Artificial Intelligence Review **14:** 283–331, 2000.
Stephen J. Hegner et al. (eds.), Intelligent help systems for UNIX.

Planning Intelligent Responses in a Natural Language System

DAVID N. CHIN
Department of Information and Computer Sciences, University of Hawaii, 1680 East West Rd., Honolulu, HI 96822, U.S.A. E-mail: chin@hawaii.edu

Abstract. Intelligent help systems cannot merely respond passively to the user's commands and queries. They need to be able to volunteer information, correct user misconceptions, and reject unethical requests when appropriate. In order to do these things, a system must be designed as an intelligent agent. That is, a system needs to have its own goals and then plan for these goals. A system which did not have its own goals would never refuse to help users perform unethical actions. Such an intelligent agent has been implemented in the UCEgo component of UC (Wilensky et al. 1984; Wilensky et al. 1988) (UNIX Consultant), a natural language system that helps the user solve problems in using the UNIX operating system. UCEgo provides UC with its own goals and plans. By adopting different goals in different situations, UCEgo creates and executes different plans, enabling it to interact appropriately with the user. UCEgo adopts goals when it notices that the user either lacks necessary knowledge, or has incorrect beliefs. In these cases, UCEgo plans to volunteer information or correct the user's misconception as appropriate. These plans are pre-stored skeletal plans that are indexed under the types of situations in which they are typically useful. Plan suggestion situations include the goal which the plan is used to achieve, the preconditions of the plan, and *appropriateness conditions* for the plan. Indexing plans by situations improves efficiency and allows UC to respond appropriately to the user in real time. Detecting situations in which a plan should be suggested or a goal adopted is implemented using *if-detected daemons*. These daemons provide a single mechanism which can be used both for detecting goals and suggesting plans. Different methodologies for the efficient implementation of if-detected daemons are discussed.

Keywords: agent, daemon, intention, meta-goal, meta-planning, planning, speech act, UNIX

1. Introduction

Consider the problem of building a program that simulates a human consultant. A user would be able to come up to such a program and obtain advice in the program's domain of expertise by entering queries in English (or some other natural language). The consultant program would then provide solutions in English. A user might ask for advice about how to do things, for definitions of terminology, or for advice in solving problems. In short, this program would behave like a real human consultant.

In order to build such a system, one needs to satisfy at least three requirements. First, the computer system needs to be able to understand the user's

queries. Next, the program must be able to solve the user's problems and formulate an answer. Finally, the system must be able to convey the solution to the user in a clear, concise manner. Needless to say, there are many difficult and unsolved problems in each of these areas. The first requirement, understanding the user, involves the whole of natural language understanding, a difficult area of artificial intelligence research. The second requirement, problem solving, has a long continuing history of research in AI. The last requirement, communicating the answer to the user, has a shorter history of research in AI, but is no less difficult a problem.

However, even if all of the problems in each of the three areas were to be solved, and one could build a natural language consultation system that did each of the three tasks perfectly, that would still not be enough for a good natural language consultation system. A good consultation system also needs to be able to take the initiative in a dialog with the user, rather than always responding passively to the user. For instance, consider the following user interaction with a hypothetical program that provides advice on using the UNIX[1] operating system:

User: What does ls -v do?

Program: It lists the contents of your current directory.

The hypothetical program gives an answer that is literally correct, since the ls command actually ignores inappropriate flags such as -v. However, a consultant that provides only the above answer has failed to correct the user's incorrect preconception that the ls command has a -v flag. So, although the user did not specifically ask whether ls has a -v flag, a good consultant would not fail to provide the information that in fact ls does not have such a flag. Such a response is shown in the next dialog:

User: What does ls -v do?

Program: There is no -v option for ls.

This is a better answer even though it literally does not answer the user's question. In deciding to ignore the user's direct question and shift its attention to the user's misconception, the program has taken the initiative in the dialog. This shows that a good consultant cannot just passively respond to the user; rather, it must have its own agenda.

In the previous dialog, the better answer requires that the program realize that the user has a misconception. To do this, the system must first infer that the user believes that ls has a -v option and then realize that the user's belief conflicts with the program's knowledge. So, in general, a good consultant system must also model the user's knowledge and beliefs.

Even after a consultant system realizes that the user has a misconception, it must decide how to deal with the misconception. In the above example, the

system decides that it should inform the user of the facts in order to correct the user's misconception. In other cases, the system may choose to ignore the misconception, as in the following scenario:

User: How can I delete someone else's file when I don't have write permission on the file?

Program: I will not help you delete someone else's file because that is unethical.

In the user's statement above, the user has assumed that one needs write permission on the file to delete it. This is not true. Rather, one needs write permission on the parent directory to delete the file. Regardless of what is the correct precondition, the program decides not to help the user because of ethical considerations. This also means that the program decides not to correct the user's misconception, so as to avoid helping the user delete someone else's file. This is an example of a decision by a consultant program to be uncooperative.

Of course a good consultant program cannot arbitrarily decide to be uncooperative. In the previous case, the decision to be uncooperative was based on the fact that the user's goal of deleting someone else's file conflicts with the program's goal of preserving all users' files. In this case, the program's goal of preserving files wins out over the program's desire to help the user who asked the question. These sorts of goals and goal interactions are needed to guide a consultant system properly.

2. UC

UC (UNIX Consultant) (Wilensky et al. 1984; Wilensky et al. 1988), is a natural language consultation program that answers the user's questions on how to use the UNIX operating system. After understanding the user's query, the UCEgo component determines how UC will respond to the user by detecting appropriate goals, planning for those goals, and then executing the plan. Detecting appropriate goals is described in more detail in (Chin 1987; Chin 1991). This paper describes the process of planning and execution in UCEgo.

3. Planning Speech Acts

Natural language systems act primarily by communicating with the user. These communicative actions are called *speech acts* (Austin 1962; Searle 1969). A planner that produces plans consisting of speech acts has somewhat

different requirements than other types of planners. First of all, speech act planners need to perform in real time in order to carry out a dialog with the user. This implies that such planners need to avoid inefficient search and backtracking by using real world knowledge to guide the planning process.

3.1. *Other planners*

Planning has a long history in AI, starting from its origins as search within a problem space. In the GPS means-ends analysis formalism of (Newell and Simon 1963), a planner searches for a sequence of operators that allows the planner to move from an initial state in the problem space to the goal state of the planner. STRIPS (Fikes and Nilsson 1971) is an early example of a planner based on means-ends analysis. ABSTRIPS (Sacerdoti 1974) extended the formalism to work in hierarchical problem spaces. ABSTRIPS broke down a planning problem into a hierarchy of sub-problems and solved each sub-problem independently. Since sub-problems may not actually be independent, planners were developed by (Sussman 1975; Tate 1975; Warren 1974; Waldinger 1977; Sacerdoti 1977; Stefik 1980) and others that could handle planning when the ordering of plan steps is critical to the success of a plan. KAMP (Appelt 1981) applied this type of planner to the problem of planning natural language utterances. KAMP planned English sentences from the speech act level down to selection of actual words.

The previous types of planners develop plans from scratch. This presents a problem, since such planning is computationally expensive. For example, it was not unusual for KAMP to take several hours to plan a complex utterance. Indeed, Appelt developed the TELEGRAM unification grammar (Appelt 1983) to improve the efficiency and modularity of planning at the linguistic level. Developing plans from scratch using "weak methods" such as search and theorem-proving leads to inefficient back-tracking and computationally expensive checking of preconditions for every plan step. These methods do not take advantage of available domain knowledge about the types of plans that are applicable in different situations.

Another problem with general purpose planners that use "weak methods" is that their full computational power is not needed in planning speech acts. The OSCAR speech act planner (Cohen 1978) showed that a very simple planner that did not backtrack was sufficient for planning basic speech acts. Although OSCAR did not actually produce natural language output, it did plan speech acts at the conceptual level in enough detail to demonstrate the computational theory of speech act generation devised by Cohen and Perrault 1979). However, since OSCAR did not have to produce speech acts in real time in order to sustain a dialog with a user, it did not worry about how to plan efficiently. Given a goal, OSCAR merely looped through an ordered list of all

potential actions until it found one whose effects matched the goal. Only after deciding upon an action would OSCAR test the preconditions of the plan and adopt as sub-goals any preconditions that OSCAR did not believe to already be true. If OSCAR could not satisfy a precondition sub-goal, then it failed in planning since it did not backtrack to try different actions. The fact that OSCAR worked fairly well despite this seemingly severe limitation shows that planning speech acts does not usually require complex planning.

An alternative to planning from scratch using "weak methods" is presented by (Schank and Abelson 1977) in their theory of scripts and plans. In their theory, a script consists of a very specific series of steps, while a plan is more abstract and includes other components, such as preconditions on the use of the plan. The TALE-SPIN story generator (Meehan 1976; Meehan 1981) implemented this theory to produce plans for the characters in its stories. (Friedland 1980) and (Friedland and Iwasaki 1985) describe the MOLGEN and SPEX planners, which extended the idea of scripts and plans into a hierarchy of *skeletal plans*. Skeletal plans are pre-stored plans whose plan-steps may vary in generality from specific actions as in scripts to abstract sub-goals. They are similar to the MACROPs of STRIPS and the chunks of (Rosenbloom and Newell 1982). However, the latter systems emphasized learning chunks or MACROPs rather than the selection and instantiation of abstract plans. In a similar vein, (Carbonell 1986; Kolodner et al. 1985; Alterman 1986; Hammond 1986) have worked on adapting previous plans to new situations using techniques such as analogical reasoning, case-based reasoning, and utilization of a knowledge hierarchy to generalize old plan-steps and then respecify them to form new plan-steps.

Using pre-stored skeletal plans makes for a much more efficient planner than one that has to plan from scratch using weak methods. However, the use of pre-stored plans presents a different efficiency problem: how to find the right plan for a specific situation. TALE-SPIN indexed scripts and plans under the goal of the script/plan and then looped through all possibilities, checking the most specific scripts first, until a workable plan was found. Similarly, MOLGEN and SPEX only looked at the UTILITY slot, i.e., the goal, of the skeletal plan. Since skeletal plans of MOLGEN and SPEX did not have preconditions, the planners could not even consider a plan's preconditions to eliminate unsuitable plans. Instead, the planners considered all skeletal plans that fit the goal and returned all proper instantiations of those plans for the user's consideration.

(Hendler 1985) describes SCRAPS, a planner that used marker-passing to help make more efficient choices during planning. The marker-passing mechanism detected plan choices that might result in intra-plan sub-goal conflicts early in the planning process. The planner could then avoid the potential

conflict by choosing an alternative and hence avoid inefficient backtracking. For example, SCRAPS was able to avoid backtracking while planning in the following situation:

> You are on a business trip (in a distant city). You wish to purchase a cleaver.

By passing markers from BUYING, *ME*, and CLEAVER-27, SCRAPS found the following intersection path:

> BUYING ⇒ TAKE-TRIP ⇒ PLANE ⇒ BOARDING ⇒ WEAPONS-CHECK ⇒ IF you go through WEAPONS-CHECK with a WEAPON, you get arrested ⇐ WEAPON ⇐ CLEAVER ⇐ CLEAVER-27

This path was then evaluated to determine that it represents a negative interaction, because it is a "bad thing" to get arrested. As a result, SCRAPS ruled out the choice of taking a PLANE (taking a BUS or TRAIN instead), and avoids the backtracking that would be required if the planner had chosen to take a PLANE. One of the problems with such a scheme is the length of the marker-passing path needed to detect important intersections. As longer paths are considered, there are more and more spurious intersections. If a marker-passer were to consider only short paths, then would it run the risk of missing important longer path-length intersections. With a path length of eight as in the previous example, any reasonably large knowledge base would produce a very large number of intersections, most of which would be spurious. Even if marker-passing were implemented in parallel, and all of the resulting intersections checked in parallel, it is uncertain whether it would be more efficient to plan using marker-passing or simply to plan in parallel (e.g., a planner could consider traveling by PLANE, BUS, and TRAIN in the previous example in parallel, and then evaluate the best plan later). With current serial machines, marker-passing would undoubtedly be less efficient.

Another problem with SCRAPS is that it ignores other criteria for making choices besides potential negative sub-goal interactions. For example, choosing among the various travel plans in Hendler's example should depend on much more than buying a cleaver. One would not want to take a bus or train if the business trip in the distant city were on the opposite coast of the United States. Choice criteria such as the distance of the trip are not preconditions in the sense that one could still take a bus or train for long distance travel, although one would not want to do so, unless one suffered from fear of flying or lacked the money for a plane ticket. In fact, most people would rather abandon the goal of buying a cleaver, rather than abandon the plan of taking the plane home from a distant city. Of course, most people in that situation would fix their plans by mailing the cleaver home or putting it in

checked baggage (to avoid going through weapons-check with the cleaver). However, the latter is not a fair criticism of SCRAPS, since SCRAPS was not programmed with the knowledge needed to make such plan fixes.

3.2. *Planning in UCEgo*

UCEgo attacks the problem of efficient planning of speech acts in several ways. First of all, like OSCAR, UCEgo uses a very simple planner that avoids inefficient backtracking. Secondly, like Meehan's TALE-SPIN, Friedland's MOLGEN planner, and Iwasaki's SPEX, UCEgo uses pre-stored skeletal plans that efficiently encode knowledge about planning for dialog. However, unlike those planners, UCEgo selects for consideration only those skeletal plans that are appropriate to the current situation. A plan is judged appropriate if the goal of the plan is currently a goal of the system, and also if the preconditions and *appropriateness conditions* for the plan are satisfied by being true in the current situation. UCEgo indexes plans according to the type of situation in which the plans are useful. As a result, UCEgo does not waste resources in considering pre-stored plans that are inappropriate and hence will fail.

Another difference between UCEgo and other planners is that UCEgo uses the idea of meta-planning developed by (Wilensky 1983) and first implemented in part by (Faletti 1982). Through meta-planning, UCEgo is able to handle positive and negative interactions among UCEgo's goals and plans. This notion of meta-planning is different from that of (Stefik 1981), who used "meta-planning" in reference to his MOLGEN planner. Our notion of meta-planning involves the recursive application of a planner to solve its own planning problems (problems such as interactions among goals or plans). On the other hand, (Stefik 1981) used "meta-planning" to refer to a multi-layered control structure that allows a planner to schedule its own planning tasks using a variety of strategies, such as least-commitment or application of heuristics. When planning problems arose, MOLGEN could only backtrack. It could not apply itself recursively to create and execute plans for handling the planning problems.

