Modeling the Goal-Driven Learning of Novice UNIX Users

Alex Quilici
Department of Electrical Engineering
University of Hawaii at Manoa
2540 Dole St, Holmes 483
Honolulu, HI, 96822

E-mail: alex@wiliki.eng.hawaii.edu

Abstract

This paper describes a model of the goal-driven learning process by which novice UNIX users acquire UNIX expertise. Transcripts indicate that one way users acquire knowledge about UNIX commands is by planning to achieve UNIX-related goals, having these plans fail, and then trying to explain and repair these failures. This process gives rise to numerous questions, or knowledge goals, which themselves require planning, execution, failure explanation, and repair. We model this process with a variant of case-based planning that has been extended to explicitly represent and plan for knowledge goals and include them in the planning process. This paper presents this model and provides an overview of its implementation.

1 Introduction

Our interest is in modeling the incremental learning process by which novice UNIX users become UNIX experts. Transcripts show that they learn by making mistakes, such as executing the wrong command, and then trying to understand why those mistakes occurred [7, 8]. These transcripts make clear that this process is both failure-driven [13, 14, 15, 11, 2] and goal-driven [10, 6]: user planning failures cause users to ask themselves numerous questions and to explicitly plan to find out answers to these questions. In particular, these users execute plans, observe and record their effects, try to explain those effects, and then try to repair plans or modify goals based on those explanations. But unlike most models of the planning process [2, 4, 1], they not only execute plans to achieve domain goals but also to find answers to questions that arise during planning, such as which plan is appropriate for a goal, why a plan had a particular effect, whether a particular condition holds, and so on.

Figure 1 is an example of a novice UNIX user struggling to remove a file that turns out to be a non-empty directory. The user starts off knowing only that "rm" is the command for removing files, and winds up knowing that "rm" doesn't remove directories and that "rmdir" only removes non-empty directories.

The user's learning is driven by questions that arise from trying to achieve the goals of removing a file and removing a directory and from trying to explain why the plans tried for these goals failed. The user starts with the goal of removing a file, which gives rise to the question "What's the plan for removing a file?" The user recalls that the plan is to use the "rm" command and executes this plan. That results in an unexpected effect: the message "rm: junk is a directory." That, in turn, gives rise to the question "Why did this effect occur?" The explanation the user forms is that the error message resulted because "rm" failed to remove "junk", and that's because "rm" can't remove directories and "junk" is a directory. This potential explanation gives rise to the questions "Is 'junk' a directory?" and "Was 'junk' removed?" The user then executes the "ls -l" command to try to answer these questions. As a result of confirming this explanation, the user revises his original goal to the more specialized goal of removing a directory named "junk" and runs through a similar process to try to achieve this new goal.

This paper describes our model of this learning process. At its heart, it extends case-based planning [2, 4] with question-driven text-understanding techniques [9, 12]. The model has been implemented in the QUACK program (QUestion-driven ACquisition of UNIX Knowledge) and has been applied to learning about several of the basic UNIX file manipulation commands.

```
[%] rm junk
rm:  junk is a directory
```
Why did I get this message? Because "rm" can't remove directories? Was "junk" removed? And was it a directory?
```
[%] ls -l
...
drw-r--r--   1 student  users      1048  Dec  1 17:13 junk
...
```
So "junk" is a directory and it was not removed. What is the command to remove a directory?
```
[%] man -k remove
...
rm, rmdir (1) - remove (unlink) files or directories
...
```
"rmdir" is the command to remove a directory.
```
[%] rmdir junk
rmdir:  junk:  Directory not empty
```
Does "rmdir" require the directory to be empty? Was "junk" not empty ?
```
[%] ls junk
login.bak cshrc.bak
```
How can I remove a non-empty directory? I have to make it empty and then use "rmdir". I can make it empty by removing each of the files in it.
```
[%] cd junk
[%] rm login.bak
[%] rm cshrc.bak
```
Now the "junk" directory should be empty.
```
[%] ls
```
The "junk" directory is now empty.
```
[%] cd ..
[%] rmdir junk
```

Figure 1: A sample session of a user trying to remove a file.

