concept:
        (PHYS-OBJ &PHYS-OBJ.3
            TYPE FILE)

    Spawning demon: (MERGE-THING.2 #(WMN.5) #(Procedure 253))
    Executing +act: (EXPECT.1)
        Slot "(OBJECT)" in #(ACTION.2) <-- #(PHYS-OBJ.3)
    Executing -act: (MERGE-THING.2)
```

Concept Reference – Whenever a physical object is mentioned, AQUA searches its representation of the user's problem to see if a similar object already exists. If one does and has no contradictory features, AQUA assumes the newly mentioned object refers to the existing object and the features of the two are merged. Here, however, no file has been mentioned, so AQUA creates a new instance. Since memory search takes place for each mentioned object, articles are ignored except when they occur in phrases.

```
WITH RM ==> Adding to *wm*: #(WMN.7)
    Created concept:
        (PLAN &PLAN.10
            TYPE  USE
            NAME  RM
            STEPS &SCRIPT.11)

    Spawning demon: (EXPECT.5 #(WMN.7) #(PLAN.10) . . . PHYS-OBJ BEF)
    Spawning demon: (NOTE-ITEM.3 #(WMN.7) PLAN)
    Executing +act: (EXPECT.5)
        Slot "(STEPS OBJECT)" in #(PLAN.10) <-- #(PHYS-OBJ.3)
    Executing +act: (EXPECT.3)
        Slot "(REALIZES)" in #(EVENT.13) <-- #(PLAN.10)
    Executing +act: (NOTE-ITEM.3)
        Adding PLAN #(PLAN.10) to explain list
    Executing +act: (INFER-INTENDS-LINK.1)
        Inferring that user believes:
            GOAL #(GOAL.15) intends PLAN #(PLAN.10)

*PERIOD* ==> Adding to *wm*: #(WMN.8)
    Spawning demon: (EXPLAIN-CONCEPTS.1 #(WMN.8))
        Spawn a demon for each concept on the explain list onto an
        explain agenda.  Then run the demons on that agenda.
    Spawning demon: (IGNOR.3 #(WMN.8))
    Executing +act: (IGNOR.3)
    Executing +act: (EXPLAIN-CONCEPTS.1)
    Spawning demon: (EXPLAIN-PLAN.1 #(PLAN.10) #(WMN.7))
        Apply rules to see if plan can be explained
    Spawning demon: (EXPLAIN-GOAL.1 #(GOAL.15) #(WMN.2))
        Apply rules to see if goal can be explained
    Spawning demon: (EXPLAIN-EVENT.1 #(EVENT.13) #(WMN.2))
        Apply rules to see if event can be explained
    Executing +act: (EXPLAIN-EVENT.1)
        Event #(EVENT.13) explained: realizes plan #(PLAN.10)
    Executing +act: (EXPLAIN-GOAL.1)
        GOAL #(GOAL.15) explained: normal goal
    Executing +act: (EXPLAIN-PLAN.1)
        Plan #(PLAN.10) explained: intended-by #(GOAL.15)
```

End of Sentence Processing – At the end of each sentence, an attempt is made to explain each event, goal, plan, or state that occurred in the sentence. An event is explained when it is found to be part of a plan. A goal is explained by determining its motivation, noticing that it is part of an existing plan, or realizing that it is a normal goal. A plan is explained when it is found to be intended by an active goal. A

state is explained when it is found to be the result of an event, or the refinement of another state.

Once all explanation stops, the highest level concepts that result from parsing the sentence are output. Here, the sentence parses into a single event with the user as its actor, which realizes the plan of using the rm command, and is intended to to achieve the goal of removing a file. That is, AQUA now knows the user tried to remove a file, but does not know the result of the attempt.

```
RESULT OF PARSE:
     (EVENT &EVENT.13
          ACTOR    &HUMAN.2
          ACHIEVES &GOAL.15
          REALIZES &PLAN.10)
```

6.2.2 The Second Sentence

```
PROCESSING SENTENCE: THE FILE IS STILL THERE *PERIOD*

THE ==> Adding to *wm*: #(WMN.9)
  Spawning demon: (IGNOR.4 #(WMN.9))
  Executing +act: (IGNOR.4)
  Killing demon: IGNOR.4

FILE ==> Adding to *wm*: #(WMN.10)
     Created concept:
          (PHYS-OBJ &PHYS-OBJ.4
               TYPE FILE)

     Spawning demon: (MERGE-THING.3 #(WMN.10) #(Procedure 253))
     Executing +act: (MERGE-THING.3)
          #(PHYS-OBJ.4) Should Be Merged With #(PHYS-OBJ.3)
          New concept for #(WMN.10) is #(PHYS-OBJ.3)
```

Concept Reference – AQUA has to realize that this "file" refers to the same file mentioned in the previous sentence. Since there are no contradicting features between this file and the file already found in memory, AQUA assumes they are the same and merges there features. Again, the article is basically ignored.