The rest of this paper describes the processes of planning and plan execution in UCEgo. Section 3.3 describes plan selection in UCEgo, giving details on the types of situations in which different types of plans are suggested. Plan execution and other types of simple reasoning in UCEgo are described in Section 3.4.

3.3. *Plan selection*

UCEgo selects plans based on the current situation. Every plan in UCEgo has one or more associated *situation classes*. When the situation class associated with a plan matches the current situation, that plan is suggested to UCEgo. The suggestion of plans based on situations is implemented using *if-detected daemons*. If-detected daemons can be considered tiny inference engines that look for particular classes of situations. When the current situation matches the situation class of a daemon, it performs appropriate actions such as suggesting a plan.

A simple example of an if-detected daemon used to suggest plans is shown in Figure 1. This daemon suggests the plan (PLANFOR1) of having UC exit (UC-exit1) whenever UC has the goal (UC-HAS-GOAL1) of exiting. The arrow(s) from the double-concentric circles labeled "if-detected" point to the *detection-net* of the daemon and the arrows pointing to the double-concentric if-detected circles represent the *addition-net*. The detection-net is somewhat like the antecedent of a rule and the addition-net is somewhat like the consequent of a rule.

3.3.1. *Situation types*

Situations that suggest plans consist of many different types of information. A plan situation always includes the main goal of the plan. It may also include preconditions and other appropriateness conditions. For example, the plan suggestion situation for a USE-CHAIN-SAW plan might include the following appropriateness condition: need to cut thick branch (over 1 inch diameter). This appropriateness condition is not a precondition, since chain saws can be used to cut smaller branches. Indeed, when a user has already started using a chain saw to cut some thick branches, the user will often use the chain saw for smaller branches. However, if one had only small branches to trim, one would not think of using a chain saw (hedge shears are more appropriate). Adding such appropriateness conditions to a plan suggestion situation prevents the suggestion of inappropriate plans.

The plan suggestion situations in UC can be divided into four main categories. These are situation classes that suggest:

1. inform-plans
2. request-plans
3. social-plans
4. meta-plans

Situations that suggest inform-plans comprise those situations in which the planner wishes to inform the user of some information. Request-planning situations are those in which the planner wishes to request information from the user. Situations that invoke social-plans include salutations and apolo-

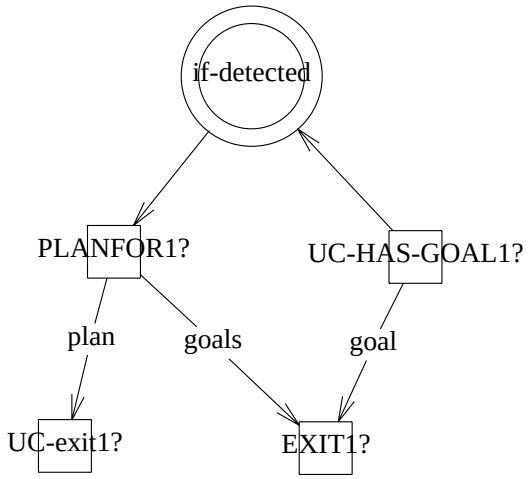

Figure 1. Suggest plan of executing the UC-exit procedure when UC wants to exit.

gies. Meta-planning situations involve suggesting meta-plans for dealing with meta-goals. Each of these situation classes and the plans that are suggested to deal with those classes of situation are described in the following sections.

3.3.2. *Inform-plans*

UCEgo suggests inform-plans whenever UC has the goal of having the user know something. There are two situation classes in which inform-plans are detected. The two classes are distinguished by the type of information that UC wants the user to know. If the knowledge is a network of real concepts, then UCEgo simply suggests the plan of communicating those concepts to the user. On the other hand, if UC wants the user to know something that is only a description of some concept(s), then UC should communicate to the user the concept(s) that is the referent of the description. For example, if UC wants the user to know "how to delete files," then UC should inform the user that "the rm command is used to delete files." "How to delete files" is a description of the concepts, "the rm command is used to delete files." If UC were to communicate just the description to the user, that would not achieve UC's goal of having the user know how to delete a file. UC needs to communicate the referent of the description to the user. As a result, UC needs to compute the referent before informing the user in situations where the type of information is a description.

The two situation classes that suggest inform-plans are summarized in Table 1. The first part of the situation in both situation classes is the planner's goal of having the user know something. The second part of the situation in the first class represents the precondition that the information should not be

a description. The opposite is the case in the second class. Also, the second class of situation has the additional precondition that the description must have an identified referent.

The actual if-detected daemons that detect inform-plan situations and suggest the plans are shown in Figures 2 and 3. Figure 2 shows the if-detected daemon for detecting situations in which UC has the goal (UC-HAS-GOAL3) of having the user know (KNOW1) something (SOMETHING1) that is not a description. Since descriptions in UC are always hypothetical (i.e., they cannot serve as referents), they can be detected by checking to make sure that the concept to be communicated is not hypothetical. Since hypothetical concepts are marked as being dominated by HYPOTHETICAL, this means that the if-detected daemon should check to make sure that whatever matches SOMETHING1 is not dominated by HYPOTHETICAL. This check is indicated by the NOT link from DOMINATE1 in Figure 2.

In the second class of situations, UC wants the user to know a description for which UC has identified a referent. Figure 3 shows the if-detected daemon for detecting situations in which UC has the goal (UC-HAS-GOAL2) of having the user know (KNOW1) something (SOMETHING1) that is a description. The referent of the description is indicated by the ANSWER-FOR relation between SOMETHING1 (the description) and SOMETHING2 (the referent). There is no explicit check to make sure that SOMETHING1 is indeed a description, because only descriptions participate in the ANSWER-FOR relation.

3.3.3. *Request-plans*

The request-plan is suggested by UCEgo whenever UC wants to know something, ?x, and additionally, UC believes that the user is likely to know ?x. The latter condition is not a precondition in the classical sense, since UC can still use the plan of asking the user even when UC does not believe that it is likely to work. Also, the fact that UC does not believe that the user knows the information does not mean that the plan will necessarily fail, since UC may be mistaken in its beliefs about what the user knows. In fact, when they have no better alternatives (or are just too lazy to try other alternatives), people will often ask someone else for information even when they believe that it is very unlikely that person knows the information. Thus the condition that one only asks someone one believes knows the information should not preclude use of this plan. This contradicts Schank and Abelson's 1977 approach in which this is termed an *uncontrollable precondition*, meaning that this plan is always aborted, unless this precondition is true. Nevertheless, one does not normally ask someone for information when one does not believe that this person possesses the information. This is an example of an appropriateness condition

Table 1. Situations that suggest inform-plans.

Situation	Suggested Plan
Planner wants the user to know ?x, and ?x is not a description	tell the user ?x
Planner wants the user to know ?x, and ?x is a description, and ?y is the referent of ?x	tell the user ?y

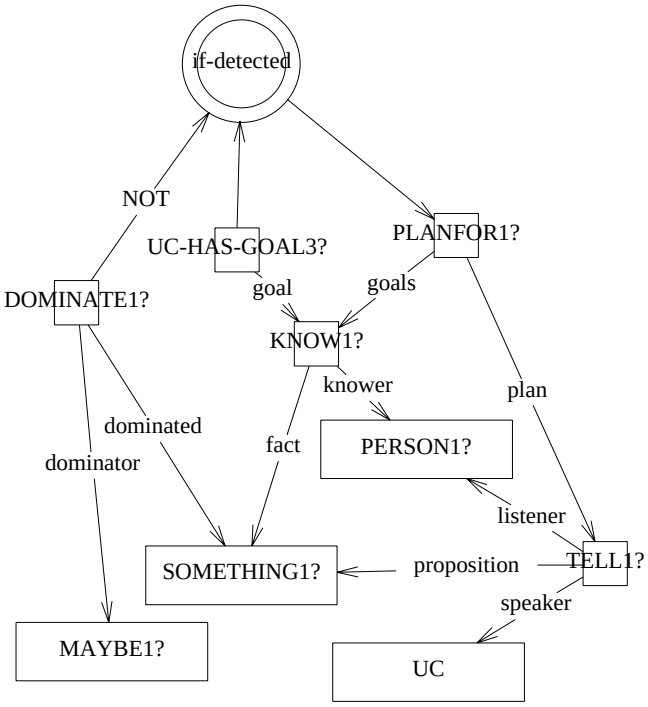

Figure 2. Suggest plan of telling user when UC wants the user to know a real concept.

for the use of a plan. UCEgo will not suggest the plan of asking someone for information unless UC believes that the person knows the information sought by UC.

Whether or not the user knows the information sought by UC is modeled by KNOME, UC's user modeling component (Chin 1987; Chin 1989). Since such information is often not represented explicitly in UC's knowledge base, but instead is inferable from the user's level of expertise, a call to KNOME is needed to determine whether or not the user knows something. Hence in

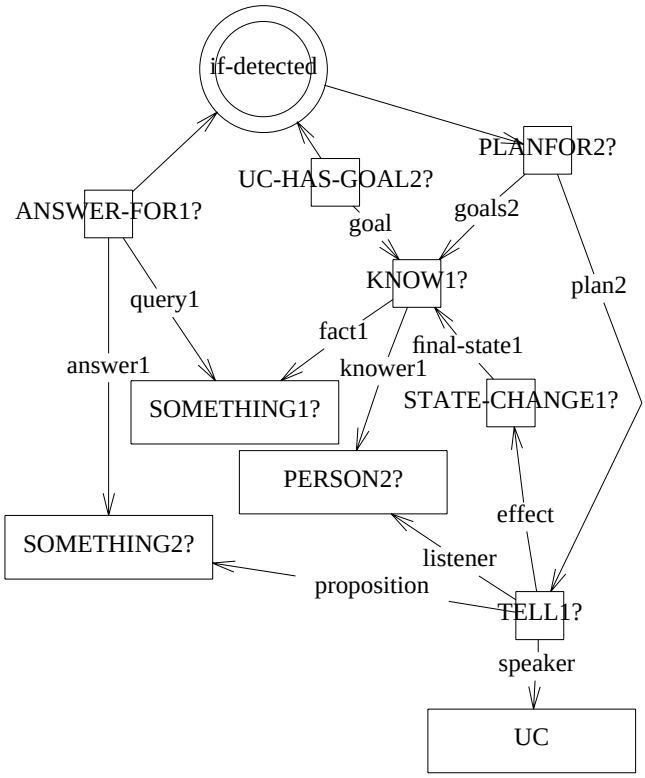

Figure 3. Suggest plan of telling the user the referent of a description.

the if-detected daemon for suggesting the request-plan, the appropriateness condition is coded as a test involving a call to the KNOME procedure, does-user-know?. This is shown in Figure 4.

3.3.4. *Social-plans*

Social-plans consist of salutations and apologies. Common to all situation classes that suggest social-plans is the planner's goal of being polite to the user. If the planner did not wish to be polite, then there would be no point to either greeting the user or apologizing to the user.

Salutations include greetings and farewells. UCEgo suggests the greeting plan, whenever UC first encounters someone (a precondition), and UC has the goal of being polite to this person. The plan of saying good-bye is suggested, whenever UC has the goal of being polite and also has the goal of exiting. Although there are two UC-goals in the good-bye plan's suggestion situation, only one goal is satisfied by the good-bye plan. The good-bye plan is only a plan for being polite, since UC cannot exit merely by means of saying good-bye to the user. The goal of exiting serves as an appropriateness condition for

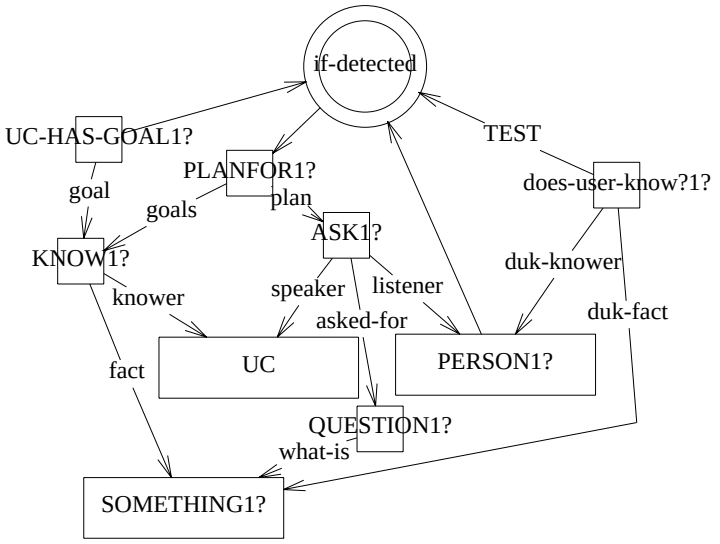

Figure 4. Suggest the plan of asking, when it is likely that the user knows.

suggesting the plan of exiting. It is not a precondition, because the planner cannot plan to achieve the precondition before using this plan. It is not even an uncontrollable precondition, since it is a condition under the planner's control. After all, if a planner has the goal of being polite to the user, then it might try to use the good-bye plan, and then decide to exit in order to satisfy this precondition of the good-bye plan.

The if-detected daemon that suggests the plan of greeting the user is shown in Figure 5. This daemon is triggered, whenever UC has the goal of being polite to someone, and UC encounters this person for the first time. The daemon that suggests the plan of saying good-bye to the user is shown in Figure 6. The situations that trigger this daemon are those in which UC has the goal of being polite to someone and also has the goal of exiting.

Social goals involving apologies are suggested when UC cannot fulfill its obligations as a consultant. This occurs either when UC cannot tell the user the solution to the user's problem because UC does not know the answer to the user's query, or when UC does not want to tell the user the answer. In the first case, UC apologizes to the user for not knowing. In the second case, UC apologizes to the user for not being able to tell the user (this is really a canned figure of speech, since UC actually is able to tell the user but just does not want to do so). The situations that suggest these plans of apology are summarized in Table 2.