2 User Learning As Case-Based Planning

The user's session in Figure 1 *superficially* fits well into the case-based planning paradigm [2, 4]. Given a goal or combination of goals, a case-based planner searches memory to find a plan that satisfies as many active goals as possible, executes the plan, determines whether the plan failed by checking whether its observed effects achieve the goals or conflict with other goals, detects the source of the failure by constructing a causal chain of the steps and effects that lead to it, and repairs the plan by applying repair strategies associated with different classes of planning failures. This user recalls that using "rm junk" is the command for removing "junk", executes "rm junk", determines that it did not remove the file "junk", detects that the source of its failure is that the file is really a directory and "rm" cannot remove directories, repairs the plan by discarding it and using the "rmdir" command instead.

There are, however, several devils in the details of trying to model UNIX learning as case-based planning. First, *the user may be unaware of all plan effects immediately after its execution.* Instead, the user may execute additional plans to determine its effects and whether it succeeded or failed. For example, the only explicit effect of the user's executing "rm junk" is the message "junk is a directory". The user infers that "rm" did not remove the file and then confirms this inference by executing "ls". Second, *the user may execute various plans to find an appropriate plan for a goal or to verify a proposed causal explana-*

tion for a failure. For example, to find a plan for removing a directory, the user executed "man -k remove" and scanned its output. Similarly, after the user decided that "rm" failed because "junk" is a directory, the user executed "ls -l" to determine whether or not this hypothesis was correct. Third, *the user may be unable to fix a failure by repairing an existing plan.* Instead, the user may have to revise his goals or seek alternative plans. For example, after forming an explanation for why "rm" failed, the user specialized his goal to removing a directory, abandoned the plan of using "rm", and tried to find a plan that achieves this new goal. Similarly, after forming his explanation for why "rmdir" failed, the user generated an additional goal of making the directory empty.

The bottom line is that we cannot directly model the learning behavior of UNIX users with case-based planning because it assumes that that the planning process functions in large part independently from executing plans. That is, it assumes that plans can be found, their effects determined, their failures explained, and their flaws repaired, all without having to execute anything other than the plan that the planner has formulated for the primary goal. Unfortunately, this assumption does not hold for the planning process of novice UNIX users. That process instead gives rise to knowledge goals [10, 9, 16], which are goals to acquire a particular piece of information and which which may themselves have to be achieved before the planning process can continue. These knowledge goals correspond directly to the questions novice users ask about UNIX. In contrast, traditional planners focus on action goals, which are goals to cause some effect in the external world. Case-based planning, in particular, does not take knowledge goals into account, yet they are crucial for modeling our UNIX users. Thus, modeling users learning about UNIX requires extending case-based planning to represent knowledge goals explicitly and to include them as part of the planning process.

There are three types of knowledge goals that frequently arise with UNIX users: *know-how* (what plan will achieve a particular goal?), *know-if* (does or doesn't a particular assertion hold true?), and *know-why* (what is the explanation for why a particular relationship holds?). In Figure 1, the user had *know-how* goals to know how to remove a file, to know how to remove a directory, and to know how to empty a directory. These *know-how* goals arise when the user has a goal, such as removing a file, and needs to find a plan for achieving that goal. The user had *know-if* goals to know whether or not "junk" existed after trying to remove it with "rm", to know if "junk" was a directory, and to know whether or not "junk" was empty. These *know-if* goals arise when the user has formed an explanation and is trying to verify whether it is correct. Finally, the user had *know-why* goals to find out why "rm junk" and "rmdir junk" resulted in the messages they did. These *know-why* goals arise when the user encounters an unexpected effect or doesn't encounter an expected effect after executing a plan.

In terms of planning, knowledge goals must be processed differently than action goals. That's because action goals can only be achieved through doing external actions. For example, removing a file requires executing the "rm" command. On the other hand, knowledge goals can be achieved not only by doing specific actions, but also by memory search or by internal reasoning. For example, *know-how* and *know-if* goals can often be achieved by searching memory, and *know-why* goals can often be achieved by generating and processing explanation patterns [12]. This paper suggests an architecture for a modified case-based planner that supports goal-driven learning by treating knowledge goals as first class citizens of the planning process.

3 Necessary Knowledge

Our model requires plans for achieving action goals (action plans) and knowledge goals (knowledge plans), descriptions of domain objects and their attributes, and a set of generally applicable explanations for unexpected plan effects. We assume this initial knowledge closely corresponding to the standard "cheat sheet" given to

novice UNIX users. That is, it contains knowledge about which goals a small set of UNIX commands achieves, but contains no information about peculiar effects, required enablements, or specialized constraints on their use.

