Artificial Intelligence Review **14:** 23–42, 2000.
Stephen J. Hegner et al. (eds.), Intelligent help systems for UNIX.

Virtues and Problems of an Active Help System for UNIX

MARIA VIRVOU[1], JOHN JONES[2] and MARK MILLINGTON[3]
[1]*Department of Computer Science, University of Piraeus, 80 Karaoli & Dimitriou St., Piraeus 18534, Greece. E-mail: mvirvou@unipi.gr;* [2]*Department of Computer Science, Hull HU6 7RX, UK;* [3]*Interactive Business Systems, Inc., 2625 Butterfield Road, Oak Brook, Illinois 60521, USA. E-mail: markmillington@msn.com*

Abstract. An empirical study undertaken on a cross-section of UNIX users at an academic site reveals a role for an active form of help system, rather than the more usual passive kind. Sample scripts supporting this view are presented and the kind of aid required for these examples is discussed. It is then proposed that to provide such aid requires the construction and maintenance of an individual model of each user.

Keywords: advice generation, intelligent help systems, user's errors, user interfaces, user modelling, UNIX commands

1. Introduction

The aim of this paper is to motivate, through examples and discussion, work on active aid systems for the UNIX file store manipulation domain. Most current help systems are passive, requiring a user to explicitly ask for help on a particular topic. Examples of this approach include the on-line UNIX manual and intelligent help systems such as UC (Wilensky et al. 1986). In contrast (in common with Breuker 1988; Jerrams-Smith 1985; Kemke 1986; Matthews et al. 2000) we will propose a role for an (automatic) active aid system to act as an "over the shoulder" adviser. Such a system is very difficult to construct and we discuss a number of the problems and issues that arise if this route is taken.

The remainder of this section introduces a number of challenging examples used to illustrate points made in later sections. Section 2 discusses some advantages of an active help system, while Section 3 outlines the modelling problems such a system may face. In Section 4, we turn to the issues to be dealt with as a system prepares to interrupt a user. In Section 5, we summarize our approach in building a system meant to tackle some of the problems that have been highlighted in this paper, and we also discuss the results of the automatic aid, with respect to the examples introduced in the present section. In Section 6, we give an account of recent developments in this research.

1.1. *Motivating examples*

This paper arises from an empirical study undertaken on a cross-section of UNIX users at an academic site. The particular version of UNIX that was used was Berkley 4.2 on a Vax; in fact, this version makes a lot of difference to many of the examples given below, for other implementations might behave rather differently.

Volunteer subjects included faculty members, research staff and post-graduate students. Each subject was automatically monitored by the shell with which s/he was interacting; in an unobtrusive manner, as users went about their normal activities on the system, each command issued was recorded in a log-file associated with that user.

However, due to the volume of feedback from UNIX, it was impossible to record the replies of UNIX to the commands issued. Thus, for example, error messages and the output from commands like *ls* were missing from the log and had to be reconstructed, as seen below. We reconstruct four partial sessions, two from each of two users, to illustrate many of the points we wish to make about the desirability and difficulties of an active help system (hereafter called an "aid system").

The summaries given with these reconstructions are intended to enable a reader to rapidly absorb the content of what happened. For the same reason, we have also made comments next to some commands. These comments start with a hash symbol.

These summaries and comments did not arise from interviews with the subjects, and merely present a single interpretation of what occurred. Even so, it is quite remarkable what a wealth of information about each user lies in these examples, waiting to be extracted by an (intelligent!) aid system.

The poplog *example*

Anna has a sub-directory *tf* of her home directory which she wishes to move into a new sub-directory, to be called *poplog*. This can be achieved by *mkdir poplog*; *mv tf poplog*. However, instead of using *mv*, she tries first to copy *tf* into *poplog* (commands 2–5), then checks that the copy worked (commands 6–8), and finally removes the original (commands 9–17).

Unfortunately, in this implementation of UNIX, command 5 will produce a data file which describes the directory entries for *tf*; a very similar effect can be gained by:

```
% cat tf > poplog/tf/tf
```

This form of *cp* is intended for use by experienced users engaged in system administration functions, who are a class of users very different from Anna.