```
IS ==> Adding to *wm*: #(WMN.11)
     Spawning demon: (HANDLE-AUX.1 #(WMN.11) PRESENT)
                Set a flag indicating an auxiliary.
          Create a new state in which the previous concept
          will have its value filled in by a later concept.
     Executing +act: (HANDLE-AUX.1)
     Spawning demon: (EXPECT.6 #(WMN.11) #(STATE.22) VALUE (STATE) AFT)
     Spawning demon: (NOTE-ITEM.4 #(WMN.11) STATE)
     Executing +act: (NOTE-ITEM.4)
          Adding STATE #(STATE.22) to explain list
```

Auxiliaries – Currently the only meaning of "is" considered by AQUA is that an object is in a state in which it has a particular property. Upon seeing "is", AQUA creates a state and then spawns demons to determine which property is being talked about; here, the property is the object's location and its value is "unchanged".

```
STILL THERE ==> Adding to *wm*: #(WMN.12)
    Spawning demon: (MODIFY-OBJECT.1 #(WMN.12) LOC UNCHANGED)
              IF  an auxiliary is present
            spawn INSERTs to modify the preceding STATE
        ELSE spawn INSERT to modify the following OBJECT
    Spawning demon: (IGNOR.5 #(WMN.12))
    Executing +act: (IGNOR.5)
    Executing +act: (MODIFY-OBJECT.1)
    Spawning demon: (INSERT.1 #(WMN.12) (STATE) PROP BEF LOC)
        IF there is a CONCEPT with the given CLASSES
        THEN Place OUR-NODE's concept (or value if specified)
             into the given SLOT in the matching concept
    Spawning demon: (INSERT.2 #(WMN.12) (STATE) VALUE BEF UNCHANGED)
    Executing +act: (INSERT.2)
        Slot "VALUE" in #(STATE.22) <-- UNCHANGED
    Executing +act: (INSERT.1)
        Slot "PROP" in #(STATE.22) <-- LOC

*PERIOD* ==> Adding to *wm*: #(WMN.13)
    Spawning demon: (EXPLAIN-CONCEPTS.2 #(WMN.13))
    Spawning demon: (IGNOR.6 #(WMN.13))
    Executing +act: (IGNOR.6)
    Executing +act: (EXPLAIN-CONCEPTS.2)
    Spawning demon: (EXPLAIN-STATE.1 #(STATE.22) #(WMN.11))
        Apply rules to try and connect the state
        to other knowledge structures in the user model
    Executing +act: (EXPLAIN-STATE.1)
        State #(STATE.22) explained:
            IF   STATE involves UNCHANGED LOCATION and
                 there is GOAL to achieve LOCATION change
            THEN mark the goal as failed, reason is STATE
```

State Explanation – AQUA explains the state of the file being in the same location as it was before by noting that this state provides information about the current state of a goal; namely, that the goal of file removal has failed. This state is not marked as *resulting-from* the event of the previous sentence, since the state did not come about because of that event. AQUA now knows the user has made a failed attempt at achieving his goal.

```
RESULT OF PARSE:
    (STATE &STATE.22
        PROP    LOC
        VALUE   UNCHANGED
        TIME    PRESENT
        TYPE    HAS-PROP
        OBJECT &PHYS-OBJ.3)
    (GOAL &GOAL.15
        ACHIEVED-BY &EVENT.13
        VALUE       &STATE.21
        TYPE        ACHIEVE-STATE
        INTENDS     &PLAN.10
        STATUS      FAILED
        INF-STATUS  &STATE.22)
```

6.2.3 The Third Sentence

```
PROCESSING SENTENCE: THE ERROR MESSAGE WAS PERMISSION DENIED *PERIOD*

THE ERROR MESSAGE ==> Adding to *wm*: #(WMN.15)
     Created concept:
          (MENTAL-OBJ &MENTAL-OBJ.5
               TYPE ERROR)

     Spawning demon: (CHECK-ERROR-STATE.1 #(WMN.15))
            IF the error message and is the object of another state that
               has a value
            THEN assume that state requires the error message