UNIX Planning Knowledge

We represent plans in a slight variant of standard planning representations [1]. Each plan has *parameters* (variables instantiated within the plan), *steps* (actions executed to carry out the plan), *goal effects* (effects that are assumed to occur if the plan is executed when its enablements have been achieved and its constraints hold), *side effects* (other states or actions that can from result from executing this plan and the conditions under which they occur), *enablements* (states or actions that must be achieved before the plan can achieve its goal effects), and *constraints* (subclasses of plan parameters for which the plan fails to achieve its goal effects). Plans begin with only their parameters, goal effects, and necessary steps filled in. Our model acquires detailed knowledge about their other attributes as these plans are executed and their results are analyzed.

In the UNIX domain, action plans differ only slightly from knowledge plans. In particular, action plans have steps that consist of the primitive action *execute-command*, which represents actually forming and giving a particular command string to UNIX. On the other hand, knowledge plans usually involve first executing action plans and then executing the primitive action *conditionally-assert*, which updates the system's knowledge base, given that a particular condition holds true.

Figure 2 shows the representation of the plan USE-RM-COMMAND that results after several different experiences with using it to remove various files. This plan achieves the action goal of removing a file and involves executing the UNIX command "rm filename". Its representation captures several pieces of planning knowledge. First, executing "rm" is the plan for removing a file, so long as the file is not a directory (a constraint). Second, "rm" requires write permission in the file's parent directory (an enablement). Third, "rm" produces an error message if the file is a directory (the constraint is violated). And finally, "rm" produces a different error message if there's no write permission on the directory that contains it (the enablement wasn't achieved).

In contrast, Figure 3 shows the representation of the plan, CHECK-IF-DIRECTORY, which achieves the knowledge goal of knowing whether a particular file is a directory and which involves executing the "ls -l" command and examining its results. This captures the planning knowledge that CHECK-IF-DIRECTORY is a plan that achieves the goal of knowing if a particular file is a directory. It consists of running the "ls -l" command and then updating the knowledge base depending on what it produces for output. Its possible effects are either updating the knowledge base to indicate that F is a directory or that F is a file.

These plans are organized and accessed through a standard discrimination net of goals.[2, 4]

UNIX Objects And Actions

Our model also requires a description of the types of UNIX objects and actions. This knowledge is represented in terms of a standard classification hierarchy describing possible attributes and the possible fillers for those attributes. The objects include files, directories, and accounts, and their attributes include items such as names, locations, and contents. The actions include accessing, creating, and removing these objects. Whenever *know-how* goals are achieved, the original hierarchy is extended by adding new specializations of existing actions, such as adding a new action of removing a directory as a specialization of removing a file.

Along with this classification hierarchy, our model contains a standard working memory containing instances of these objects and actions. These instances represent knowledge about the current UNIX environment and are used to represent specific goals, to track the effects of executing plans, and to achieve *know-if* goals without having to execute additional plans. Execut-

Parameters:	*a file F*
Goal Effects:	*remove F*
Constraints:	*F not a directory*
Enablements:	*write permission in F's parent directory*
Steps:	execute-command *"rm" with F's name*
Side Effects:	*displays prompt on line after command*
	WHEN: *remove F was achieved*
	displays message that F is a directory
	WHEN: *F is a directory*
	displays message that permission is denied.
	WHEN: *directory containing F isn't writeable.*

Figure 2: The representation of the plan USE-RM-COMMAND.

Parameters:	*a file F*
Goal Effects:	*know-if F is a directory*
Steps:	USE-LS-L-COMMAND
	conditionally-assert *F's type based on "ls" output*
Side Effects:	*know F's type is directory*
	WHEN: *display d at beginning of line*
	know F's type is file
	WHEN: *display - at beginning of line*

Figure 3: The representation of the plan CHECK-IF-DIRECTORY.

ing a plan results in creating, revising, and reclassifying these instances based on the plan's effects and the results of achieving *know-why* goals involving these effects.