Anna does try (command 7) to check that command 5 has created a directory, suggesting she has some inkling of the limitations of *cp*, but the results of this test are consequent only on (commands 1–3); the *tf* listed here is the *tf* created in (command 3), not the *tf* copied in (command 5). Alas, she is satisfied and goes on to remove the (still unique) original.

```
 1 % mkdir poplog
 2 % cd poplog
 3 % mkdir tf                 #Makes a new directory ‘tf’.
 4 % cd
 5 % cp tf poplog/tf          #This command does not do anything
                              #sensible when applied to directories.
                              #Unfortunately there is no error message
                              #to alert her.
 6 % cd poplog
 7 % ls -al                   #Has a new directory been created?
   total 3
   drwxr-xr-x 3 anna          512 May 16 18:44 .
   drwxr-xr-x 3 anna          512 May 16 18:44 ..
   drwxr-xr-x 2 anna          512 May 16 18:44 tf
 8 % cd
 9 % rm tf                    #Tries to remove the old ‘tf’; but this
                              #command does not apply to directories.
   rm: tf is a directory      #Gets an error message this time.
10 % rm -i tf                 #Tries again without success.
   rm: tf is a directory
11 % man rm                   #Turns to the manual.
12 % rmdir tf
   rmdir: tf: Directory
   not empty
13 % cd tf
14 % rm house1                #‘house1’ is lost.
15 % rm house2                #‘house2’ is lost too.
16 % cd
17 % rmdir tf                 #The old ‘tf’ eventually removed.
```

The ht *example*
Graham wishes to collect all his files from his current working-directory into a new sub-directory called *ht*. To achieve this he creates the new directory (command 1), copies everything (including *ht*!) into *ht* (command 2), enters *ht* to delete the data-file copy of *ht* itself (commands 4–8), and then neatly removes everything (except *ht*) from the original directory (command 10).

As in the *poplog* example, a directory has been copied as a data-file, but in this case, the user is seen to be well aware of the behaviour of *cp* and *rm* when applied to directories rather than files.

```
 1 % mkdir ht          #Creates a directory 'ht'.
 2 % cp * ht           #The newly created 'ht' is also matched by '*'.
 3 % ls                #The current working directory contains two more
   intro ht notes      #files apart from the new directory 'ht'.
 4 % cd ht             #Gets into 'ht'.
 5 % ls                #The directory 'ht' contains a copy of the two
   intro ht notes      #files as well as a data-file copy of itself.
 6 % rm ht             #Removes the useless data-file copy of 'ht'.
 7 % cd .              #A typing mistake.
 8 % cd ..             #Rectification of the previous typo.
 9 % ls
   intro ht notes
10 % rm *              #'rm' removes the files 'intro' and 'notes'
   rm: ht is a         #which have been copied into 'ht'.
   directory           #'rm' does not apply to directories.
11 % ls                #'ht' is not removed.
   ht
```

The popcode *example*
Graham has two directories, *ccode* and *popcode*, which he believes to contain only unwanted files. He attempts to delete each directory by entering it, removing all of its files, leaving it, and then removing it. However, in *popcode* he discovers (we deduce from commands 6–9) that he has a sub-directory to deal with.

We note how clear it is that failure (of command 6) was *not* intended in this example as it was in (command 10) of the *ht* example.

```
 1 % cd ccode          #Gets into directory ‘ccode’.
 2 % rm *              #Removes all the files.
 3 % cd ..             #Gets into the parent directory.
 4  % cd popcode       #Gets into directory ‘popcode’.
 5 % ls
   lib zip.p
 6 % rm *              #Removes all the files.
   rm: lib is a        #This time there was a directory in there
   directory           #which was not removed.
 7 % cd ..             #Gets into the parent directory.
 8 % cd popcode        #Gets back to the ‘popcode’ directory,
                       #probably because (command 7) was issued
                       #very quickly after (command 6) and before
                       #seeing the resulting message.
 9 % cd lib
10 % ls                #Has a look at the contents of the unremoved
   arcs.p nodes.p      #directory.
11 % rm *              #Removes all the files in it.
12 % cd ..
13 % rm lib            #Tries to remove the directory; but command
   rm: lib is a        #does not apply to directories.
   directory
14 % rmdir lib         #The right command to remove the directory.
15 % cd ..
16 % ls
   ccode popcode
17 % rmdir ccode       #Removes the two empty directories.
   popcode
```