WAS ==> Adding to *wm*: #(WMN.16)
     Spawning demon: (HANDLE-AUX.2 #(WMN.16) PAST)
     Executing +act: (HANDLE-AUX.2)
     Spawning demon: (EXPECT.7 #(WMN.16) #(STATE.23) VALUE (STATE) AFT)
     Spawning demon: (NOTE-ITEM.5 #(WMN.16) STATE)
     Executing +act: (NOTE-ITEM.5)
          Adding STATE #(STATE.23) to explain list

PERMISSION DENIED ==> Adding to *wm*: #(WMN.17)
     Created concept:
          (STATE &STATE.24
               TYPE ACCESSIBLE
               MODE NEG)

     Executing +act: (EXPECT.7)
          Slot "VALUE" in #(STATE.23) <-- #(STATE.24)
     Executing +act: (CHECK-ERROR-STATE.1)

*PERIOD* ==> Adding to *wm*: #(WMN.18)
     Spawning demon: (EXPLAIN-CONCEPTS.3 #(WMN.18))
     Executing +act: (EXPLAIN-CONCEPTS.3)
     Spawning demon: (EXPLAIN-STATE.2 #(STATE.23) #(WMN.16))
     Executing +act: (EXPLAIN-STATE.2)
          State #(STATE.23) explained:
               IF   STATE is an ERROR MESSAGE and EVENT is $USE-CMD
               THEN STATE results-from EVENT
          State #(STATE.23) explained:
          IF   STATE is an ERROR MESSAGE describing another STATE
               THEN STATE is required-by that other STATE
          State #(STATE.23) explained:
               IF   STATE results-from EVENT achieving failed GOAL
               THEN STATE and states required-by it disable other STATE
```

Inferences from state descriptions – AQUA parses this sentences into the description of a state in which an object (the error message) has a particular value (insufficient permission). Using the rules we described in Chapter, AQUA is able to infer that this state results-from the failed event, that this state requires a state of insufficient permission, and that insufficient permission disables the user's plan. All of these inferences are made during the process of explanation of this state.

```
RESULT OF PARSE:
     (STATE &STATE.23
          PROP          VALUE
          VALUE         &STATE.24
          TIME          PAST
          TYPE          HAS-PROP
          OBJECT        &MENTAL-OBJ.5
          REQUIRES      &STATE.24
          RESULTS-FROM &EVENT.13)
     (STATE &STATE.24
          TYPE         ACCESSIBLE
          MODE         NEG
          REQUIRED-BY &STATE.23
          DISABLES     &PLAN.10)
```

6.2.4 The Final Sentence

```
PROCESSING SENTENCE: I CHECKED AND I OWN THE FILE *PERIOD*

I ==> Adding to *wm*: #(WMN.19)
     Created concept:
          (HUMAN &HUMAN.3)

     Spawning demon: (MERGE-THING.4 #(WMN.19) #(Procedure 246))
     Executing +act: (MERGE-THING.4)
          #(HUMAN.3) Should Be Merged With #(HUMAN.2)
          New concept for #(WMN.19) is #(HUMAN.2)
```

Concept Reference – AQUA must realize that "I" refers to the same user that the previous "I" referred to. As with "file" earlier, AQUA searches memory for a similar concept (here, a human) and merges the features of the two concepts if it finds one. This process is repeated when "I" occurs again in this sentence.

```
CHECKED ==> Adding to *wm*: #(WMN.20)
     Created concept:
          (EVENT &EVENT.14
               ACTOR    ?X
               ACHIEVES &GOAL.21
               REALIZES ?*)

     . . .
     Executing +act: (EXPECT.10)
          Slot "(ACTOR)" in #(EVENT.14) <-- #(HUMAN.2)
     Executing +act: (NOTE-ITEM.7)
          Adding GOAL #(GOAL.21) to explain list

AND I OWN ==> Adding to *wm*: #(WMN.22)
     . . .
     Created concept:
          (STATE &STATE.27
              TYPE   POSS-BY
              ACTOR  ?ACTOR
              OBJECT ?OBJ)
```

```
    . . .
    Executing +act: (EXPECT.12)
        Slot "(ACTOR)" in #(STATE.27) <-- #(HUMAN.2)
    Executing +act: (EXPECT.9)
        Slot "(ACHIEVES VALUE)" in #(EVENT.14) <-- #(STATE.27)
    Executing +act: (NOTE-ITEM.8)
        Adding STATE #(STATE.27) to explain list