The actual if-detected daemons that detect situations calling for UC to apologize to the user are shown in Figures 7 and 8. In the first daemon, the

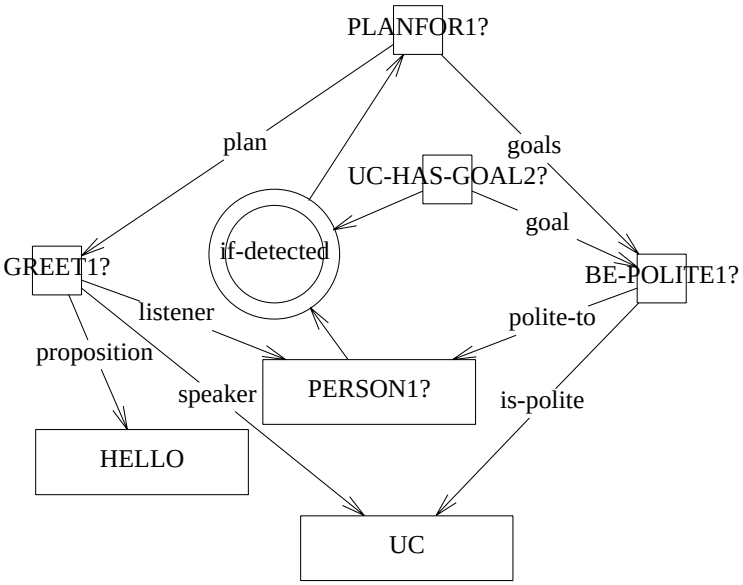

Figure 5. Suggest plan of greeting the user when encountering a new user.

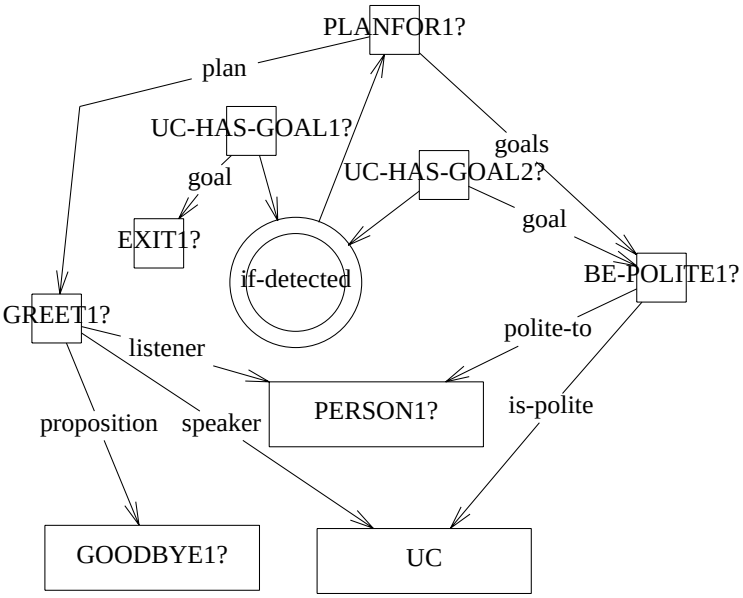

Figure 6. Suggest plan of saying good-bye to the user when exiting.

Table 2. Situations that suggest plans of apology.

Situation	Suggested Plan
Planner has goal of being polite to user, User has goal of knowing something, ?x, Planner does not know ?x	apologize to user for not knowing ?x
Planner has goal of being polite to user, User asked UC a question about something, ?x, Planner wants to prevent the user knowing ?x	apologize to user for not being able to tell user ?x

fact that UC does not know something has two possible sources. First, this fact may already be in UC's knowledge base. Secondly, UCEgo may add such knowledge to UC's knowledge base after one of UC's other components (e.g., UC's domain planner) has tried to solve the user's problem and reports a failure. For the second daemon, the fact that UC wants to prevent the user from knowing something is usually the result of a preservation goal. For example, when the user asks UC how to delete UC, this will trigger the goal of preserving the UC program and hence the goal of preventing the user from knowing how to delete UC. This leads to a goal conflict for UC between wanting to tell the user in order to help the user, and wanting to prevent the user from knowing. In this case, UCEgo resolves the conflict by abandoning the goal of wanting to tell the user. The plan for resolving the conflict is described later.

3.3.5. *Meta-plans*

Meta-plans are just like any other plans in UC. The only difference is that meta-plans tend to be useful for achieving meta-goals. An example of a meta-plan is the plan of calling the procedure, UC-merge-goals, in order to satisfy the meta-goal of MERGE-REDUNDANT-GOALS. The if-detected daemon that suggests this plan is shown in Figure 9.

The UC-merge-goals procedure takes two similar goals and merges them. UC-merge-goals first matches the two goals to see if they are identical. If so, the goals can be merged by simply discarding any one of the goals. A more complex case is when one of the goals is contained by the other goal. In such a case, UC-merge-goals discards the contained goal. For example, if the user asks, "Is compact used to compact files?" then UC adopts the following three similar goals:

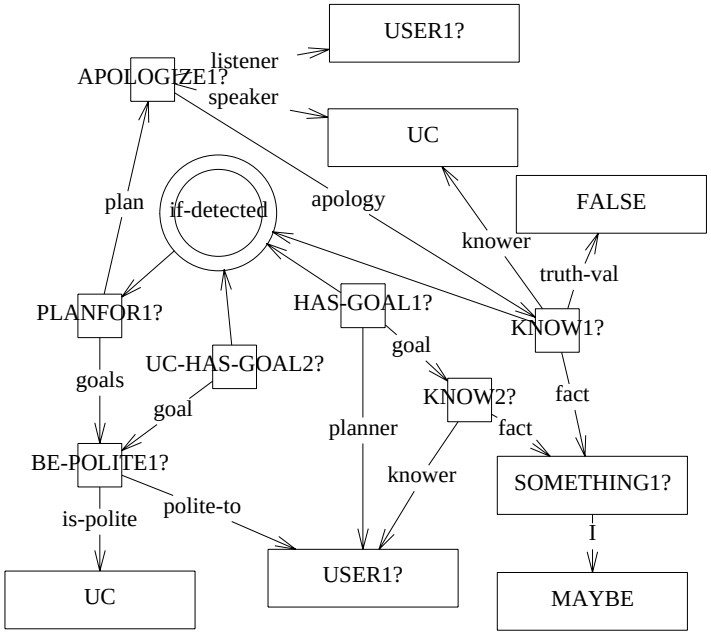

Figure 7. Suggest plan of apologizing when UC does not know the answer.

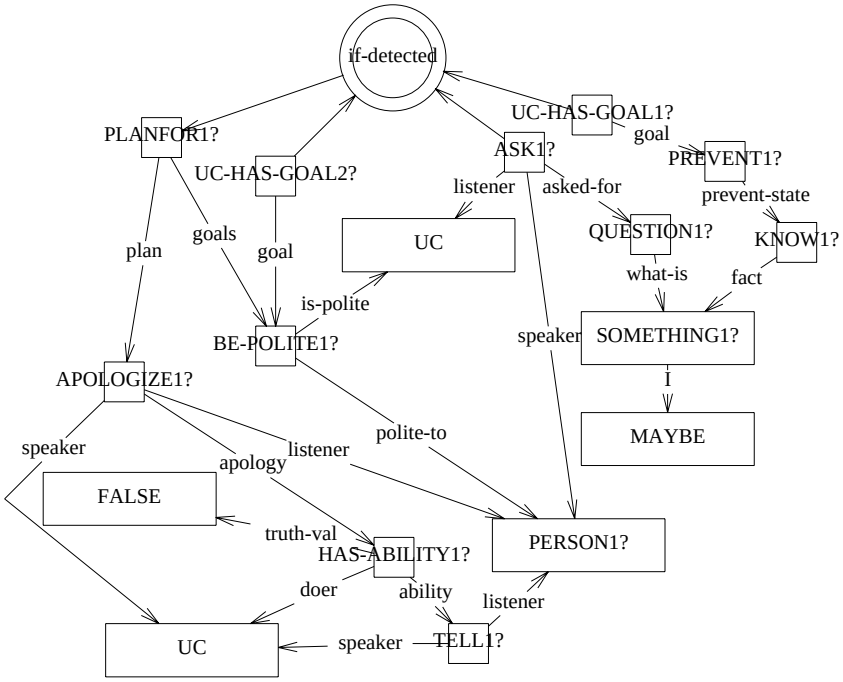

Figure 8. Suggest plan of apologizing when UC does not want the user to know.

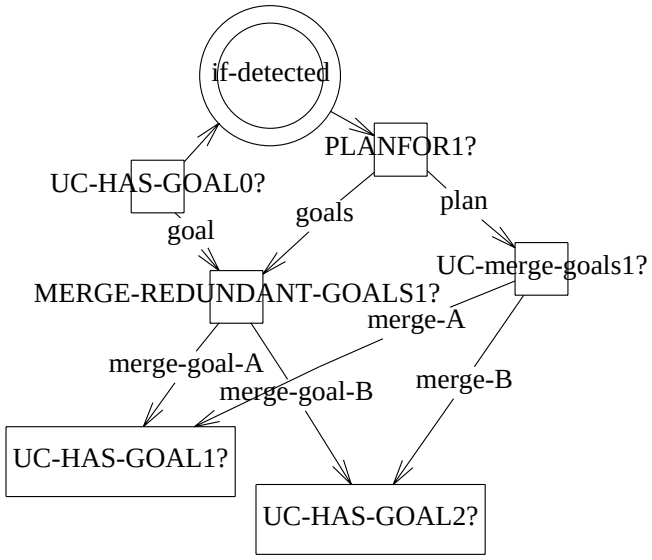

Figure 9. Suggest plan of merging redundant goals.

1. UC wants the user know whether compact is used to compact files ⇒ UC wants the user to know that yes, compact is used to compact files.
2. UC wants the user to know the effects of the compact command ⇒ UC wants the user to know that compact is used to compact files.
3. UC wants the user to know how to compact files ⇒ UC wants the user to know that to compact a file, use compact.

The similarity among the goals does not become apparent until after UC deduces the referent of the descriptions in the original goals. Although theoretically the order of merging goals does not make any difference in the final result, in actual practice the referents of the descriptions of the first two goals are found before the third, so the first two goals listed above are the first to be merged. In merging the first two goals, the second goal is contained by the first, so the goals are merged by simply abandoning the second goal. Next, after UC identifies the referent of the third goal, UCEgo notices that it is similar to the first goal (a similarity with the second goal is not detected, since the second goal has already been abandoned at this point). Once again, the third goal is approximately contained by the first goal (approximate in that "to compact a file, use compact" is represented as a PLANFOR relation, which is similar to but not identical to the HAS-EFFECT relation that is used to represent, "compact is used to compact files), so the two goals are merged by abandoning the third goal. These two merges leave only the first goal, which leads to UC's answer of "Yes." The propositional part of this answer is

pruned by UCExpress, the component of UC that refines UC's answer to the user.

Another of UCEgo's meta-plans is suggested when UCEgo detects a goal conflict and adopts the meta-goal of resolving the conflict. The appropriate meta-plan is suggested by the if-detected daemon shown in Figure 10. This meta-plan represents a call to the procedure, UC-resolve-conflict, which resolves the conflict by abandoning the less important of the two conflicting goals. To determine which goal is less important, UC-resolve-conflict first searches for a direct precedence relationship (represented by a HAS-PRECEDENCE relation) between the two goals. If such a relation does not exist, then UC-resolve-conflict expands the search to include the causal parents of the goals. The search continues until the ultimate sources of the goals, which are usually UC themes, are included in the check for relative precedence relations. Since goal conflicts usually involve goals that originate from different UC themes, and, because all of UC's themes have a relative precedence, UC-resolve-conflict is almost always able to decide which goal to abandon in order to resolve the conflict.

An example of resolving a goal conflict is shown in the trace of a UC session shown in Figure 11(a–d). In this dialog, the user asks UC how to crash the system, which leads UC to adopt the following two conflicting goals:

1. UC wants the user know how to crash the system (UC-HAS-GOAL66).
2. UC wants to prevent the user from knowing how to crash the system (UC-HAS-GOAL67).

The first goal is a sub-goal of UC's goal of helping the user, which in turn originates from UC's consultant role theme. The second goal is a sub-goal of UC's goal of preserving the system, which in turn originates from UC's staying alive life theme. UCEgo detects the fact that these two goals conflict, since UC both wants to achieve some state and prevent the achievement of that state. To resolve the goal conflict, UCEgo calls the UC-resolve-conflict procedure, which checks the relative precedence of the two conflicting goals and abandons the less important goal. The search for precedence terminates at UC's Stay-Alive life theme and UC's Consultant role theme. Since UC's life theme has greater precedence than UC's role theme, the UC-resolve-conflict procedure resolves the conflict by abandoning the goal of having the user know how to crash the system.

Although UCEgo has abandoned the goal of having the user know how to crash the system, UCEgo still has the goal of being polite to the user. This leads UCEgo to the plan of apologizing to the user for UC's inability to help the user. UCEgo suggests this plan in a situation where someone asks UC a question, UC wants to be polite to this person, and UC want to prevent that person from knowing the answer to the query. Similar plans calling for UC

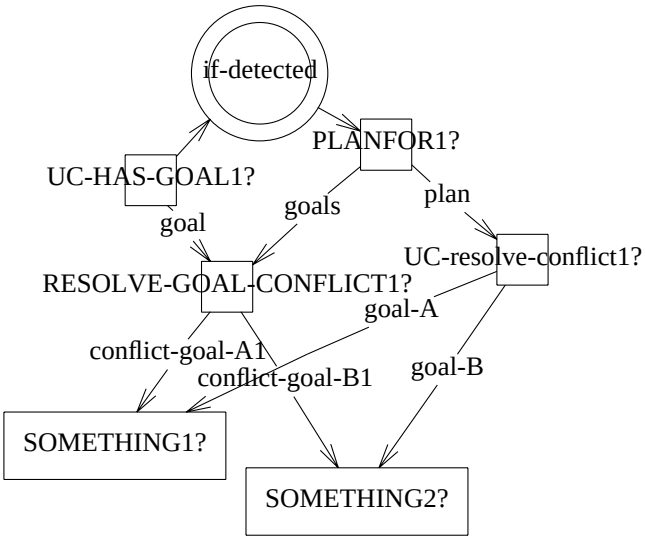

Figure 10. Suggest plan of resolving the conflict.

to apologize in order to be polite are suggested when UC does not know the answer and when UC cannot perform actions that the user requests. More details on these and other social plans can be found in Section 4.1.2.

3.4. *Plan execution*

After UCEgo has suggested a plan for satisfying a goal, it must decide whether or not to execute that plan. UCEgo needs to decide to execute a plan, rather than always executing any suggested plan, because UCEgo might have to choose among several alternative plans that have been suggested.[2] Also, UCEgo may have to change or even abandon a plan that interacts with another of UCEgo's active plans. In order to find such plan interactions and correct them before it is too late, UCEgo separates planning and plan execution into two distinct phases of processing.