The perquish *example*

Anna wishes to collect two existing files, *fred* and *fred1*, from her home directory, together in a new directory called *perqish*, renaming them to *iread.pas* and *drconsult.pas*, respectively, in the process. However, at command 3 she mis-types *perqish* as *perquish*, an error which lies latent until command 6, and uncorrected until command 19! A version of this example is tackled in a companion paper in this volume (Jones et al. 2000).

```
 1 % mkdir perqish          #Creates a new directory ‘perqish’.
 2 % cp fred perqish        #Copies ‘fred’ into the new directory
                            #‘perqish’.
 3 % cp fred1 perquish      #Creates a new file ‘perquish’ as a copy of
                            #the file ‘fred1’; or was ‘perquish’ meant to
                            #be ‘perquish’ as at the previous command?
 4 % cd perqish
 5 % mv fred iread.pas
 6 % mv fred1               #She expects fred1 to be in the recently
   drconsult.pas            #created directory ‘perqish’.
                            #Gets an error message showing that ‘fred1’
   mv: fred1: No such       #is not actually there.
   file or directory
 7 % ls                     #Confirms ‘fred1’ is not in the recently
   iread.pas                #created directory ‘perqish’.
 8 % cd
 9 % cp fred1 perqish       #Repeats command 2 without the typo this
                            #time.
10 % cd perqish
11 % mv fred1 drconsult     #Back to the command that made her
                            #discover the error. But (command 11) is not
                            #identical to (command 6); an oversight?
12 % cd
13 % rm fred                #Another typo; ‘fred’ should read ‘fred1’.
   rm: fred: No such
   file or directory
14 % rf fred1               #A typo again! ‘rf’ should read ‘rm’
   rf: Command not
   found
15 % rm fred1               #Here is the correct command.
16 % ls                     #The accidentally created file ‘perquish’ is
   perqish perquish         #still there.
17 % rmdir perquish         #Tries to remove it unsuccessfully;
   rmdir: perquish:
   Not a directory
```

```
18 % rm perqish          #Another attempt fails because of another
   rm: perqish is a      #typo!
   directory
19 % rm perquish         #She removes the unwanted file.
20 % ls
   perqish               #Confirms the removal.
21 % cd perqish
22 % ls
   drconsult iread.pas
23 % mv drconsult        #Command 11 was not meant to be different
   drconsult.pas         #from (command 6) after all.
```

2. Some Points in Favour of Active Help

There are several questions concerning automatic aid systems that seek an answer in users' actual command sequences. First of all, is an aid system needed at all? In what way could an aid system offer help? Are there any cases where an active help system could be more suitable to help the user than a passive one? Some answers to these questions come straight out of real-life examples like those four illustrated in the previous section.

2.1. *Not realising help is needed*

There are many cases where users type other than what they mean, without any error message being produced to alert them. This does not mean that UNIX feedback is always to blame; in fact, it is not simple to spot those errors that UNIX would not have complained about without having reasoned about the user's actions and beliefs. Rather, this kind of reasoning is beyond the scope of UNIX or any other existing operating system, and hence, a help system would be needed instead.

However, even a passive help system would not have been of much use to a user who had not been aware of a problematic situation, because this user would not have turned to it; *how could one expect users to ask for help when they do not even know that they need it?* This is when an aid system would be useful, since an aid system is constantly supposed to watch and reason about the user's actions and make its own decisions, whether or not the user needs help. This can be one of the most important advantages of an aid system.

The sample scripts clearly illustrate this situation, where users make errors without being aware of them. This was clearly the case in the *poplog* example, as well as the *perquish* example. The consequences of such errors vary, depending on how soon the user realises there has been an error, and how difficult it is to recover from it, if this is at all possible.

In the remainder of this subsection, we are going to discuss the consequences of this kind of error, in the way they occurred in the *poplog* and the *perquish* examples. We are also going to point out the fact that inefficient usage of commands may well fall into the same category of problem, where users may not be aware that they possibly need help. The *popcode* example will be used to illustrate this point.