General Effect Explanations

Finally, our model requires a set of domain-independent explanations for plan effects. These explanations are used to determine why a plan failed or produced an unexpected effect. They consist of a situation, an explanation, and, if the explanation indicates that a plan failed, a strategy for overcoming this planning failure. Our current model requires four of these explanations, all of which which deal with failures caused by problematic plan parameters. These are: the plan fails for a particular type of parameter, the plan fails for a particular role filler of a parameter, the plan works only with a particular type of parameter, the plan works only with a particular parameter role filler. Each effect explanation is intended to be general enough to apply in a variety of different situations. These explanations aid learning, because whenever an explanation applies, its repair strategy can correct mistaken beliefs in the knowledge base, as well as decide how to proceed next.

Figure 4 contains the effect explanation FAILS-FOR-SPECIFIC-PARAMETER-TYPE. It explains an unexpected plan effect consisting of a plan action that informs the user that a plan parameter has a particular type. The explanation is that the plan failed because it does not work for a parameter with that type. The repair is to generate a new goal that takes into account this parameter's type.

The user in Figure 1 can use this effect explanation to explain why executing "rm" produced the message "rm: junk is a directory". The explanation is that "rm" didn't remove "junk" because "junk" is a directory and "rm" can't remove directories. After accepting this explanation, the user updates memory to indicate that "junk" is a directory and that "rm" has a con-

Situation:	executing an instance P' of plan P produced effect E'
	where:
	P' was executed for goal G' and has parameters $X'_1 \ldots X'_n$
	E' is of the form: A informs X'_i names object of type T
	A is an action within P
Explanation:	P' produced effect E'
	because
	P does not have effect G when X_i is an instance of type T
	X_i' is an instance of type T
Repair:	generate goal with X_i' of type T
	mark P as inappropriate for this goal

Figure 4: The effect explanation FAILS-FOR-SPECIFIC-PARAMETER-TYPE.

straint that the object it removes can not be a directory. Regardless of what type the user believed "junk" was or whether the user knew "rm" had this constraint, the repair is the same: to try to achieve the goal of removing the directory "junk".

These effect explanations are similar in spirit to other knowledge structures such as explanation patterns (XPs) [14, 12] and thematic organization patterns (TOPs) [2, 4]. Our effect explanations differ from XPs in that XPs are specific explanations designed to be adapted to explain anomalous events during story understanding, while our effect explanations are more general explanations designed to be instantiated to explain why plans have anomalous effects. Our effect explanations differ from TOPs in that TOPs are collections of plan repair strategies for classes of plan failures that arise due to goal interactions, while our effect explanations contain repair strategies to address plan failures caused by mistaken user beliefs about plan parameters and applicability conditions and have repair strategies that revise these beliefs and generate new goals. Despite the differences, all of these knowledge structures are complementary.

4 A Modified Case-Based Planner

We model the process of a UNIX user trying to achieve a goal by modifying a standard case-based planner to explicitly generate knowledge goals as part of the planning process, and by adding a special component for achieving each type of knowledge goal. Each of these components first attempts to achieve the goal through memory search and task-specific reasoning and, if that fails, by then explicitly attempting to find and execute an external action to achieve the knowledge goal.

Figure 5 shows how QUACK models the process of a UNIX user trying to achieve a goal G. It begins by classifying G as either an action goal or a knowledge goal. It then handles the two classes of goals very differently. Action goals result in finding, executing, analyzing, and repairing a plan. Knowledge goals result in searching memory, performing goal-specific reasoning, and possibly generating action goals to obtain the desired knowledge from external sources.

Processing Action Goals

Given an action goal G, QUACK first tries to find a plan that achieves it. To do so, it generates a knowledge goal K to *know-how* to achieve G and then recursively tries to achieve K. The outcome is either a plan for achieving G or an indication that K can't be achieved.