THE FILE ==> Adding to *wm*: #(WMN.24)
    Created concept:
        (PHYS-OBJ &PHYS-OBJ.5
            TYPE FILE)
    Spawning demon: (MERGE-THING.6 #(WMN.24) #(Procedure 253))
    Executing +act: (EXPECT.11)
        Slot "(OBJECT)" in #(STATE.27) <-- #(PHYS-OBJ.5)
    Executing +act: (MERGE-THING.6)
        #(PHYS-OBJ.5) Should Be Merged With #(PHYS-OBJ.3)
        New concept for #(WMN.24) is #(PHYS-OBJ.3)

*PERIOD* ==> Adding to *wm*: #(WMN.25)
    Spawning demon: (EXPLAIN-CONCEPTS.4 #(WMN.25))
    . . .
    Executing +act: (EXPLAIN-CONCEPTS.4)
    Spawning demon: (EXPLAIN-STATE.4 #(STATE.27) #(WMN.22))
    Spawning demon: (EXPLAIN-GOAL.2 #(GOAL.21) #(WMN.20))
    Executing +act: (EXPLAIN-STATE.2)
        State #(STATE.23) explained:
            IF   STATE can refine another STATE and
                 other STATE is an EVENT outcome
            THEN assume STATE refines other STATE
            Assuming that user refines #(STATE.24) with #(STATE.27)
    Executing +act: (EXPLAIN-GOAL.2)
        GOAL #(GOAL.21) explained: motivated-by #(STATE.24)
```

The final conceptual representation – At this point, AQUA has built a conceptual representation that describes the user's goal, the plan realized to achieve that goal, the plan's failure, and the user's hypothesis about what caused the failure. Note that AQUA could easily answer a question such as "what does the user believe is the cause of the goal failure?" by traversing the i-links it has constructed.

```
RESULT OF PARSE:
    (GOAL &GOAL.21
        ACHIEVED-BY  &EVENT.14
        TYPE         VERIFY-STATE
        ACTOR        &HUMAN.2
        VALUE        &STATE.27
        MOTIVATED-BY &STATE.24)
    (EVENT &EVENT.14
        ACTOR    &HUMAN.2
        ACHIEVES &GOAL.21
        REALIZES ?*)
```

6.2.5 Providing the Advice

```
SEARCHING FOR REMINDINGS
     Searching for AQUA plan for #(GOAL.15)
     Found plan: #(PLAN.4)
     Found normal experience: #(PEXP-REMOVED-FILE)
     Found similar experience: #(PEXP-PERM-DENIED-FILE)
          Similar features:
               ACHIEVES
               INTENDED-BY
               RESULTS-FROM
               REQUIRES
```

Recalling a similar experience – Using the planning information attached to the user's goal, AQUA searches its memory of planning experiences trying to recall a similar episode. The user's goal is used to index AQUA's normal plan for that goal. AQUA then uses features of the user's planning experience as indices to retrieve a similar experience. Using the result of a permission denied error message indexes the desired experience.

```
TRYING TO CLASSIFY PROBLEM
     . . .
     User problem is: INCORRECT DISABLEMENT
          identical plans:
               user: #(PLAN.10)
               aqua: #(PLAN.4)
          differing disabling links:
               user: #(STATE.27)
               aqua: #(STATE.8)
     Advice:
          (#(AP.1)) Tell USER reverse of AQUA's disablement as PRECONDITION
          (#(AP.2)) Tell USER that USER's disablement is not disablement

     (AP &AP.1
          STATE &STATE.29
          GOAL  &GOAL.24
          TYPE  ENABLES)
     (AP &AP.2
          STATE &STATE.27
          GOAL  &GOAL.15
          TYPE  NOT-ENABLES)

GENERATING ADVICE
     Turning advice (#(AP.2) #(AP.1)) into English
```

To remove a file, you do not need to own it.
To remove a file, you need directory write permission.

Providing Advice – Once a similar experience has been recalled, the user's problem is determined by comparing various features of the user's problem description with the recalled experience. Here, INCORRECT DISABLEMENT is recognized because there is a goal failure, a similar plan was used, but the user has a different disablement condition. The advice heuristic is applied and the conceptual representation for the advice is created and then passed to the generator.

6.3 Trace of COMPLICATED REMOVE

Because the trace of STUBBORN FILE has illustrated the parsing process, we have deleted the parse details from the trace of COMPLICATED REMOVE and instead concentrated our explanatory effort on the planning process required to find a solution for the user.