Catastrophic error in the poplog *example*
A simple *copy and remove* plan in the *poplog* example ended up in the single catastrophic action *remove* since the *copy* part of the plan failed without the user having realised it.

More specifically, Anna's misconception about the way directories are copied to other directories leads her to the erroneous command 5. The lack of an error message following this command results in command 9, where she attempts to remove a directory, without it having been copied first, contrary to what she believes. Initially, she only intended to move the directory elsewhere in the file store.

This is a very tricky case indeed, where the user continues issuing commands as though a previous goal has already been accomplished. She did not seek help (in the manual) before typing command 5, which was the cause of the trouble. Perhaps she thought she knew how to handle this. After which, she did not even realize her mistake, and therefore, would not have asked for help even if this had been available. This is a typical situation in which a spontaneous intervention of an aid system can save trouble. In this case, it could save the user from a catastrophic error.

Considerable effort wasted in the perquish *example*
The name of a directory called *perqish* is mistyped as *perquish* in command 3. As a result, a new file named after the mistyped name is created. At this point the user is not aware of her error. However, after command 6, she realises that there is a problem, unlike the previous example where she never even realised that something went wrong. Still, the recovery from this error proves quite expensive in terms of the number of commands.

As a matter of fact, she issued 23 commands to complete her initial goal, with 10 of them (commands 7, 8, 9, 10, 11, 16, 17, 18, 19 and 23) typed in the context of recovering from the typo and consequences of command 3. In this case, almost half of the effort was wasted due to a small typing error.

Evidently, it would not have been so much if she had not made other errors at commands 11, 17 and 18. However, it may have been worse. While trying to recover from her mistake, she gets involved in new ones and this goes on for some time.

This is a case where the lack of an error message from UNIX is absolutely justified, simply because the error occurs in the name of a directory or file. When new files are created, users are free to name files and directories as they please. Hence, UNIX cannot possibly complain when the user mistypes a name in a command to create a new file or directory. This kind of error could only be spotted if a system kept track of the user's intentions and beliefs, which means that an explicit *user model* would be necessary. An aid system, which is supposed to provide this kind of reasoning could help the user recover quickly from an error like this.

Another interesting point about this example is that this sort of typing mistake could have been made by anyone, novice or expert, suggesting that all types of users could benefit somehow from an aid system.

Optimisation in the poplog *and* ht *example*

Graham's goal in the *popcode* example could have been obtained by only 2 commands (*ls -R; rm -r ccode popcode*), instead of the 17 he issued! Similarly, the effect of the first 10 commands he issued in the *ht* example could have been obtained by using only 2 commands (*mkdir ht; mv * ht*). Would he be interested in seeing these very short command sequences, so that he could use them at some later time?

This is a case where the user is probably not aware of some system commands that would achieve a goal via a simpler plan than the one formed in the user's mind. Actually, many users probably feel safer using commands they are already familiar with, and do not bother to find out more about the operating system that they are using. Instead, they try to fit their needs to their existing knowledge. An aid system could suggest an optimisation of the user's very own actions. In this aspect, the aid system could act as a tutoring system. Again, the individuality of the user would play a significant role in deciding what help the aid system could provide.

2.2. *Not knowing how to ask for help*

Users may not know how to ask for help. It seems that sometimes the existing manual help is not sufficient for a user who turns to it for aid. The wealth of information may confuse the non-expert, who might pick up the wrong command for their case. There are times when users need advice for their individual case, and instead, all they can get is general information. Quite

often, they do not even know exactly what their problem is, and therefore, they cannot form the right questions.

Would they not be interested in getting an expert's advice, tailored to the individual circumstances from which their problem occurred? In the following two subsections we first illustrate this case where the manual proves inadequate, and then we describe the case where the user is looking for the wrong kind of information. Both cases occur in the *poplog* example.

Failure following the man *command in the* poplog *example*
Anna finds out about *rmdir* from the *rm* manual entry at command 11. However, command 12, coming right after the "*man rm*" command fails to remove her directory as Anna then wanted. In fact, she probably wanted *rm -r*, but did not manage to find it in the manual.

This is only an example of a case where the user finds it difficult to retrieve information from the manual. Anna was quite lucky to encounter only one failure before she used a plan which worked. There are even worse cases where some users do not even get to find the right keyword for the command that they seek.