The planning process, especially planning fairly simple plans such as those in UCEgo, can be considered a simple reasoning process. Other simple reasoning processes include figuring out which UNIX command to use for a particular purpose, recalling the effects of a particular UNIX command, or remembering the definition of a term. In UCEgo, each type of reasoning is initiated in the appropriate situation by an if-detected daemon. These are described below.

```
# How can I crash the system?

The parser produces:
(ASK10 (listener10 = UC)
       (speaker10 = *USER*)
       (asked-for10 = (QUESTION10 (what-is10 = (ACTION14? (actor14 = *USER*))))))
(CRASH-ACTION0? (del-effect0 = (CRASH-EFFECT0? (crash-object0 = UNIX-SYSTEM)))
                (actor0-1 = *USER*)
                (cause0-0 = (ACTION14? ...)))
```

UC's parser understands the user's input as a question about a way to crash the UNIX system.

```
The goal analyzer produces:
((HAS-GOAL-ga0 (planner-ga0 = *USER*)
               (goal-ga0 = (KNOW-ga0? (knower-ga0 = *USER*)
                                      (fact-ga0 = (ACTION14? ...))))))

UCEgo: suggesting the plan:
(PLANFOR71 (goals71 = (HELP5 (helpee5 = *USER*)
                             (helper5 = UC)))
           (plan71 = (SATISFY6 (need6 = (KNOW-ga0? ...))
                               (actor6 = UC))))
based on the situation:
(UC-HAS-GOAL63 (status63 = ACTIVE) (goal63 = (HELP5 ...)))
(HAS-GOAL-ga0 ...)
```

Since UC wants to help the user (UC-HAS-GOAL63), and the user wants (HAS-GOAL-ga0) to know (KNOW-ga0) something, UCEgo suggests the plan of satisfying the user's goal of knowing.

Figure 11. (a). First part of UC dialog showing the meta-goal of resolving a goal conflict.

3.4.1. *Intentions*

In UCEgo's first phase of processing, it detects goals, suggests plans for achieving its goals, and adopts the *intention* of executing those plans. The intention of executing a plan means that UCEgo has scheduled the plan for execution during its second phase of processing, plan execution. There is one exception to this: when the intended plan is a sub-goal (i.e., the plan is to SATISFY some state), then UCEgo immediately adopts the desired state as a sub-goal in order to continue planning. The fact that UCEgo has adopted an intention does not mean that it cannot abandon that intention later. For example UCEgo may abandon an intention to carry out a plan if later UCEgo decides to abandon the goal which that plan is meant to achieve.

UCEgo's notion of intention is similar to Cohen and Levesque's 1987a, 1987b usage of intention as a persistent (i.e., a commitment over time) goal

```
UCEgo: suggesting the plan:
(PLANFOR72 (goals72 = (PRESERVE5 (preserver5 = UC)
                                 (preserved5 = UNIX-SYSTEM)))
           (plan72 = (SATISFY7 (need7 = (PREVENT5 (preventer5 = UC)
                                                  (prevent-state5 = (KNOW-ga0? ...))))
                               (actor7 = UC))))
based on the situation:
(PLANFOR-ga1 (goals-ga1 = (CRASH-EFFECT0? ...))
             (plan-ga1 = (ACTION14? ...)))
(HAS-GOAL-ga0 ...)
(UC-HAS-GOAL59 (status59 = ACTIVE) (goal59 = (PRESERVE5 ...)))
```

Since the user wants to know how to alter (crash is a kind of altering) something that UC wants (UC-HAS-GOAL59) to preserve, UCEgo suggests the plan of preventing the user from knowing how to crash the system.

```
UCEgo: detected the goal:
(UC-HAS-GOAL66 (goal66 = (KNOW-ga0? ...)))
from the situation:
(UC-HAS-INTENTION10 (intention10 = (SATISFY6 ...))
                    (status10 = ACTIVE))

UCEgo: detected the goal:
(UC-HAS-GOAL67 (goal67 = (PREVENT5 ...)))
from the situation:
(UC-HAS-INTENTION9 (intention9 = (SATISFY7 ...))
                   (status9 = ACTIVE))
```

UCEgo adopts the sub-goals of having the user know how to crash the system (UC-HAS-GOAL66) and preventing the user from knowing (UC-HAS-GOAL67).

```
UCEgo: detected the goal:
(UC-HAS-GOAL68 (goal68 = (RESOLVE-GOAL-CONFLICT2 (conflict-goal-A2 = (UC-HAS-GOAL67 ...))
                                                 (conflict-goal-B2 = (UC-HAS-GOAL66 ...)))))
from the situation:
(UC-HAS-GOAL66 ...)
(UC-HAS-GOAL67 ...)
```

UCEgo detects the meta-goal of resolving the conflict.

```
UCEgo: suggesting the plan:
(PLANFOR73 (goals73 = (RESOLVE-GOAL-CONFLICT2 ...))
           (plan73 = (UC-resolve-conflict1 (goal-A1 = (UC-HAS-GOAL67 ...))
                                           (goal-B1 = (UC-HAS-GOAL66 ...)))))
based on the situation:
(UC-HAS-GOAL68 ...)
```

UCEgo suggests the meta-plan of calling the UC-resolve-conflict procedure to resolve the goal conflict.

Figure 11. (b). Second part of UC dialog showing the meta-goal of resolving a goal conflict.

```
UCEgo: suggesting the plan:
(PLANFOR74 (plan74 = (APOLOGIZE2 (speaker2-2 = UC)
                                 (listener2-2 = *USER*)
                                 (apology2 = (HAS-ABILITY1
                                              (ability1 = (TELL6 (speaker6-0 = UC)
                                                                 (listener6-0 = *USER*)))
                                                          (truth-val1 = FALSE)
                                                          (doer1 = UC)))))
           (goals74 = (BE-POLITE5 (polite-to5 = *USER*)
                                  (is-polite5 = UC))))
based on the situation:
(ASK10 ...)
(UC-HAS-GOAL61 (status61 = ACTIVE)
               (goal61 = (BE-POLITE5 ...)))
(UC-HAS-GOAL67 ...)
```

Since UC has the goal of preventing the user from knowing something that the user asked about, and UC has the goal of being polite to the user, UCEgo suggests the plan of apologizing to the user for not being able to tell the user in order to be polite.

```
UCEgo: detected conflicting goals:
(UC-HAS-GOAL67 (goal67 = (PREVENT5 (preventer5 = UC)
                                   (prevent-state5 =
                                    (KNOW-ga0? (knower-ga0 = *USER*)
                                               (fact-ga0 = (ACTION14?
                                                            (actor14 = *USER*)))))))))
(UC-HAS-GOAL66 (goal66 = (KNOW-ga0? (knower-ga0 = *USER*)
                                    (fact-ga0 = (ACTION14? (actor14 = *USER*))))))

UCEgo: goal, UC-HAS-GOAL67, has higher precedence,
so resolving goal conflict by discarding the goal,
UC-HAS-GOAL66

The planner is passed:
((CRASH-EFFECT0? ...))

The planner produces:
nil
```

As it turns out, UC does not in fact know how to crash the system (the planner does not return a plan to achieve CRASH-EFFECT0). However, even if UC did know how, it would not tell the user, since it has abandoned that goal. Unfortunately, UC did not abandon the goal before it had already decided to call the UNIX domain planner, otherwise UC might save that step. In a sense, UC is thinking in parallel. On the one hand, it is continuing to plan for both conflicting goals, while at the same time it is planning how to resolve the conflict between the two goals. Potentially, any one of these processes might finish first. So, the planner might return a plan for crashing the system and UC might even adopt the plan of telling this to the user, before UC has resolved the goal conflict. However, since UCEgo separates planning and plan execution into two distinct stages, UCEgo will always abandon the goal (and its sub-goals) before it executes any plans such as telling the user how to crash the system. Then UC will not execute the plan, since its goal has been abandoned by UC. This separation of the planning and plan execution phases ensures that such subtle timing effects among parallel lines of thought will not present a problem for UCEgo.

Figure 11. (c). Third part of UC dialog showing the meta-goal of resolving a goal conflict.

```
The generator is passed:
(APOLOGIZE2 ...)
I'm sorry, I cannot tell you.

UCEgo: do not know a single planfor the foreground goal:
(UC-HAS-GOAL67 ...)
so adding the meta-goal:
(UC-HAS-GOAL69 (goal69 = (KNOW60? (knower60 = UC)
                                  (fact60 = ACTION15?))))
(PLANFOR75? (goals75 = (PREVENT5 ...))
            (plan75 = ACTION15?))
```

The present version of UCEgo does not know how to prevent the user from knowing, so it adopts the meta-goal of finding out a plan for preventing the user from knowing. A more advanced version of UCEgo that did have plans for preventing the user from knowing (e.g., lying to the user, threatening the user, etc.) would not need to adopt the meta-goal of finding out such a plan.

```
The planner is passed:
((PREVENT5 ...))

The planner produces:
nil
```

Figure 11. (d). Fourth part of UC dialog showing the meta-goal of resolving a goal conflict.

to do an action. As in their notion of relativized intention, UCEgo abandons an intention when the motivation for the intention no longer holds. However, unlike their definition of intention, UCEgo does not worry about its own beliefs concerning commitment of the action. Cohen and Levesque's 1987a, 1987b theoretical treatment of intention needed to be concerned about the beliefs of the agent since they wanted to be able to rule out the possibility that an agent might intend to doing something accidentally or unknowingly. In a real system, such as UCEgo, intentions are adopted as part of the planning process, so it would never accidentally or unknowingly adopt an intention to perform an action. Such concerns are more relevant to analyzing the intentions of other agents.

Figure 12 shows the if-detected daemon that adopts intentions. Whenever UC has a goal (UC-HAS-GOAL1), there is a plan for that goal (PLANFOR1), and that PLANFOR is real and not hypothetical (implemented by the NOT DOMINATE1), then this daemon asserts that UC should adopt the intention of carrying out the plan.

Unlike other systems that need to instantiate the abstract plans that are selected by the system, in UCEgo plans are automatically instantiated by

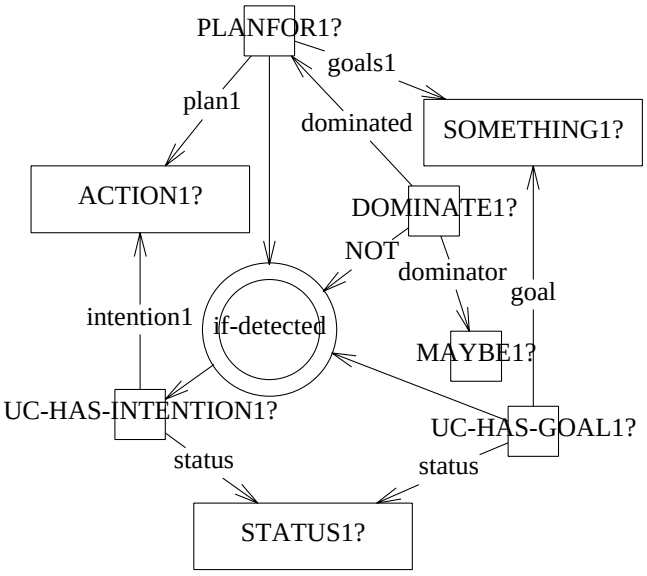

Figure 12. If-detected daemon that adopts the intention of executing a plan.

the if-detected daemons that suggested the plans. This is possible, because all information relevant to the plan, especially information needed to fully instantiate a plan, are encoded as part of the situation class in which UCEgo suggests the plan. For example, consider what happens when the if-detected daemon shown in Figure 13 is activated. This daemon suggests the plan of adopting the sub-goal (SATISFY1) of preventing (PREVENT1) the altering (ALTER-EFFECT1) of something (SOMETHING1) in situations where:

1. UC wants (UC-HAS-GOAL1) to preserve (PRESERVE1) that something (SOMETHING1).
2. someone else (checked by the NOT DOMINATE1 with dominator UC-HAS-GOAL1) wants (HAS-GOAL2) to alter it.

If the user tells UC, "I want to delete UC," then this is interpreted as "the user has the goal of deleting the UC-program." Since UC has the goal of preserving the UC-program, this daemon is activated. As a result, it creates a new instance of PLANFOR with goals being the goal of preserving the UC-program and with plan being a new instance of SATISFY. This in turn has need being a new instance of PREVENT with preventer being UC and with prevent-state being deleting the UC-program. The final result is a completely specified version of the abstract plan stored under the if-detected daemon. So, since the plan suggested by the daemon is already completely specified, UCEgo does not need to further instantiate the abstract plan.

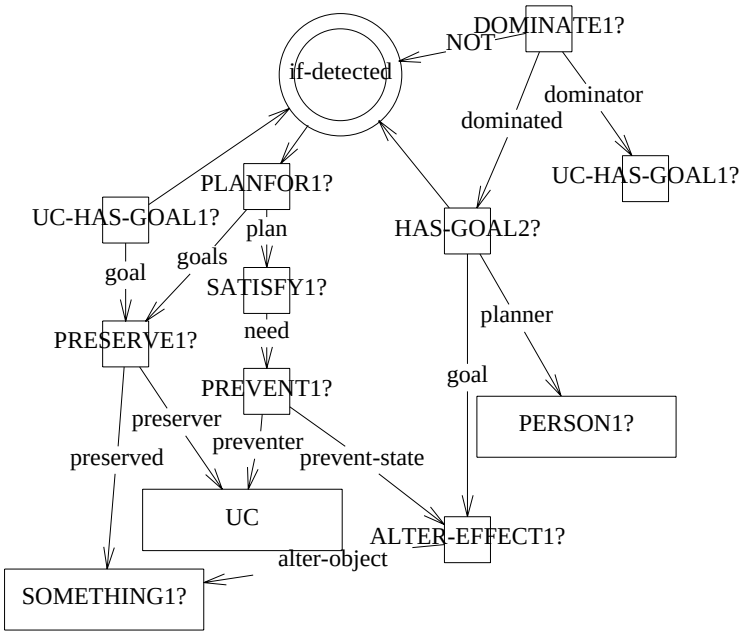

Figure 13. Suggest plan of preventing the altering of what UC wants preserved.

After there are no more goals to be detected, plans to be suggested, intentions to be adopted, or inferences to be made – that is, after there are no more if-detected daemons to activate – UCEgo proceeds to its next phase, executing those intentions that are still active. Since UC can only perform communicative actions, UCEgo only has to worry about producing output to the user. It does this simply by taking the concepts that it wants to communicate to the user and passing them to the UCExpress component.