After finding a plan, QUACK executes the plan by passing it to the EXECUTOR. It executes an *execute-command* action by passing the command to a UNIX simulator, which simulates its execution, and returns a conceptual representation of its observable results. It executes a *conditionally-assert* action by simply checking

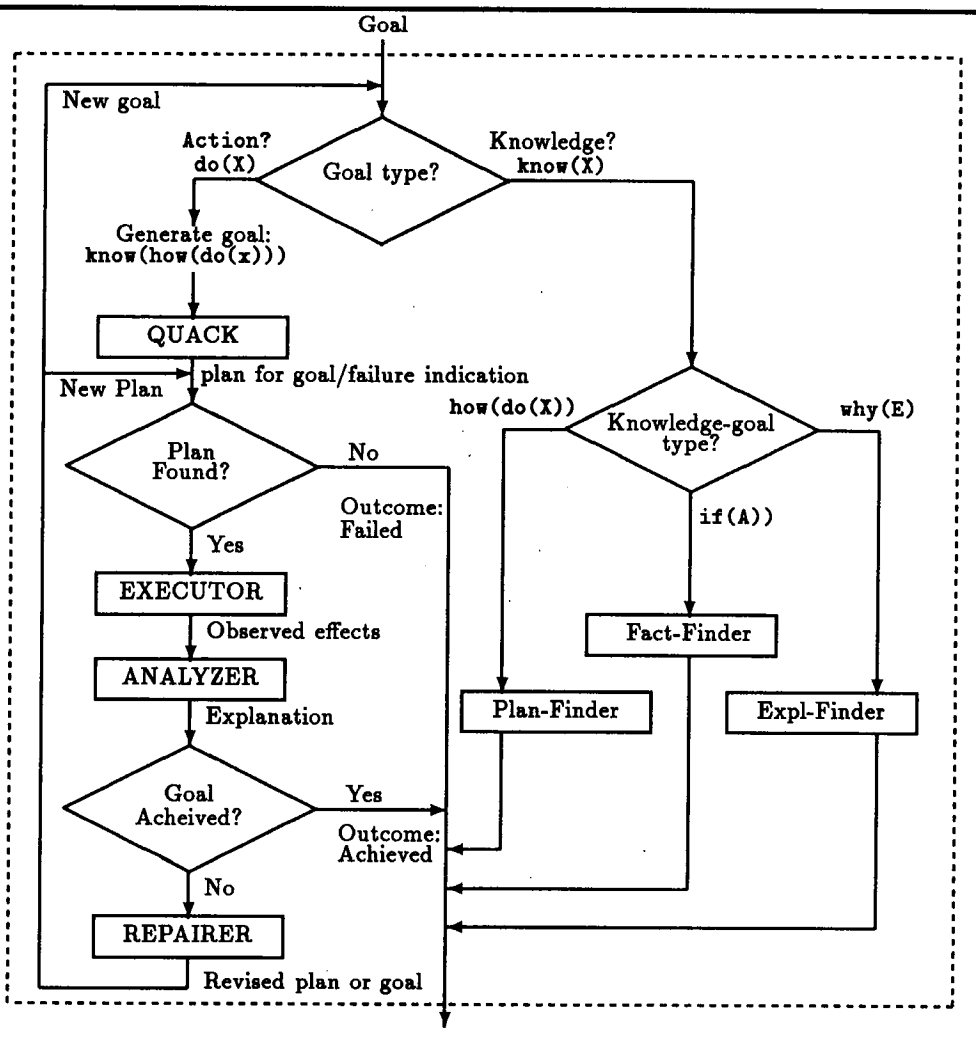

Figure 5: How QUACK attempts to achieve a goal.

whether the condition is known to be true or false and making the corresponding assertion to memory.

Once QUACK has executed a plan, it needs to determine what happened and why. That is, it must determine whether the plan achieved the goal and explain any unexpected effects. This task is done by the ANALYZER, which compares the observed effects to the effects expected to be observed. It generates a *know-if* goal for each expected effect that wasn't observed and a *know-why* goal for each observed effect that wasn't expected. It then tries to recursively achieve these knowledge goals. The result is an indication that the goal has been achieved or an explanation for the plan failure.

The final step is to try to overcome the plan failure. That's the job of the REPAIRER. It takes the ANALYZER's explanation for the failure and applies the repair strategy to generate a revised goal to achieve or a revised plan to execute.

Processing Knowledge Goals

Given a knowledge goal K, QUACK sends it to a component that depends on K's type: *know-how* goals go to the PLAN-FINDER, *know-if* goals go to the FACT-FINDER, and *know-why* goals go to the EXPLANATION-FINDER. Each of these components updates the knowledge base and returns the desired knowledge or an indication that the

knowledge goal can not be achieved.