Getting help on the wrong issue in the poplog *example*
Anna believes that she has to find out how to remove the directory *tf* at command 11 of the *poplog* example. But in fact, she would not want to remove it if she knew that it had not actually been copied first. In this case, the help she got from the manual did her no good. Ironically enough, no efficient reference facility would do any better if it did not have a user model. What Anna actually needed was a help system to tell her what was really going on, instead of just giving a reply to her question.

2.3. *Trying vs. looking up*

Although not always recommended, users may find it quicker to actually try commands out rather than looking them up. For example, Anna probably had some idea of the existence of the option *-r* in the *poplog* example. After the failure of command 9, because *tf* was a directory, she decided to take another guess before she issued the *man* command. However, she could not remember the exact name of the option she wanted, and in this case, typed the completely irrelevant option *-i*.

Perhaps this kind of user would be happier and safer if an aid system could guarantee interruptions at dangerous junctures; safety encourages experiment and experiment encourages learning. Bad typists, especially, would probably be happier to be interrupted, even if a facility for requesting help in (written) English were provided, because the latter would take some time to type.

2.4. *More human-like interaction*

Frustration can result for some users when the operating system is not able to spot even the most obvious (to a human) typing error. They somehow expect the computer to have some features of humans, and may get upset when commands fail because of some small typing error.

For example, a human expert would easily understand what Anna meant by command 14 in the *perquish* example. The whole context of the command sequence, and the similarity of the wrong command to the correct one, give sufficient clues to a human. However, intelligence is required for such an error to be spotted by the computer, especially when the typo can change the meaning of a command. This was the case in command 2 of the *perquish* example. A monitoring help system could make the interaction more human-like by recognizing these errors and interrupting discreetly like a human listener.

3. Modelling Requirements

In this section we address the issue of how accurately an aid system must model the user it serves. Firstly, we consider how clearly it must reason about what actually happened, using a general understanding of the interactive nature of command-driven systems like UNIX. Secondly, it is shown that in order to reason about what should have happened the aid system must maintain a model of each individual user.

3.1. *What happened?*

For an automatic system to provide acceptable on-line aid it must show a good understanding of the user-UNIX interaction. Whatever their differences, all users interact with a shell in basically the same way; the user issues a command and UNIX may issue a reply. This notion of interaction is important; it is not enough for an aid system to ignore the flow of information to a user and concentrate merely on the file store transformation. For example, we can strip all occurrences of *ls* from a command sequence leaving the effect (on the machine) unchanged, but this would mean that the aid system would not model the information that the user has seen. This might lead the aid system to interventions proposing rather suspect improvements. In the *popcode* example, the user learned that there was a sub-directory of *popcode*, only after attempting to remove it. After investigation it was eventually removed, but to suggest an improved sequence "*rm -r popcode*" with the same effect does not capture the true sequence of events.

Similarly, a user's current working-directory is an important object of manipulation, facilitating the simplified expression of (currently) non-local pathnames. If an aid system sees a command sequence *cd a; cd b; cd c*, should it propose an "equivalent" sequence *cd a/b/c* as being better? If it works it definitely is better, but the problem arises if the user is imperfect in some way (e.g. a bad typist, or forgetful about his file store environment), and then an error in the former is easier to recover from; if it does not work you will know why and be in the right place to do something about it. All users distinguish these sequences, and therefore, so must an aid system.

Thus, realistic aid systems will have to employ a rich model of the interaction between a user and UNIX. To do this, it must almost certainly make some generic assumptions about users; for example, users are (reasonably) attentive, consistent, sometimes forgetful, fat-fingered, cautious, etc. Modelling UNIX is the least of our problems.

3.2. *What was intended?*

Through an understanding of what actually happened we may hope to reason about what was intended, even though the two may be very different. If the two are significantly different, the user ought to be informed. One approach to this might be to wait for an error message to arise, and then attempt a diagnosis for the user. However, as we will see, this cannot work; some users make productive use of failure conditions, and some errors do not provoke error messages.

Providing effective aid must involve modelling individual characteristics of each user, for it is easily seen that not doing so, quickly leads to confusion. What aspects of a user must we model?