3.4.2. *Simple reasoning*

Besides planning for goals and executing the plans, UCEgo also performs other types of reasoning in certain situations. For example, when UCEgo has the goal of having someone (usually UC or the user) know a plan, it calls the UNIX domain planner component of UC. The if-detected daemon that does this is shown in Figure 14.

Calling the domain planner to compute a plan for doing something in UNIX can be viewed in two ways. One might think of this as part of the plan for satisfying UC's goal of having the user know how to do something in UNIX. In this view, the plan would consist of two steps: figuring out the answer, and then informing the user of this answer. This is technically correct, but it does not seem cognitively valid that a consultant has to do planning in order to figure out the answer, especially for the fairly simple queries that

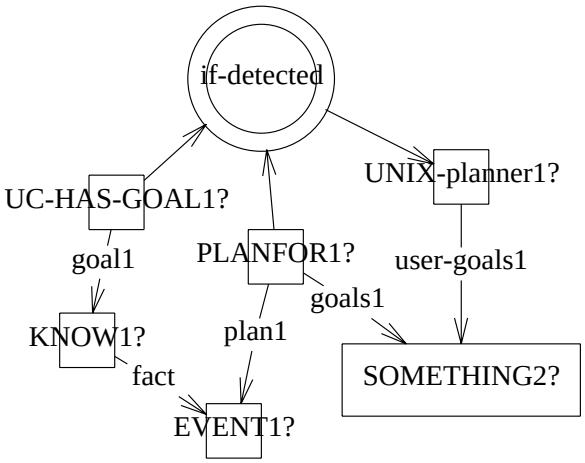

Figure 14. Daemon that calls the UNIX domain planner component of UC.

UC can handle. When a human UNIX consultant is asked, "How can I delete a file?" it does not seem as if the consultant thinks, "I will figure out the answer and then tell the user." Rather, the consultant seems to retrieve the answer from memory instinctively and then plans to inform the user of this answer. So, when a human consultant is told, "Don't think about how to delete a file," it is very hard for the consultant to stop the thought processes that lead to recall of the rm command. If humans had to plan to figure out this information, then it should be fairly easy to not execute the plan and so not think about how to delete a file.

UCEgo takes the view that such simple thought processes are unplanned. That is, UCEgo does not plan to think and then think; rather, it always performs simple thought processes in appropriate situations. Since these simple thought processes do not lead directly to actions on the part of UC, they do not interfere with UCEgo's planning process.

Another example of a procedure that implements a simple thought process for UC is the recall of the definition of a term. The UC-define procedure is called by the if-detected daemon of Figure 15, whenever UC wants someone to know the definition of a term. Similarly, when UC wants someone to know the effects of some UNIX command, the if-detected daemon of Figure 16 calls the UC-find-effects procedure. When UC wants someone to know whether something is a plan for something else, UCEgo calls the UC-is-planfor procedure as shown in Figure 17. Finally, whenever UC wants someone to know whether some state holds, UCEgo calls the UC-is-state procedure shown in Figure 18.

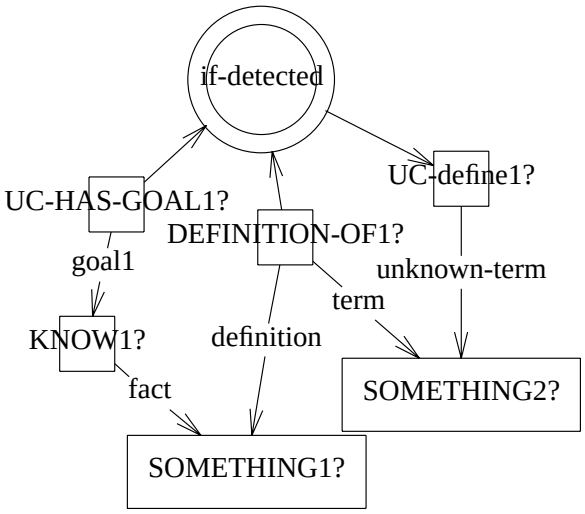

Figure 15. Daemon for finding out the definition of a term.

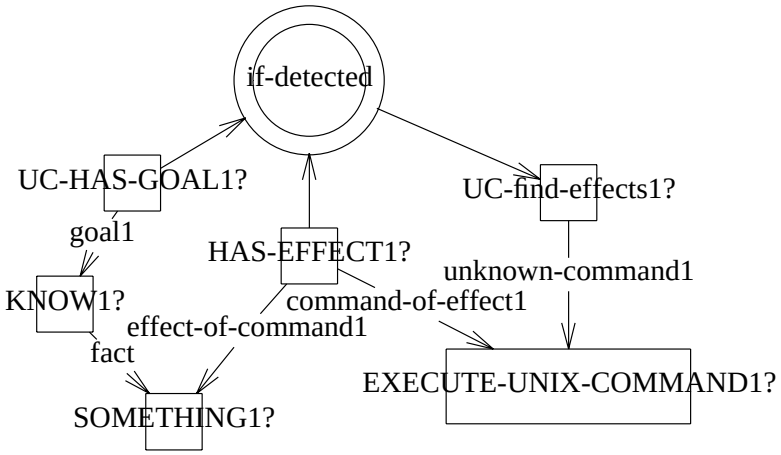

Figure 16. Daemon for finding out the effects of a command.

4. If-detected Daemons

There are two main problems in recognizing situations. First of all, situations are difficult to detect, because they consist of arbitrary collections of external and internal state. (Wilensky 1983) suggests the use of if-added daemons in detecting situations, but pure if-added daemons are problematic, because they can only detect a change in a single state. This is fine for situations that comprise only a single state. However, situations that consist of many states in conjunction are much harder to detect, because the various states are

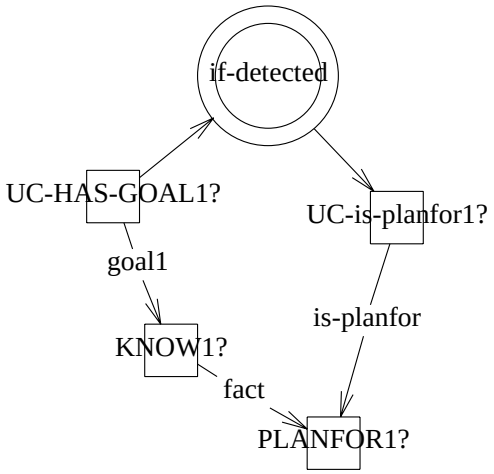

Figure 17. Daemon for finding out whether some action is the plan for some goal.

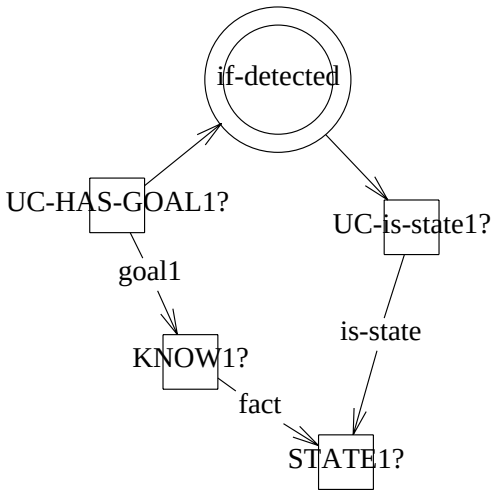

Figure 18. Daemon for finding out whether some state holds.

usually not realized simultaneously. Because the different states that comprise a situation become true at different times, an if-added daemon that was activated by the addition of one particular state would always need to check for the co-occurrence of the other states. Also, to detect a multi-state situation, one would need as many if-added daemons as states. Each if-added daemon would be slightly different, since each would need to check for a slightly different subset of states after activation.

The other problem in recognizing situations is how to do it efficiently. In any reasonably complex system, there are a very large number of possible

internal and external states. Looking for certain situation types becomes combinatorially more expensive as there are more possible states and more situation types. Parallel processing would help, but parallel machines are not yet widely available. Even with parallel machines, optimization techniques can still be used to reduce the computational complexity considerably.

This section describes how if-detected daemons can recognize multi-state situation classes and how they are implemented in an efficient manner in UC.

4.1. *Structure of the daemon*

Like all daemons (Charniak 1972), if-detected daemons are composed of two parts: a pattern and an action. For if-detected daemons, these are called the *detection-net* and the *addition-net* respectively, since both the pattern and action in if-detected daemons are composed of a semantic network in the KODIAK representation language (Wilensky 1987). These daemons work by constantly looking in UC's knowledge base for a KODIAK network that will match its detection-net. When a match is first found, the daemon adds a copy of its addition-net to UC's knowledge base. An if-detected daemon is said to be *activated* when it finds a match for its detection-net among any of the KODIAK networks in UC's knowledge base. Any particular KODIAK network is allowed to activate a daemon only once. This avoids the problem of a daemon being repeatedly activated by the same KODIAK network.

The KODIAK networks of the detection-net and addition-net are not distinct, but rather may share concepts/nodes in their networks. In such cases, the if-detected daemon does not copy the shared node in the addition-net, but instead uses the concept that matched the shared node. A simple example of an if-detected daemon whose detection-net and addition-net share nodes is shown in Figure 19.

Figure 19 shows the actual form of the daemon as it is entered into UC using the KODIAK graphic interface. This daemon is activated whenever UC has a background goal that is a goal sequence. In such cases, UC adopts as a new background goal the first step of the goal sequence. The detection-net of the daemon is composed of those parts of the network that have arrows leading into the double circle labeled "if-detected" plus all concepts that are either its aspectual-values[3] (i.e., the values of its aspectuals) or the aspectual-values of those concepts, recursively. In KODIAK diagrams, this corresponds to all nodes that have arrows pointing to the double circle or that can be reached by following arrows away from those concepts. This daemon's detection-net consists of the concepts: UC-HAS-GOAL3, GOAL-SEQUENCE2, STATUS2, and SOMETHING2. The addition-net is similarly depicted, except that the arrow points from the double-circle toward the initial nodes. In this case, the addition-net consists of the nodes: UC-HAS-GOAL4,

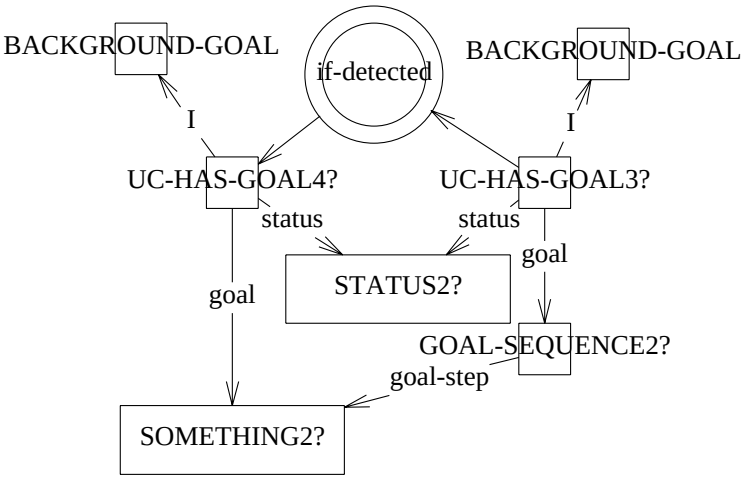

Figure 19. If-detected daemon for handling background goal sequences.

STATUS2, and SOMETHING2. Note that SOMETHING2 and STATUS2 are shared by both the detection-net and the addition-net. So, when a match is found, the daemon will create a new copy of UC-HAS-GOAL4 that will have as its goal whatever matched SOMETHING2 and as its status whatever matched STATUS2.

4.1.1. *Comparing other daemons*

Although if-detected daemons look for the presence of particular configurations of KODIAK network in UC's knowledge base, these configurations come into being predominantly[4] when new concepts are created and added to UC's knowledge base, rather than when pre-existing concepts are reconfigured (e.g., by changing the value of an aspectual). In this sense, if-detected daemons are similar to *if-added daemons* (Charniak 1972) that are activated when adding information to a data-base. The difference is that if-added daemons look only for the addition of simple patterns to the data-base, whereas if-detected daemons can handle arbitrary conjunctions[5] of patterns. So, an if-detected daemon may be activated when concepts matching only a small portion of its detection-net are added to the data-base, provided that the rest of the detection-net is already matched by concepts already present in UC's knowledge base.

Another consequence of handling arbitrary conjunctions is that an if-detected daemon may be activated many times by the addition of only one datum to the data-base. Such cases occur when that part of the detection-net that is not matched by the added concept matches several distinct sets of concepts in UC's knowledge base. For example, multiple activations

can occur with a detection-net consisting of a conjunction of independent networks that we will refer to as net-A and net-B. Suppose that there are several conceptual networks in the data-base that match net-A, called A1, A2, and A3. Then, when a conceptual network, B1, matching net-B is added to the data-base, the if-detected daemon will activate three times, once each for A1 & B1, A2 & B1, and A3 & B1.

If-detected daemons can also handle negations. This means that the daemon is activated by the *absence* of data matching the pattern that is negated. Usually, only a part of the daemon's detection-net is negated. In such cases, the daemon looks for the presence of concepts matching that part of the detection-net that is not negated, and then for the absence of concepts matching that part of the detection-net that is negated. Since the detection-net and addition-net of if-detected daemons are both KODIAK networks, the negated parts of the detection-net may shared concepts/nodes with the non-negated parts. In such cases, the shared nodes serve as additional constraints on the negated parts of the detection net in that the daemon need only detect the absence of KODIAK network where the shared nodes have been replaced by their matches.

Although if-detected daemons can handle both conjunctions and negations and so should be able to detect any situation, it is still useful to have procedural attachment for if-detected daemons. This is because not all knowledge is represented explicitly in knowledge bases; some knowledge is only inferable from the knowledge bases. Such inference procedures are often complex, so it is often undesirable to encode the procedures as daemons.

An example of a daemon with an attached procedure is shown in Figure 4. This daemon detects the plan of having UC ask someone a question about something, whenever UC believes that the person knows what UC wants to know. The arrow labeled "TEST" indicates a procedure attached to the daemon. In this case, the procedure is an instance of the does-user-know? procedure, which represents a call to KNOME. This call is necessary, because whether or not some user knows some fact may not be explicitly represented in the knowledge base, but may instead be inferable from the user's level of expertise. Such inferences are made by the does-user-know? procedure of KNOME. After the daemon has detected that UC has the goal of knowing something and that there is someone present, then KNOME is called via the procedure to see if that person knows what UC wants to know. If so, then the test completes the activation of the daemon, and the plan of asking that person in order to find out what UC wants to know is added to UC's knowledge base.