```
> (story 'COMPLICATED-REMOVE)

PROCESSING SENTENCE: I TRIED TO REMOVE A DIRECTORY WITH RM *PERIOD*
    . . .
RESULT OF PARSE:
    (EVENT &EVENT.16
        ACTOR    &HUMAN.6
        ACHIEVES &GOAL.30
        REALIZES &PLAN.15)

PROCESSING SENTENCE: THE ERROR MESSAGE WAS DIRECTORY NOT EMPTY *PERIOD*
    . . .
RESULT OF PARSE:
    (STATE &STATE.35
        PROP          VALUE
        VALUE         &STATE.36
        TIME          PAST
        TYPE          HAS-PROP
        OBJECT        &MENTAL-OBJ.6
        CAUSED-BY     &STATE.36
        RESULTS-FROM &EVENT.16)
    (GOAL &GOAL.31
        VALUE        &STATE.37
        TYPE         ACHIEVE-STATE
        REFINES      &GOAL.30)
```

The conceptual representation – COMPLICATED REMOVE's first sentence is similar to STUBBORN FILE's first sentence, and is parsed as an event realizing the "rmdir" command in an attempt to achieve the goal of removing a directory. Its second sentence is parsed as a state of the directory not being empty when the attempt was made to remove it. From this state AQUA infers that the user's goal has been refined into "remove a nonempty directory".

```
SEARCHING FOR REMINDINGS
    Searching for AQUA plan for #(GOAL.30)
    Found plan: #(PLAN.5)
    Found normal experience: #(PEXP-REMOVED-DIR)
    No similar experience.

    Searching for AQUA plan for #(GOAL.31)
    No plan found.

    Remembering current experience under #(PLAN.5)
    Indices are:
        ACHIEVES #(GOAL.31)
        RESULTS-IN #(STATE.35)
```

Remembering new experiences – Since AQUA has not had a similar planning experience, it remembers the user's. The planning experience is indexed under the plan of using **rmdir**, with the goal (removing a nonempty directory) and the result (the error message about a nonempty directory) used as the indices.

```
TRYING TO CLASSIFY PROBLEM
     . . .
    User problem is: UNKNOWN PLAN FOR GOAL
         user: #(GOAL.31) intends ?
        aqua: #(GOAL.31) intends ?
    Advice:
         (#(AP.5)) Tell USER AQUA's plan
```

Classifying the problem – The user has two goals in this story: removing a directory, and then its refinement, removing a nonempty directory. For the first the user has the correct plan of using **rmdir**. However, the user doesn't have a plan for the more specialized goal. Unfortunately, neither does AQUA, so it invokes its planning component to try and find a solution.

6.3.1 Planning for COMPLICATED REMOVE

```
INVOKING PLANNER

Planning for #(GOAL.31)
          (GOAL &GOAL.31
               ACHIEVED-BY &EVENT.16
               VALUE       &STATE.35
               TYPE        ACHIEVE-STATE
               INTENDS     &PLAN.15)
     Starting with experience #(EVENT.16)
     Recognized planning situation UNDO-DISABLEMENT
          GOAL G1 has feature X, fails
          GOAL G2 without feature X has known plan P
     Applying UNDO-DISABLEMENT
          (1) Create Plan: UNDO feature X,Use plan P
     Applying Strategies Returns #(PLAN.22)
```

Selecting a planning strategy – Because there is no workable plan for the user's current goal, AQUA attempts to create a new plan by applying planning strategies to the current situation. UNDO-DISABLEMENT is appropriate, suggesting a two step plan: empty the directory, and then use **rmdir**.

```
Planning for subgoals of #(PLAN.22)
Planning for #(GOAL.43)
          (GOAL &GOAL.43
               VALUE &STATE.53
               TYPE  ACHIEVE-STATE)
     Starting with experience ...none found
     Recognized planning situation GENERALIZE-GOAL-FEATURE
          GOAL G1 has no planning experience in memory, G1 modifies an object
     Applying GENERALIZE-GOAL-FEATURE
          (1) Generalize object
          (2) Search for plan
          (3) Instantiate plan
     Applying Strategies Returns #(PLAN.24)
```

Planning for subgoals – As might be expected, the planner is recursively invoked to plan for subgoals. Searching memory for a plan that empties a directory fails, so a planning strategy must be applied to continue the planning process. For this situation, GENERALIZE-GOAL-FEATURE is appropriate, and suggests generalizing the goal of emptying a directory to emptying a physical object, searching memory for a generalized plan for this goal, and then instantiating the object in this plan to a directory. applying this strategy results in another two step plan: know the contents of the directory, and then remove them. These subgoals must now be planned for.