The PLAN-FINDER is given a goal K to know how to achieve a particular goal G. It first searches memory to determine whether it contains a plan that achieves G. If it does, then K has been achieved and this plan can be returned. However, if it doesn't, the PLAN-FINDER then gives QUACK a goal to do some external action to achieve K. QUACK then takes this goal and generates a new knowledge goal to know how to achieve it, which is then passed back to the PLAN-FINDER. Essentially, this new knowledge goal is to know which plan to execute to achieve the original goal of knowing how to achieve G. The PLAN-FINDER is always able to find some plan for this new goal, as memory contains a default plan of asking someone for the necessary information. But memory also contains more specific plans for achieving more specific knowledge goals. For example, from the original "cheat sheet" it contains knowledge that the plan for the goal of knowing how to achieve a UNIX-related action is to execute "man -k" with the desired action as an argument. The PLAN-FINDER returns the most specific plan possible to QUACK, which goes through the standard cycle of executing it, analyzing it, and possibly repairing it.

The FACT-FINDER is given a goal to know if some assertion A holds. It first tries to determine whether the assertion is already known to be true or false. This is done by searching memory for A and its negation. If it can't find either, it then tries to determine whether A can be derived from what it already knows. This is done by trying to prove that A or its negation holds. Finally, if the truth of the assertion is still unknown, the FACT-FINDER generates a goal to perform an action to find out whether that fact holds and gives this goal to the system to achieve.

The EXPLANATION-FINDER is given a goal to know why a plan had a particular effect. It first searches the knowledge base for a known explanation for this effect. If it can't find an explanation, it then tries to find and instantiate an applicable general explanation for the effect, and to then provide QUACK with *know-if* goals to verify whether each of its premises hold. If it can't find an explanation or if it finds that the explanation's premises are contradicted, it provides QUACK with a goal to perform some action to find out why the effect occurred. Whenever it finds a satisfactory explanation, it adds all of its premises to knowledge base.

5 Limitations

We have implemented our model and had it participate in a variety of scenarios involving simple UNIX file manipulation commands. It is capable of processing approximately half of the messages produced by these commands in normal user use. One implementation limitation is that it interacts a simulation of UNIX and not directly with UNIX itself. That is, our implementation currently works with conceptual representations of command output, not the actual output text itself. The model itself also several significant limitations.

Limited Explanatory Capabilities

Our model's current effect explanations deal only with descriptions of object classes and attributes. However, many command responses describe actions that did not or could not occur (messages such as "command not found", "cannot remove .", and "permission denied"). Along the same lines, our model focuses on finding an appropriate general explanation to apply to a specific situation, which limits it to those situations explainable by instances of its general explanations. An alternative is to adapt more-specific explanations for closely-related situations, as was done for understanding novel events in stories [12, 5].

Limited Knowledge Goal Types

Our model currently handles only *know-how*, *know-if*, and one class of *know-why* knowledge goals. However, transcripts show that other knowledge goals also frequently arise, such as to know what the filler of a particular role should be. For example, the explanation for the effect "permission denied" is that the plan failed because there is an unfulfilled enablement of hav-

ing "sufficient" permission to do the goal action. This gives rise to a goal of knowing what exactly is sufficient permission to remove a file (it turns out to be write permission on the directory containing the file).

Similarly, transcripts show other *know-why* goals besides explaining an unexpected plan effect. In particular, users must often explain why a conflict has arisen between their beliefs and what they infer as a result of a plan's execution. An example is a user who executes the command "ls old", sees a file "junk" inside "old", and then tries to "remove junk". This results in the error message "junk not found". Using effect explanations, our model assumes that this effect indicates that "rm" failed to remove "junk" because it has an unfulfilled enablement that "junk" exists. But this conflicts with the belief that "junk" exists that arose from the previous "ls". This gives rise to goal to explain this conflict (the explanation is that "ls" doesn't change to the directory it is listing) and explaining this conflict is the key to overcoming this plan failure.

Limited Planning Capabilities

We deliberately kept our planner simple to allow us to focus on how to integrate explicit knowledge goals into the case-based planning process. As a result, it currently assumes that it has only a single active goal and therefore does not attempt to anticipate goal conflicts before executing plans or to determine whether plan effects affect other active goals, as does CHEF [2, 4]. Several other over simplifications are that our planner does not consider a plan's enablements before executing a plan, nor does it record and index failed or blocked goals so that they can opportunistically be achieved later, as does AQUA [9] (for knowledge goals that arise during story understanding) and TRUCKER [3], (for the action goals that arise during planning).