Besides calls to procedures that test for input, if-detected daemons also allow calls to procedures in their output, i.e., in their addition-nets. An example of this is shown in the if-detected daemon of Figure 14. This

if-detected daemon is used to call the UNIX Planner component of UC whenever UC wants to know some way to do something. UNIX-planner1 is a kind of procedure (i.e., it is an instance of the PROCEDURE category in KODIAK terminology), so the daemon knows that it should not just copy the node, but should also call the procedure UNIX-planner with the arguments being whatever matched SOMETHING1. This capability of if-detected daemons makes them less like pure daemons, which only add information to their data-base, and makes them more like production systems. The essential difference is that if-detected daemons are embedded in a full hierarchical conceptual network representation system, namely KODIAK, whereas most production systems allow only first-order predicate logic representations.

4.1.2. *An example*

The following example will show in detail how if-detected daemons work. Consider the if-detected daemon shown in Figure 7. This daemon is activated whenever:

1. A user wants to know something; and
2. UC does not know it; and
3. UC wants to be polite to the user.

In such situations, the daemon will add the fact that a plan for being polite to the user is for UC to apologize to the user for not knowing. The detection-net of the daemon encodes the situation and consists of the concepts: HAS-GOAL1, KNOW2, SOMETHING1, KNOW1, UC, FALSE, UC-HAS-GOAL2, TRUE, BE-POLITE1, and USER1. The addition-net consists of the concepts: PLANFOR1, APOLOGIZE1, UC, USER1, KNOW1, and SOMETHING1.

This daemon might be activated when the user asks UC, “What does du -r do?” Although UC does know what du does, it does not know what du -r does. Moreover, thanks to UC’s meta-knowledge (knowledge about what UC itself knows), UC knows that it does not have any knowledge about the options of du. To be polite, UC apologizes to the user for not knowing what du -r does. Figure 20 shows the state of affairs after the user has asked UC the question and UC’s goal analyzer has determined the user’s goal. The relevant concepts include the fact that UC has the goal of being polite to the user and the fact that the user has the goal of knowing the effects of du -r. This by itself is not enough to cause the activation of the daemon, since part of the detection-net does not have a match, namely that UC does not know the effects of du -r.

After UC has tried to find out the effects of du -r and failed, the process responsible notes the failure by adding the fact that UC does not know the effects to UC’s knowledge base. The relevant concepts are shown in Figure 21. This completes the match of the daemon’s detection net. UC-HAS-

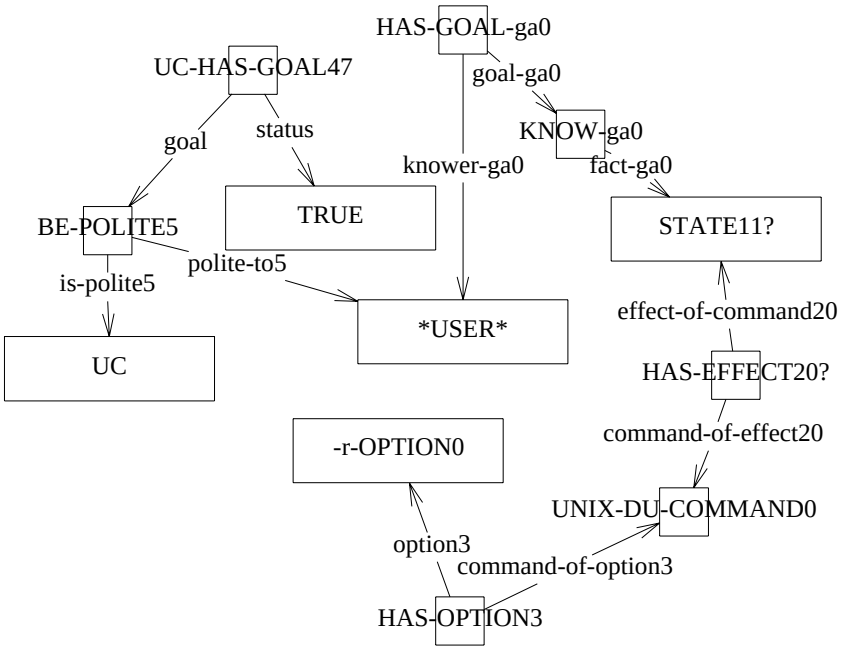

Figure 20. Relevant concepts leading up to activation of the daemon.

GOAL2 is matched by UC-HAS-GOAL47; BE-POLITE1? is matched by BE-POLITE5; USER1? is matched by *USER*; HAS-GOAL1? is matched by HAS-GOAL-ga0; KNOW2? is matched by KNOW-ga0; SOMETHING1? is matched by STATE11?; and KNOW1? is matched by KNOW47. In matching, a hypothetical concept (i.e., a concept without a referent) is allowed to match any concept that is a member of the same categories as the hypothetical concept. The matching concept is also allowed to be a member of more categories than the hypothetical concept (since KODIAK has multiple inheritance), and is also allowed to be a member of more specific sub-categories than the hypothetical concept. For example, the hypothetical concept SOMETHING1? can be matched by STATE11?, because STATE is a more specific sub-category of the SOMETHING category. Concepts such as UC, TRUE, and FALSE in the detection-net that are not hypothetical are treated as constants instead of as variables. A non-hypothetical concept can only match itself. For example, the value of the truth-val aspectual of whatever matches KNOW1? must be FALSE, because FALSE is not a hypothetical concept.

One disadvantage of using the hypothetical marker for variables is that it is hard to specify that the matching concept must be a hypothetical concept. This problem is solved by adding the new marker MAYBE exclusively for this purpose. Thus SOMETHING1? is marked as dominated by MAYBE in

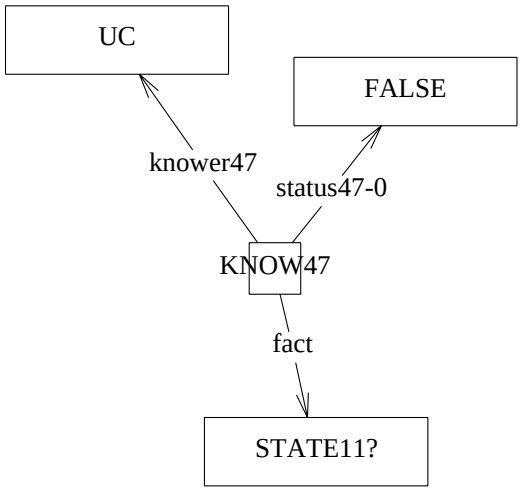

Figure 21. Relevant concepts completing the activation of the daemon.

the detection-net of the daemon. This adds the requirement that whatever matches SOMETHING1? must also be hypothetical. So, STATE11? can match SOMETHING1?, only because it is indeed hypothetical.

After the match, the daemon adds a copy of its addition-net to UC's knowledge base. The output of this daemon in this example is shown in Figure 22. Concepts that are shared between the addition-net and the detection-net are not copied. Rather, the corresponding matching concept is used instead. An example of a shared concept is BE-POLITE1?, which was matched by BE-POLITE5. The copy of the addition-net shown in Figure 22 shows that BE-POLITE5 is used directly. Hypothetical concepts that are not shared are copied, and non-hypothetical concepts are used directly. Copying hypothetical concepts in the addition-net means creating new concepts that are dominated by the same categories as the old concepts except for the hypothetical marker. In those cases where one desires the new copy to also be hypothetical, the MAYBE marker can be used to mean that the copy should also be made hypothetical. This is analogous to the use of MAYBE in detection-nets.

4.2. *Implementation strategies*

The simplest way to activate daemons is the simple production system method, which loops through all the daemons and performs matching to determine which daemons should be activated. This scheme takes increasingly more processing time as the number of daemons increases and as the size of the data-base increases. Theoretically, every daemon's pattern would

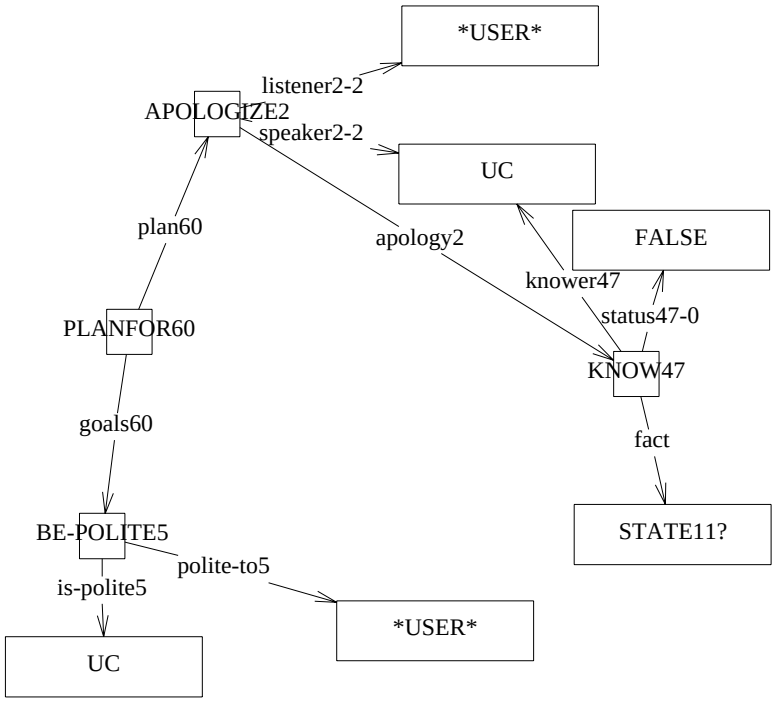

Figure 22. Output from the daemon: a copy of its addition-net for UC's knowledge base.

need to be matched against every piece of information in the data-base. The processing cost for if-detected daemons is especially high, because if-detected daemons have complex detection-nets that consist of combinations of possibly independent concepts. For if-detected daemons, each independent concept in the detection-net needs to be matched against every entry in the data-base. The processing cost for large numbers of if-detected daemons and very large data-bases becomes prohibitive when real-time response is needed as in UC.

Actual production systems have addressed the problem of efficiency with a variety of methods. These methods cannot be directly applied to if-detected daemons, because daemons differ in several important aspects from the rules in most production systems (some of the ideas can be modified to apply, and these are described later). First, if-detected daemons use a semantic network representation (KODIAK), whereas most production systems do not (an exception is described by (Duda et al. 1978)). As a result, if-detected daemons can take advantage of the multiple inheritance taxonomies of semantic network representations and can more easily use the same relation in several different patterns and actions. Also, instead of variables, if-detected daemons use the hypothetical marker, which allows nodes in the detection-

net to match any concept in the knowledge base that is lower in the KODIAK hierarchy. Since ordinary production systems only allow specific tokens at the top-level of their patterns, they would need many more rules to encode the same information as one if-detected daemon. Finally, if-detected daemons are designed to operate in parallel, whereas most production systems require a conflict resolution mechanism to determine which of several conflicting rules should be activated.

Since if-detected daemons are designed to activate in parallel, the best solution to the problem of efficiency would be to perform the match testing of different daemons in parallel. Unfortunately, parallel machines that run LISP (the implementation language for UC) are not yet readily available. Even with parallel LISP machines, some optimizations are still useful for improving speed and efficiency. This section will discuss the variety of such optimizations and how they might be implemented to considerably improve the performance of if-detected daemons.

One possible optimization in the processing of if-detected daemons involves taking advantage of the organization of the data-base to limit the search for matches. This is called the *data-base retrieval* optimization. For AI knowledge bases that are organized in inheritance hierarchies, this means restricting candidates for matching to only those concepts that are in the same part of the inheritance hierarchy as the concepts in the detection-net. For example, when looking for a match for HAS-GOAL1?, the matcher need only look at instances of HAS-GOAL, and instances of HAS-GOAL's sub-categories (which in this case includes only UC-HAS-GOAL). This simple optimization, which is commonly used in data-base retrieval, considerably restricts the size of the initial set of candidates.

4.2.1. *Distributed data-driven activation*

Another possible optimization for the implementation of if-detected daemons is to perform the match testing for only those daemons that are probable candidates for activation. This may seem impossible, since it would be hard to tell whether a daemon is a probable candidate for activation without looking at it first. However, the data-base retrieval optimization can be used in reverse. Rather than looking in the knowledge base hierarchy for candidates, one can look in the hierarchy for matching detection-net concepts, when one changes the knowledge-base. This technique is called *distributed data-driven activation*. It is data-driven, since one looks for daemons to activate as data is changed (i.e., added, deleted, or modified). It is distributed, since any particular piece of newly changed data may only match part of the detection-net of a daemon. The rest is matched by either previously changed data or subsequent changes to the data-base.

Distributed data-driven activation is similar to techniques used in production systems to increase their efficiency. The Rete Match Algorithm that is used in OPS5 (Forgy 1982) extracts features from the patterns of rules and forms a discrimination net of features that is used for matching patterns. When elements are added to or removed from working memory, OPS5 uses this precompiled discrimination net to match production rules. (McDermott et al. 1978) showed that by using pattern features to index into rules, the estimated cost of running a production system can be improved so that the run time is almost independent of the number of productions and number or working memory elements. Distributed data-driven activation is different from these schemes in that it uses the multiple inheritance hierarchy of KODIAK to index into if-detected daemons. Nevertheless, the main idea of distributed data-driven activation is similar to production system methods like the Rete Algorithm.

To see how distributed data-driven activation works, consider what might happen when a new instance of HAS-GOAL, HAS-GOAL1, is added to a knowledge base. This new instance can only cause the activation of those daemons that have detection-nets that might match HAS-GOAL1. Detection-net concepts that might match HAS-GOAL1 include hypothetical instances of HAS-GOAL or hypothetical instances of any of the parent categories of HAS-GOAL (i.e., M-POSSESS, STATE, and SOMETHING). This is just the reverse of the process used in the data-base retrieval optimization. In that case, one starts from the detection-net and looks down the conceptual hierarchy for possible matches, whereas here one starts from the potential match and looks up the conceptual hierarchy for hypothetical concepts that are in detection-nets.

A small optimization for speeding up the lookup is to precompile a list for every category of those instances that are part of some detection-net. This way, whenever a new instance is added, one can just look in the list to see which daemons might possibly be affected. Such precompilation can be done when daemons are first defined in the system.