```
Planning for subgoals of #(PLAN.24)
Planning for goal #(GOAL.47)
          (GOAL &GOAL.47
               TYPE  D-KNOW
               VALUE &STATE.63)
     Starting with experience...#(PEXP-ASK-MEMORY)
     #(PLAN.25) marked as execute immediately
     Executing #(PLAN.25) ...result is #(STATE.68)
     Instantiating #(PLAN.24) with #(STATE.68)
     Applying Strategies Returns #(STATE.68)
Plan for goal #(GOAL.47) is already executed.

Planning for goal #(GOAL.48)
          (GOAL &GOAL.48
               TYPE  ACHIEVE-STATE
               VALUE &STATE.64)
     Starting with experience #(PEXP-REMOVED-FILE)
     Applying Strategies Returns #(PLAN.26)

Planning for subgoals of #(PLAN.26)
No subgoals of #(PLAN.26)

Plan for goal #(GOAL.48) is #(PLAN.26)
End planning for subgoals of #(PLAN.24)

Plan for goal #(GOAL.43) is (#(PLAN.26) #(PLAN.21))
End planning for subgoals of #(PLAN.22)

Newly created plan for #(GOAL.30) is (#(PLAN.26) #(PLAN.21))
```

Immediate plan execution – The first subgoal is to know the content's of the directory. AQUA finds a plan for doing this: searching its memory for the information. Since this plan is marked to be executed immediately, AQUA does so, and remembers that directories contain files. The plan containing the subgoal is instantiated with this information, and the next subgoal, removing files, is planned for. Searching memory finds the plan for this subgoal, using **rm**. AQUA now has a plan for the user's goal, and proceeds to tell the user about its newly created plan.

```
     (AP &AP.5
          GOAL &GOAL.30
          PLAN (&PLAN.26 &PLAN.21)
          TYPE INTENDS)