Limited learning focus. Our model focuses solely on acquiring knowledge about plan constraints, effects, and enablements, and about the types and role fillers of particular instances of known classes of objects. Transcripts of novice UNIX users, however, show that they also acquire knowledge about object role fillers, such as which characters are legal within a file name, and possible ways of referring to objects, such as that ".." is another way to refer to the object that fills the location role of the current directory.

6 Conclusions

This paper has presented a model of the goal-driven learning process by which novice UNIX users become UNIX experts. Essentially, it extends case-based planning to treat knowledge goals as first-class citizens of the planning process. In particular, we have modified the planner to generate knowledge goals whenever it requires new information, to provide explicit components that process different classes of knowledge goals, and to have a collection of specific plans for achieving specific knowledge goals. Each component handles knowledge goals by first finding out if the information already exists in or can be derived from the knowledge base, and if that fails, by then generating goals to find specific plans to achieve these knowledge goals.

Our current model has a variety of shortcomings, but its planning architecture appears to be easily extensible to handle other classes of knowledge goals that arise during UNIX learning, as well as to acquire knowledge needed to provide advise in other plan-based domains. In particular, handling a new class of knowledge goals requires only modifying the planner to generate them and adding a new component to process them. And handling a new plan-oriented domain requires only adding domain-specific plans for domain-specific knowledge goals.

References

[1] J.F. Allen, J. Hendler, and A. Tate (Eds), *Readings in Planning*, Morgan Kaufman:San Mateo, CA, 1990.

[2] K. J. Hammond, "Explaining and Repairing Plans That Fail", *Artificial Intelligence*, vol. 45–2, pp. 173–228, 1990.

[3] K.J. Hammond, "Opportunistic Memory", in *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Detroit, MI, 1989, pp. 504–510.

[4] K.J. Hammond, *Case-Based Planning: Viewing Planning as a Memory Task*, Academic Press:Cambridge, MA, 1989.

[5] A. Kass, "Developing Creative Hypotheses by Adapting Explanations", Institute for the Learning Sciences, Northwestern University, Evanston, IL, Technical Report #6, 1990.

[6] D. Leake and A. Ram, "Goal-Driven Leaning: Fundamental Issues and Symposium Report", Cognitive Science Program, Indiana University, Bloomington, IN, Technical Report #85, 1993.

[7] P. McKevitt, "Acquiring User Models for Natural Language Dialogue Systems through Wizard-of-Oz Techniques", in *Proceedings of the Second International Workshop on User Modeling*, Honolulu, HI, 1990.

[8] P. McKevitt, "Wizard-of-Oz Dialogues in the Computer Operating Systems Domain." Computing Research Laboratory, New Mexico State University, Las Cruces, New Mexico, Technical Report MCCS-89-167, 1989.

[9] A. Ram, "A Theory of Questions and Question Asking", *Journal of the Learning Sciences*, vol. 1–4, pp. 273–318, 1991.

[10] A. Ram and L. Hunter, "The Use of Explicit Goals for Knowledge to Guide Inference and Learning", *Applied Intelligence*, vol. 2–1, pp. 47–73, 1992.

[11] C. Riesbeck, "Failure-Driven Reminding for Incremental Learning", in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vancouver, Canada, 1981, pp. 115–120.

[12] R.C. Schank, *Explanation Patterns: Understanding Mechanically and Creatively*, Lawrence Erlbaum:Hillsdale, NJ, 1986.

[13] R.C. Schank, *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*, Cambridge University Press:New York, NY, 1982.

[14] R.C. Schank and D. Leake, "Creativity and Learning in a Case-Based Explainer", *Artificial Intelligence*, vol. 40–3, pp. 353–385, 1989.

[15] G. Sussman, *A Computer Model of Skill Acquisition*, American Elsevier:New York, NY, 1975.

[16] D. Wu, "Active Acquisition of User Models: Implications for Decision Theoretic Dialog Planning and Plan Recognition", *User Modeling and User-Adapted Interaction*, vol. 1–2, pp. 149–172, 1991.