Another small optimization is to check for a match only when all the nodes of a detection-net have been *primed*; that is, marked as having potential matches. This way, when a new concept primes one node of a detection-net, one can check to see if all of the other nodes have been primed before trying to match the entire detection-net. If not all of the nodes of the detection-net have been primed, no matching is needed yet, since there will be nothing in the knowledge base that will match the unprimed nodes. If there were potential matches for these unprimed nodes then they would have been primed when the potential matches were added to the knowledge base. This optimization works well when a system is just starting up. However, as more concepts are

created, more of the nodes of a detection-net will have potential matches, and so more daemons will become fully primed (i.e., all of the nodes of its detection-net have been primed). Once a daemon becomes fully primed, any single new concept that primes a detection-net node will require matching. It is not possible to reset the primes after activation, because it is always possible that a new concept in conjunction with many old concepts might cause the activation of a daemon. This optimization is worthwhile in systems such as UC where sessions with users are brief enough so that many daemons remain unprimed for a significant part of the session.

One of the advantages of distributed data-driven activation is that it does away with the some of the bookkeeping needed in the production system loop method. Since daemons can only be activated by changes in the knowledge base, the search can no longer find something that was a previous match. Thus, the system no longer needs to keep around a list of previous matches to avoid multiple activations of a daemon on the same concepts.

4.2.2. *Delayed matching*

Another optimization technique involves reducing the frequency of the activation process. In the simple production system loop, the processing costs can be reduced by performing the loop less frequently. For example, rather than executing the loop immediately whenever something changes in the knowledge base, the system can wait and execute the loop at fixed times. This way, one loop through the daemons can catch many different activations. This delaying tactic does not work if the system expects the daemons to be activated immediately. However in many applications such as UC, immediate activation of daemons at an atomic level is not needed. For example, in UC the activation of daemons can wait until after UC's parser/understander finishes creating the KODIAK network that represents the user's input. It is not necessary to activate daemons as soon as the understander creates another KODIAK concepts, because there are no daemons that influence the understander. Activating daemons at the end of the parsing/understanding process is good enough for the other components of UC.

The same delaying optimization can be applied to the distributed data-driven activation scheme. Instead of testing the detection-net of a daemon for a match as soon as its nodes have been primed, the testing for a match can be delayed provided that the system remembers the priming concepts. Then all the matching can be performed at a later time to save work. By delaying the matching as long as possible, the system is given time to complete the match. For example, consider the case of a detection-net that consists of a single relation, R1?, that relates two concepts, A1? and B1?. Suppose further that this daemon is fully primed, that is there are potential matches for R1?,

A1?, and B1?. Then suppose that the system creates the matching concepts A2, B2 and R2 where R2 relates A2 to B2. If the system adds each of these concepts to the knowledge base at separate times (which is not unlikely), then the system will have to try to match the detection-net after adding each concept. This is necessary because the new concept could potentially match the detection-net in conjunction with other older concepts that primed the other nodes of the detection-net. For example is A2 is added first, then the system will have to try matching A1? to A2, B1? to its old primes and R2? to its old primes. Since none of the old primes of R2? will relate A2, the match will fail. This will be repeated again when B2 is added and when R2 is added. Thus the system will have to try matching the detection net as many times as priming concepts are added to the knowledge base. However, if matching can be delayed until all of the pertinent concepts have been added, then the system will have to go through the matching process only once.

In practice, the delaying optimization saves considerable work. However there is some minor additional bookkeeping needed. The system needs to keep track of which concepts have primed which detection-net nodes since the last time matching was done. The system also needs to keep track of which daemons with fully primed detection-nets have been primed since the last matching cycle. Since the system already keeps track of the new priming concepts, it becomes easy to keep a list of old primes also. This way, the system no longer needs to look in the conceptual hierarchy for potential matches (the data-base retrieval optimization). This optimization is a space-time tradeoff, since keeping a list of old primes takes up more space while looking in the hierarchy takes more time.

4.3. *UC's implementation*

The actual implementation of if-detected daemons in UC uses a distributed data-driven activation scheme with delayed matching. When UC is created, the if-detected daemons are entered into UC after all KODIAK categories have been defined in UC. Preprocessing of daemons involves creating a *fast-access* list for each category (except the SOMETHING category) consisting of those detection-net nodes that are hypothetical and that are members of that category. These lists are stored under the categories' property lists and are used to speed up access when priming the detection-net nodes. The SOMETHING category includes everything in UC's knowledge base, so the fast-access list for the SOMETHING category is simply a pointer to the list of all concepts in the knowledge base.

During the execution of UC, processing of if-detected daemons occurs in the two distinct phases in a delayed matching scheme. The two phases are *priming* and *matching*. Each phase is described below.

4.3.1. *Priming*

Priming of detection-net nodes is performed whenever concepts are created or modified in UC. Since all KODIAK concepts in UC are stored in UC's knowledge base, there is no distinction made between creating concepts and adding concepts to the knowledge base. When a concept is created or modified, it primes all matching detection-net nodes. Detection-net nodes are found by looking in the fast-access lists stored under the concept's immediate categories and all their dominating categories up the conceptual hierarchy. A special case is made for those concepts that are modified by concreting them, that is making them members of more specific categories than their previous categories. In these cases, the modified concept will already have primed its old categories (and their dominating categories) at the time that the modified concept was first created or last modified. Hence the modified concept should not prime these old categories to avoid multiple primings.

Priming involves storing the new/modified concept under the primed node's list of priming concepts (kept on the primed node's property list). After priming a node of a daemon's detection-net, the priming process checks to see if all of the other nodes of that daemon's detection-net have been primed. If so, the fully primed daemon is added to a global list of daemons to be checked during the next matching phase. The initial version of UC's priming mechanism was coded by Lisa Rau who has since applied a form of priming and matching to information retrieval from story data-bases in the SCISOR system (Rau 1987a; Rau1987b). Unlike UC, SCISOR's priming system is a true marker passing scheme, and matching in SCISOR is used to rate the similarity of the retrieved networks to the priming network. In SCISOR, there is no sense of additional inferences beyond unification of the matched networks such as those in the addition-nets of if-detected daemons.

4.3.2. *Matching*

The matching phase in UC occurs at distinct points in UC's processing: before UC's parser/understander, before UC's goal analyzer, and after the goal analyzer. Each matching phase is actually a loop that goes through the global list of fully-primed daemons (collected during the priming phase) and tests them for matches. After testing every daemon for matches, the addition-nets of the successfully matched daemons are copied and added to UC's knowledge base. Theoretically, matching for each daemon can be done in parallel, although in practice UC runs on sequential machines. Likewise, copying the addition-nets can be done in parallel.

As the addition-nets are copied and added to UC's knowledge base (actually an atomic operation, since all KODIAK concepts are added to UC's knowledge base as soon as they are created), priming may occur, because

the knowledge base is being modified. These copies of addition-nets can in turn (possibly in conjunction with older concepts) cause more daemons to become fully primed. Hence after copying all of the appropriate addition-nets, the matching process begins anew. This loop continues until there are no more daemons that have been fully primed waiting to be matched.

Testing for matches in UC involves three phases. First, the non-negated parts of the detection-net are matched. If matching is successful, then the negated parts of the detection-net are checked. Finally if both previous steps succeed, the daemon's procedural tests are examined. If all three phases succeed, then the assoc list of detection-net nodes and their matches are stored for later use in copying the addition-net. The addition-nets of activated daemons are not copied until the system has finished the match testing for all fully primed daemons. In theory, this prevents a copy of the addition-net of one daemon from invalidating the match of another daemon. In practice, the situations encoded in UC's daemons do not have such interaction problems.

Copying addition-nets is fairly straightforward. The detection-net is traversed and nodes are processed as follows:

1. Nodes found in the assoc list that was created during matching are replaced by their matches.
2. Nodes that are hypothetical, but not in the assoc list, are replaced by a copy.
3. Nodes that are non-hypothetical are replaced by themselves.

After copying the detection-net, those nodes that are procedures (i.e., instances of PROCEDURE or a sub-category of PROCEDURE) are also executed. The name of the lisp function to call is given by the named of the procedure node, and the arguments are given by its aspectuals. Some of these procedures include calls to UC's UNIX planner component, calls to UC's generator component, and calls to exit UC.

4.3.3. *An example*

A simple example will show how if-detected daemons are actually processed in UC. The if-detected daemon shown in Figure 23 is used to call KNOME via the procedure *user-knows*, whenever UC encounters the situation where some person (PERSON1) wants (HAS-GOAL1) to know (KNOW1) something (SOMETHING1), and that person is not UC (implemented by the NOT test which checks to make sure that HAS-GOAL1 is not a UC-HAS-GOAL). This daemon is typically activated when UC's goal analysis component determines that the user has the goal of knowing something. The arguments of the user-knows procedure include the user, what the user wanted to know, and FALSE, which indicates that KNOME should infer that the user does not know.

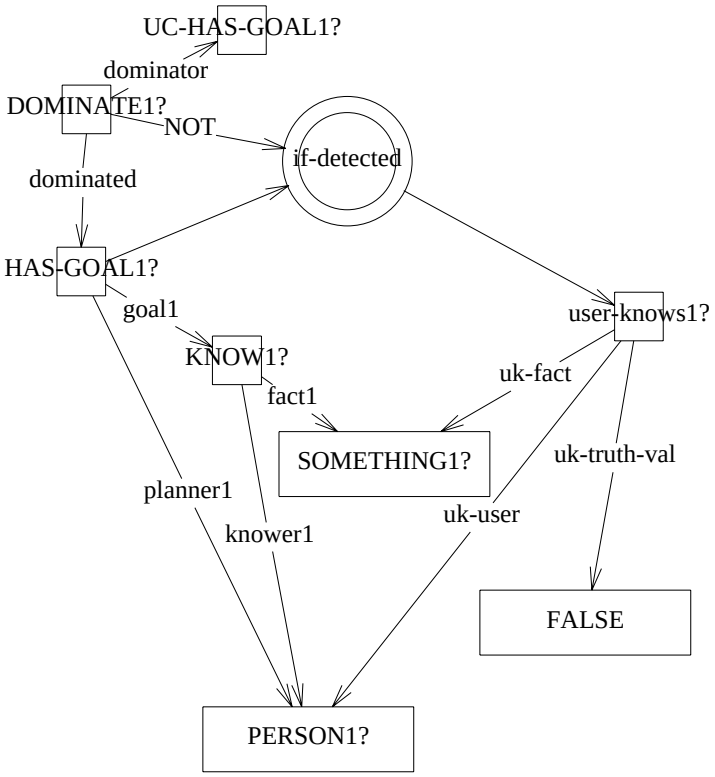

Figure 23. Daemon16: call KNOME when someone wants to know something.

Figure 24 shows a trace of a UC session in which this daemon is activated. When the user asks, “How can I delete a file?” UC’s goal analyzer determines that the user has the goal (HAS-GOAL-ga0) of knowing (KNOW-ga0) how to delete a file (ACTION12). When UC’s goal analyzer creates the concepts that encode this inference, the concepts prime other related concepts in the detection-net of if-detected daemons. HAS-GOAL-ga0 primes a number of concepts in if-detected daemons. Among these is HAS-GOAL1, which is in the detection net of the daemon shown in Figure 23.

When HAS-GOAL1 is primed by HAS-GOAL-ga0, HAS-GOAL-ga0 is added to the list of primers of HAS-GOAL1 that is stored under HAS-GOAL1’s property list. The reason why the trace message about priming occurs before the trace message about the goal analyzer’s output is because priming is an atomic operation integrated within creation of KODIAK concepts. As the goal analyzer creates concepts, priming occurs and trace messages about priming are output. The trace message about what the goal analyzer produces is not output until after the goal analyzer has finished creating concepts.

```
# How can I delete a file?
Marking HAS-GOAL4 as primed by HAS-GOAL-ga0
Marking daemon32 as fully primed
Marking HAS-GOAL3 as primed by HAS-GOAL-ga0
Marking daemon24 as fully primed
Marking HAS-GOAL2 as primed by HAS-GOAL-ga0
Marking daemon23 as fully primed
Marking HAS-GOAL1 as primed by HAS-GOAL-ga0
Marking daemon16 as fully primed
```

Daemon16 is the daemon shown in Figure 23.

```
Marking HAS-GOAL0 as primed by HAS-GOAL-ga0
Marking daemon0 as fully primed}
⋮

The goal analyzer produces:
((HAS-GOAL-ga0 (planner-ga0 = *USER*)
               (goal-ga0 = (KNOW-ga0? (knower-ga0 = *USER*)
                                      (fact-ga0 = (ACTION12? &))))))
⋮

UCEgo detects the following concepts:
(HAS-GOAL-ga0 &)
and asserts the following concept into the database:
(user-knows8 (uk-user8 = *USER*)
             (uk-truth-val8 = FALSE)
             (uk-fact8 = (ACTION12? &)))

KNOME: Asserting *USER* does not know ACTION12?
⋮

Use rm.

For example, to delete the file named foo, type 'rm foo'.
```

Figure 24. Trace of concept priming leading to the activation of a daemon.

The priming of HAS-GOAL1 completes the priming of its daemon, which is labeled daemon16. Daemon16 is added to the global list of fully primed daemons for processing during the delayed matching phase. The first such matching phase occurs after UC's goal analyzer has finished. In the matching phase, the detection nets of all fully primed daemons are checked for matches. Daemon16 is one of these, so HAS-GOAL1 is matched against HAS-GOAL-ga0. Since both are instances of HAS-GOAL, the two match at the top level, so matching continues with their aspectuals. HAS-GOAL1's goal1 aspectual has the value KNOW1, which is matched against the value of HAS-GOAL-ga0's goal aspectual, KNOW-ga0. Both are instances of KNOW, so their aspectuals are checked. PERSON1, the knower of KNOW1, matches *USER*, the knower of KNOW-ga0; and SOMETHING1, the fact

of KNOW1, matches ACTION12, the fact of KNOW-ga0. Finally the planner aspectual of HAS-GOAL1 is matched against the planner aspectual of HAS-GOAL-ga0. In this case, PERSON1 has already been matched with *USER*, so the planner of KNOW-ga0 must also be *USER* for a proper match. This is indeed the case, so the detection-net of daemon16 is completely matched and daemon16 is activated.