GENERATING ADVICE
     Turning advice (#(AP.5)) into English
```

To remove a nonempty directory, use the "rm" command to empty the directory.
Then use the "rmdir" command to remove the directory.

7. Conclusions and Future Work

We have presented a model of human problem understanding and advice giving, and have described a program named AQUA that embodies this model. We now summarize our results and discuss directions for future research.

7.1 Conclusions

We have viewed advice giving as a three step process: (1) understand the problem, (2) recall a similar, previous experience, whose solution is applicable to the current situation, and (3) compare the remembered experience with the problematic experience to decide what to actually generate as advice. Thus, advice giving is driven by memory search, and only when a similar experience cannot be recalled or does not provide a solution is it necessary to do any novel planning.

We have described various heuristic rules that an advice giver can use to build a conceptual representation of a user's problem. These rules allow the user's goals, the motivations for those goals, and the results of plan executions to be inferred. Once a conceptual representation has been built, the advice giver can use features of the user's problem, such as the plan tried and the results of its execution, to try and index similar experiences in its memory. The more experiences the advice giver has had, the more likely it is that a similar experience will be found. Experts have more experiences, and are therefore more likely to find an experience that can provide a solution.

We have claimed that once the advice giver has been reminded of a similar episode, the user's problem, the expert's remindings, and their differences can be used to classify the user's problem and to index heuristics that determine appropriate advice. We identified nine classes of user problems, provided advice heuristics for each class, and provided rules for recognizing the situations in which preventive advice should be given.

But, when a problem is described to us we don't always get reminded of a similar experience and sometimes even when we do the reminding does not provide a solution. However, in many situations, by combining various planning heuristics along with the information provided by the user, it is still possible to provide a solution. We have presented four planning strategies people appear to use in creating new plans and have demonstrated their effectiveness in planning.

We have argued that remindings drive both the advice giving and planning process. To provide advice, memory must be organized so that similar experiences can be retrieved accurately and efficiently. Failures and their solutions must be remembered so that similar failures in the future are resolved easily. We have suggested a memory organization for planning experiences. Experiences of using a particular plan in different situations are indexed by their difference from the prototypical planning experience with the plan. Failed planning experiences have a pointer to their solution.

7.1.1 AQUA Versus Current Computer Documentation

Our early experience in building AQUA has led us to believe that current computer documentation is not indexed appropriately. Most computer manuals have a single index that alphabetically lists the names of various plans, along with a pointer to the plan's description. The UNIX manual, for example, has an alphabetical index listing the names of all of the commands, along with the page number of the description of what the command does. However, this indexing scheme is inappropriate for a user who knows only his goal but does not know which plan to use. Adding an additional index of user goals would improve documentation. For example, the entries in the goal index would be "remove a file," "create a file," and so on, rather than **rm** and **touch**.

Another problem with current documentation is that the descriptions of plans rarely describe failed experiences with the plan. Under each plan there should be a list of its common failures and their results, together with a pointer to an explanation for the failure and a suggested solution. With this organization, when a planning failure occurred users could simply examine the manual, using a combination of their plan and their failed experience to index the explanation and solution. Such an organization would make it much easier for people to solve their own problems without doing complete searches of inappropriately indexed documentation or having to ask an expert.

7.2 Future Work

Unfortunately, in building AQUA we have not solved the "advice problem" — that is, although we have new insight into how people understand problems, provide advice, and create novel solutions for problems, we do not yet have a complete model of how people perform these tasks. We now discuss some of the areas left open for future research.

7.2.1 Deciding How Much Advice To Give

The appropriateness of a response depends on the perceived level of the user's expertise. As an example, consider the following story and several possible responses AQUA could give:

WRONG CREATE

USER: I tried to create an empty file "stuff" with "vi", but "vi" always creates a file with at least one character in it. Any suggestions?

AQUA: Use the "echo" command to create the file. The "vi" command is used to edit files.

AQUA: Use "echo > stuff" to create an empty file "stuff". The "vi" command is used to edit files.

The first response clearly assumes more user expertise than the second, providing a pointer to an appropriate plan instead of a fully instantiated plan. We can also imagine responses that explain what > does or why **echo** is used instead of some other command. Clearly, a model of the user that includes inferences about the user's level of expertise is necessary to determine the level of detail and explanation that should be provided in the solution.

7.2.2 Integrating the Advice Giving Process

Currently AQUA views the process of advice-giving as being composed of distinct steps: understand the problem, recall similar episodes, classify the problem, and provide advice. It is clear, however, that people work on all these steps simultaneously. For example, consider the following story:

PRINTING PROBLEM

USER: I tried to print my file on the laser writer using the "lpr" command. There was no output on the laser writer, even though I tried several different options.

AQUA: The "lpr" command causes output to go the printer, not the laser writer. Use "enscript" to print on the laser writer.

In informal protocols experts decided **enscript** was the appropriate plan to use and were reminded that **lpr**'s output appears on the printer, *before* they had processed the entire story. Therefore AQUA must be modified to allow remindings to occur while it is building the user model. Information from these remindings can also provide top-down expectations that aid in building the user model. For example, remembering that **lpr**'s output goes to the printer easily explains why there was no output on the laser writer, without the need for additional inferencing.

7.2.3 Remembering and Generalizing Problem Solutions

Solutions for one situation may be reasonable for other situations as well. Consider the following story:

DASHED

USER: I tried to remove a file named "-stuff" but "rm" failed.

AQUA: "-stuff" is treated as an option instead of a file name. Use the command "rm ./-stuff" to remove the file named "-stuff".

Suppose AQUA did not know this solution before being asked the question but somehow arrived at it. AQUA should then be able to solve the following user's problem:

DASHED AGAIN

USER: I tried to remove a directory named "-foo" but "rmdir" failed.

AQUA: "-foo" is an option, not a file name. Use the command "rmdir ./-foo" to remove the directory named "-foo".

Since the problems are similar, DASHED AGAIN should remind AQUA of DASHED and its solution.

In addition, at some point the solution of prefacing a name with ./ should be generalized as solving the problem of referring to a file whose name begins with a special character, instead of just working with **rm** or commands that remove a file. This solution would then be available for situations such as the following:

ANOTHER STUBBORN FILE

USER: I tried to list a file named "~foo" but the "cp" command failed and printed "unknown user".

AQUA: "~foo" is a user name, not a file name. Use the command "cat ./~foo" to list a file named "~foo".

AQUA's planner currently uses generalized plans in creating new plans; however, there is no model of how those generalized plans were created. How to generalizing the applicability of a planning problem's solution and index it appropriately is an important area of future research.

7.2.4 Other Issues

By examining protocols of simple planning problems, we have discovered four planning strategies people use and implemented several of these. As we examine more complex planning problems, more strategies will be discovered and need to be integrated into the planner. There are also several unanswered questions involving planning strategies: How many other planning strategies do people have? How are they indexed in memory? Do they serve to index planning experiences?

AQUA's planner currently arrives at a single solution to a given problem. However, people frequently arrive at a workable plan but continue planning in an attempt to improve the plan or to find alternate plans. AQUA's planner should be modified to model this behavior. To do this, the planner can be extended to apply all planning strategies that are applicable to a given experience. In addition, AQUA currently indexes only the plan created for the highest level goal. However, the subplans create for novel subgoals encountered during the planning process should also be remembered. For example, in COMPLICATED REMOVE a subgoal of emptying a directory arises and a plan of using **rm** to remove the subgoals in the directories is created and should be remembered.

Finally, there is a need for psychological validation of our theories of memory organization and planning. One interesting experiment would be to record the planning experiences of a novice user learning to use UNIX. Then, the same planning situations can be given to AQUA and its performance compared with the user's.

References

[Alterman 1985] R. Alterman, "Adaptive Planning: Refitting Old Plans to New Situations," *Proceedings of the Seventh Annual Conference of the Cognitive Science Society* (August 1985).

[Carbonell 1983] J.G. Carbonell, "Learning by Analogy: Formulating and Generalizing Plans from Past Experience," in *Machine Learning: An Artificial Intelligence Approach*, ed. T. Mitchell, Tioga Publishing Co., Palo Alto, California (1983).

[Charniak 1980] E. Charniak, C. Riesbeck, and D. McDermott, *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Hillsdale, New Jersey (1980).

[Cullingford 1978] R.E. Cullingford, "Script Application: Computer Understanding of Newspaper Stories," Ph.D.Thesis, Technical Report #116, Yale University, Department of Computer Science (1978).

[Dyer 1983] M.G. Dyer, *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension*, MIT Press, Cambridge, Massachusetts (1983).

[Faletti 1982] J. Faletti, "PANDORA: A Program for Doing Common Sense Planning in Complex Situations," *Proceedings of the National Conference on Artificial Intelligence* (August 1982).

[Hammond 1983] K.J. Hammond, "Planning and Goal Interaction: The use of past solutions in present situations," pp. 148-151 in *Proceedings The National Conference on Artificial Intelligence*, Washington, D.C. (August 1983).

[Johnson 1984] W. Lewis Johnson and Elliot Soloway, "Intention-Based Diagnosis of Programming Errors," pp. 162-168 in *Proceedings National Conference on Artificial Intelligence*, Austin, Texas (August 1984).

[Kay 1985] Dana S. Kay and John B. Black, "The Evolution of Knowledge Representations with Increasing Expertise in Using Systems," pp. 140-149 in *Proceedings Cognitive Science Society*, Irvine, California (August 1985).

[Kolodner 1984] J.L. Kolodner, *Retrieval and Organization Strategies in Conceptual Memory: A Computer Model*, Lawrence Erhbaum Associates, Hillsdale, New Jersey (1984).

[Lebowitz 1980] M. Lebowitz, "Generalization and Memory in an Integrated Understanding System," Ph.D.Thesis, Technical Report #186, Yale University, Department of Computer Science (1980).

[Quilici 1985] A. Quilici, J. Reeves, and S. Turner, "Rhapsody: Yet Another Graphical AI Tool," Technical Note UCLA-AI-N-85-?, Artificial Intelligence Laboratory, University of California, Los Angeles (1985). Release date uncertain.

[Rees 1984] J.A. Rees, N.L. Adams, and J.R. Meehan, *The T Manual*, Yale University, Department of Computer Science (1984).

[Reeves 1985] J. Reeves, "RHAP: A Natural Language Generator," Technical Report UCLA-AI-R-85-?, Artificial Intelligence Laboratory, University of California, Los Angeles (1985). Release date uncertain.

[Sacerdoti 1974] E. Sacerdoti, "Planning in a Hiearchy of Abstraction Spaces," *Artificial Intelligence*(5) (1974).

[Schank 1972] R.C. Schank, "Conceptual Dependency: A Theory of Natural Language Understanding," *Cognitive Pyschology* 3(4) (1972).

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

[Schank 1977] R.C. Schank and R.P. Abelson, *Scripts, Plans, Goals, and Understanding: An Inquiry into Human Knowledge Structures*, Lawrence Erhbaum Associates, Hillsdale, New Jersey (1977).

[Schank 1982] R.C. Schank, *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*, Cambridge University Press, Cambridge, Massachusetts (1982).

[Wilensky 1978] R. Wilensky, "Understanding Goal Based Stories," Ph.D.Thesis, Technical Report #140, Yale University, Department of Computer Science (1978).

[Wilensky 1982] R. Wilensky, "Talking to UNIX in English: An Overview of UC," *Proceedings of the National Conference on Artificial Intelligence* (August 1982).

[Wilensky 1983] R. Wilensky, *Planning and Understanding: A Computational Approach to Human Reasoning*, Addison-Wesley, Reading, Massachusetts (1983).

[Wilensky 1984] R. Wilensky, Y. Arens, and D. Chin, "Talking to UNIX in English: An Overview of UC," *Communications of the ACM* 27(6) (1984).