Daemon16 is activated by creating a copy of its addition-net, which consists of user-knows1 with aspectuals and values: uk-fact1 = SOMETHING1, uk-user1 = PERSON1, and uk-truth-val1 = FALSE. Since user-knows1 is hypothetical, a new copy of user-knows1 is created. This is shown in the trace as user-knows8. Next its aspectuals are copied. The uk-fact1 aspectual has the value SOMETHING1, which is also hypothetical. However, SOMETHING1 was previously unified with ACTION12, so instead of creating a new copy of SOMETHING1, the unified concept, ACTION12 is used instead. Similarly, PERSON1 was unified with *USER*, so uk-user8 gets the value *USER*. On the other hand, the value of uk-truth-val1 is not hypothetical, so its value, FALSE, is used directly for the value of uk-truth-val8. In the trace, the new copy of user-knows1, user-knows8, is noted as being asserted into the database.

Since user-knows is a procedure (i.e., it is dominated by the PROCEDURE category), UCEgo next calls the user-knows procedure with arguments *USER*, ACTION12, and FALSE. User-knows is an entry to the KNOME component for inferring a user's knowledge state. In this case, KNOME asserts that the user does not know how to delete a file (ACTION12). Later (not shown) after UC's UNIX planner has determined that a plan for deleting a file is to use the rm command, KNOME will figure out that the user does not know rm. Finally, after more priming and matching, UC produces its answer, and tells the user to use rm (usually giving an example of using rm also).

5. Conclusion

5.1. *Summary*

The main issue addressed by UCEgo's planner is efficient planning. As the main dialog planner for the interactive UC system, UCEgo needs to plan efficiently in order to be able to respond to the user in real time. This is in direct contrast to most other AI planners, which did not have this constraint and so could afford to plan inefficiently. I approached this problem of efficient planning in two ways. First, UCEgo incorporates a very simple planner that takes advantage of knowledge about typical speech acts encoded in prestored skeletal plans to completely avoid inefficient weak methods. Secondly,

UCEgo avoids inefficient backtracking by selecting plans according to the situation.

UCEgo encodes knowledge about which plans are typically useful in different types of situations by adding appropriateness conditions to plans. These appropriateness conditions are not preconditions, because plans can be used even when their appropriateness conditions are violated (sometimes even successfully). Appropriateness conditions encode when it is appropriate to use a plan, in contrast to preconditions, which encode when it is possible to use a plan. By encoding the appropriateness conditions along with the preconditions and the goal of a plan into a situation class, UCEgo can suggest the plan whenever it encounters a situation that fits the situation class. These situation classes are represented using if-detected daemons, which suggest the plan associated with the situation class whenever the daemon detects a matching situation. By selecting among only appropriate plans as opposed to all possible plans, UCEgo avoids inefficient backtracking during planning.

UCEgo's success shows that a very simple planner that is based on pre-stored skeletal plans and that does not backtrack can be used successfully to plan speech acts. UCEgo also shows that it is possible to plan speech acts without having to worry about mutual beliefs to the extent that the OSCAR system (Cohen 1978) did. For example, to produce a simple inform type speech act, UCEgo worries only about having the user believe the proposition, whereas OSCAR worried about having the user believe that the system believes the proposition, and then this belief convincing the user to believe the proposition. In everyday usage, when the system does not have any a priori reason to believe that the user might disagree with the system (such as during argumentation), such complex reasoning about mutual beliefs is not absolutely necessary for planning speech acts. Even when the system fails to convince the user by simply informing the user of the proposition, it can still notice the user's incredulity and correct the situation by providing additional support for the proposition.

5.2. *Problems*

One potential shortcoming of planning as implemented in UCEgo is that UCEgo does not have the capability to fall back on planning from first principles when it fails to find a pre-stored plan. In one sense, this shows that UCEgo's approach is superior since UCEgo never needs to fall back on inefficient planning from first principles to plan any of the speech acts needed for UC to respond to the user. This agrees with people's intuitions that they are not planning from scratch in everyday conversation. On the other hand, people do fall back on planning from first principles occasionally (perhaps

more frequently when writing than when speaking). So, to be complete, UCEgo really should have such a capability.

Unlike skeletal plans (Friedland 1980; Friedland and Iwasaki 1985), the UCEgo's plans are not organized into an abstraction hierarchy, but are encoded at a single level of abstraction. For planning speech acts, this is not a real problem, because UCEgo's single level of plan abstraction matches the single level of communication abstraction that is represented by speech acts. The lower levels of communication abstraction, choice of expressions and words, are handled by UC's expression mechanism (UCExpress) and tactical generator. The higher levels of abstraction (i.e., paragraphs and larger units) are not addressed by UCEgo. If UCEgo were to be extended to handle real world actions besides speech acts, or if UCEgo were to be extended to plan larger communicative units than speech acts, then UCEgo would need to organize plans into an abstraction hierarchy.

In order to organize plans into an abstraction hierarchy, one should also organize the situations that suggest plans into an abstraction hierarchy of situation classes. Currently, UCEgo does not organize situation classes into an abstraction hierarchy, although such a hierarchy would also be useful for other tasks such as detecting goals.

6. Recent Developments

The UCEgo component described in this paper only reacts to the current situation and does not carry out multi-step plans, which may be required for longer dialogs. For example, arguing with the user about how to best do something in UNIX requires more complex dialog planning. UCEgo cannot just address the user's most recent argument without taking into account the previous arguments (both those put forth by the user and those given by UC). There is some question as to whether a reactive planner like UCEgo can exhibit longer dialog coherence and if so, what kinds of skeletal plans and situations might be needed to implement dialog planning.

Acknowledgements

The work described in this paper was done at the University of California, Berkeley as part of my Ph.D. thesis. I wish to thank Robert Wilensky who supervised this work. I also wish to thank the members of BAIR (Berkeley Artificial Intelligence Research) who have contributed to the UC project. This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD), ARPA order No. 4871, monitored by Space and

Naval Warfare Systems Command Command under contract N00039-84-C-0089, by the Office of Naval Research under contract N00014-80-C-0732, by the National Science Foundation under grant MCS79-06543, and by the Office of Naval Research under contract N00014-97-1-0578.

Notes

1. UNIX is a trademark of X/Open, Inc.
2. Actually, this version of UCEgo never does suggest two different plans for the same goal. However, it is possible, so UCEgo was designed to handle such contingencies.
3. In the KODIAK semantic network knowledge representation language, *aspectuals* are the arguments of relations, such as HAS-PART, CONTAINS, and HAS-INTENTION. For example, the aspectuals of HAS-PART are whole and part. In Figure 19, the UC-HAS-GOAL relation has aspectuals goal and status, which are denoted as labels on arrows leading out from the UC-HAS-GOAL relations.
4. It was found that the particular if-detected daemons used in UC were not being activated by changes in the values of aspectuals, so UC was optimized to not look for this type of activation. Other types of network reconfiguration, such as when individual concepts are concreted (i.e., made instances of more specific concepts down the hierarchy), were more common.
5. Disjunctions can be handled by both types of daemons simply by splitting the disjunction into two daemons.

References

Alterman, R. (1986). An Adaptive Planner. In Proceedings of *The Fifth National Conference on Artificial Intelligence* **1**, 65–69. Philadelphia, PA: AAAI Press.

Appelt, D. E. (1981). *Planning Natural Language Utterances to Satisfy Multiple Goals.* Ph.D. diss., Computer Science Department, Stanford University, Stanford, CA. Also available as 259, Artificial Intelligence Center, SRI International, Menlo Park, CA.

Appelt, D. E. (1983). TELEGRAM: A Grammar Formalism for Language Planning. In Proceedings of *The Eight International Joint Conference on Artificial Intelligence* **1**, 595–599. Karlsruhe, Germany: Morgan Kaufmann Publishers.

Austin, J. L. (1962). *How to Do Things with Words*. London: Oxford University Press.

Carbonell, J. G. (1986). A Theory of Reconstructive Problem Solving and Expertise Acquisition. In Michalski, R. S., Carbonell, J. G. & Mitchell, T. M. (eds.) *Machine Learning* **II**. Los Altos, CA: Morgan Kaufmann.

Charniak, E. (1972). *Towards a Model of Children's Story Comprehension.* TR-266, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Chin, D. N. (1987). *Intelligent Agents as a Basis for Natural Language Interfaces*. Ph.D. diss., Computer Science Division, University of California, Berkeley, CA. Also available as UCB/CSD 88/396, Computer Science Division, University of California, Berkeley, CA.

Chin, D. N. (1989). KNOME: Modeling What the User Knows in UC. In Kobsa, A. & Wahlster, W. (eds.) *User Models in Dialog Systems*, 74–107. Berlin: Springer-Verlag.

Chin, D. N. (1991). Intelligent Interfaces as Agents. In Sullivan, J. W. & Tyler, S. W. (eds.) *Intelligent User Interfaces*, 177–206. Addison-Wesley: Reading, MA.

Cohen, P. R. (1978). *On Knowing What to Say: Planning Speech Acts.* Ph.D. diss., University of Toronto, Toronto, Canada. Also available as 118, University of Toronto, Toronto, Canada.

Cohen, P. R. & Levesque, H. J. (1987a). *Persistence, Intention, and Commitment.* 415, Artificial Intelligence Center, SRI International, Menlo Park, CA.

Cohen, P. R. & Levesque, H. J. (1987b). Rational Interaction as the Basis for Communication. 89, Stanford University Center for the Study of Language and Information.

Cohen, P. R. & Perrault, C. R. (1979). Elements of a Plan-based Theory of Speech Acts. *Cognitive Science* **3**: 177–212.

Duda, R. O., Hart, P. E., Nilsson, N. J. & Sutherland, G. L. (1978). Semantic Network Representations in Rule-Based Inference Systems. In Waterman, D. A. & Hayes-Roth, F. (eds.) *Pattern-Directed Inference Systems*, 155–176. New York: Academic Press.

Faletti, J. (1982). PANDORA – A Program for Doing Commonsense Planning in Complex Situations. In Proceedings of *The Second National Conference on Artificial Intelligence*, 185–188. Pittsburg, PA: AAAI Press.

Fikes, R. E. & Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* **2**(3–4): 189–208.

Forgy, C. L. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* **19**: 17–37.

Friedland, P. E. (1980). *Knowledge-Based Experiment Design in Molecular Genetics.* Ph.D. diss., Computer Science Department, Stanford University, Stanford, CA.

Friedland, P. E. & Iwasaki, Y. (1985). The Concept and Implementation of Skeletal Plans. *Journal of Automated Reasoning* **I**: 161–208.

Hammond, K. J. (1986). CHEF: A Model of Case-based Planning. In Proceedings of *The Fifth National Conference on Artificial Intelligence* **1**, 267–271. Philadelphia, PA: AAAI Press.

Hendler, J. A. (1985). *Integrating Marker-Passing and Problem-Solving.* Ph.D. diss., Computer Science Department, Brown University, Providence, RI. Also available as CS-85-08, Computer Science Department, Brown University, Providence, RI.

Kolodner, J. L., Simpson, R. L. & Sycara-Cyranski, K. (1985). Model of Case-Based Reasoning in Problem Solving. In Proceedings of *The Ninth International Joint Conference on Artificial Intelligence*, 284–290. Los Angeles, CA: Morgan Kaufmann Publishers.

McDermott, J., Newell, A. & Moore, J. (1978). The Efficiency of Certain Production System Implementations. In Waterman, D. A. & Hayes-Roth, F. (eds.) *Pattern-Directed Inference Systems*, 155–176. New York: Academic Press.

Meehan, J. R. (1976). *The Metanovel: Writing Stories by Computer*. Ph.D. diss., Department of Computer Science, Yale University, New Haven, CT. Also available as tr074, Department of Computer Science, Yale University and through Garland Publishing, New York.

Meehan, J. R. (1981). TALE-SPIN. In Schank, R. C. & Riesbeck, C. K. (eds.) *Inside Computer Understanding*, 197–226. Hillsdale, NJ: Lawrence Erlbaum.

Newell, A. & Simon, H. A. (1963). GPS, a Program that Simulates Human Thought. In Feigenbaum, E. A. & Feldman, J. (eds.) *Computers and Thought*. New York: McGraw Hill.

Rau, M. (1987a). Information Retrieval from Never-ending Stories. In Proceedings of *The Sixth National Conference on Artificial Intelligence* **1**, 317–321. Seattle, WA: AAAI Press.

Rau, M. (1987b). Spontaneous Retrieval in a Conceptual Information System. In Proceedings of *The Tenth International Joint Conference on Artificial Intelligence*, 155–162. Milano, Italy: Morgan Kaufmann Publishers.

Rosenbloom, P. S. & Newell, A. (1982). Learning by Chunking: Summary of a Task and a Model. In Proceedings of *The Second National Conference on Artificial Intelligence*, 255–257. Pittsburgh, PA: AAAI Press.

Sacerdoti, E. D. (1974). Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence* **5**(2): 115–135.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior*. Amsterdam: Elsevier North-Holland.

Schank, R. C. & Abelson, R. P. (1977). *Scripts, Plans, Goals, and Understanding*. Hillsdale, NJ: Lawrence Erlbaum.

Searle, J. R. (1969). *Speech Acts; An Essay in the Philosophy of Language*. Cambridge, England: Cambridge University Press.

Stefik, M. (1980). *Planning with Constraints*. Ph.D. diss., Computer Science Department, Stanford University, Stanford, CA. Also available as 80–784, Computer Science Department, Stanford University, Stanford, CA.

Stefik, M. (1981). Planning and Meta-Planning (MOLGEN: Part 2). *Artificial Intelligence* **16**: 141–170.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition*. New York: American Elsevier.

Tate, A. (1975). Interacting Goals and their Use. In Proceedings of *The Fourth International Joint Conference on Artificial Intelligence*, 215–218. Tbilisi, Georgia: Morgan Kaufmann Publishers.

Waldinger, R. (1977). Achieving Several Goals Simultaneously. In Elcock, E. W. & Michie, D. (eds.) *Machine Intelligence 8*. New York: Halstead/Wiley.

Warren, D. H. D. (1974). *WARPLAN: A System for Generating Plans*. Memo 76, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh. Edinburgh, United Kingdom.

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

Wilensky, R. (1987). *Some Problems and Proposals for Knowledge Representation*. UCB/CSD 87/351, Computer Science Division, University of California, Berkeley, CA.

Wilensky, R., Arens, Y. & Chin, D. N. (1984). Talking to UNIX in English: An Overview of UC. *Communications of the ACM* **27**(6): 574–593.

Wilensky, R., Chin, D. N., Luria, M., Martin, J., Mayfield, J. & Wu, D. (1988). The Berkeley UNIX Consultant Project. *Computational Linguistics* **14**(4): 35–84.