- We at least have to model the user's plan; suggesting "*rm -r **" as an improvement for command 10 of the *ht* example just to eradicate the error message would be ridiculous: the user is relying on the failure and does not want the extra effect of this proposal.
- The user's overall plan should have priority over a user's subplan; Anna would not thank us for helping her remove *tf* at command 9 of the *poplog* example, although this is obviously what she wants there; the overall plan is very different and must be taken into account.
- The individual user's command knowledge is important too; interrupting Graham at command 13 of the *popcode* example would be very different from interrupting Anna at command 9 of the *poplog* example (in the hypothetical case where the overall plan would not conflict with her wanting to remove *tf*). They both attempt to remove a directory by employing the command *rm* without any flag. However, Graham has

generally shown a better knowledge and understanding of command *rm* than Anna has in the previous examples. Therefore, Graham's mistake may have been an accidental slip, whereas Anna's mistake may have been due to a deeper confusion of commands. An intervention at command 9 of the *poplog* example (or, perhaps, command 10) might be acceptable: she plainly wants to get rid of *tf* but does not know how.

- Confusion may also arise over the environment in which a command is executed. At command 6 of the *popcode* example, "*rm lib*" fails because *lib* is a directory. Is this because he does not understand *rm* removes only files, or because he forgot *lib* was a directory? Knowing the user we would conclude the latter; in fact, later on in the *ht* example at command 10 the same user is making productive use of the same failure condition. However, if he had employed *ls -F* as command 5, we might be less sure.
- General and individual users' usual imperfections have to be modelled too; What would the user's intended typing, lexical, or syntactic structure be, if not that which was actually input? Answers to such questions should be somewhere in the model of users' usual errors. For example, the fact that Anna had mistyped *perqish* to *perquish* could have been suspected right away, only because the typing, lexical and syntactic structure of the command containing it was very similar to the previous one, and people tend to make this kind of typing error. However, as we have seen, different aspects are very heavily interrelated, and an existing model provides additional strong constraint on the sensible interpretations.

4. Interventions

Given that there is a useful point that can be made to the user, the question remains as to how, when, and on what basis to intervene.

How an intervention is made ought to depend on whether it proposes an optimisation or a correction; attention to a correction should be mandatory whereas offers of optimisation may be ignored. In this way corrections may be given as interruptions, like error messages, comprising a piece of English text to describe the problem and a proposed correction. Optimisations should not interrupt, but might instead alter the prompt to, say, a question-mark. If the user immediately hits return then the optimisation is given; otherwise, the user is free to issue a normal command.

There is little point in proposing a correction or optimisation that is beyond the current understanding of the user; if too many new ideas are introduced, it will confuse the user and be rejected. Here again, is a role for an individual user-model: to tailor advice to suit the user.

When to intervene, again, depends on the nature of the proposal. Optimisations can be given as soon as they are discovered, since they rely only on what actually occurred. Corrections deal with what was intended, when it differs from what occurred; in such cases there seem to be three points at which the aid system might intervene:

- When a correction first appears to be warranted; for example, command 3 in the *perquish* example. This may be too soon if there is not enough evidence for suspecting the command, since it seems likely that almost any command might be intended as something else. However, if there has been sufficient evidence that a problem may have occurred, then it would probably be better for the system to intervene sooner than later, to prevent further complications of this problem.
- When only a correction fits all the facts; i.e., we have no user-model in which the user is not under a misconception. For example, we might expect command 6 in the *perquish* example to be of this nature.
- When the user can no longer recover from the error if the current command is executed. There is no such point in the *perquish* example; in fact, she does make the recovery herself, but such a point occurs in the *poplog* example at command 14.

The basis for each intervention can, of course, be derived only from the events preceding the intervention, and it is not envisaged that we can hope to construct a dialogue between the aid system and the user; actions speak louder than words. However, the user model may also comprise a large proportion of the basis; Graham, copying a directory, is a very different matter from Anna, doing the same.

Some interventions produced by a prototype active help system for the sample scripts are presented in the following section.

5. Summary

An active help system could be of much use to many users, and not only novices. This is the conclusion drawn from the analysis of the sample scripts. It could save the considerable amount of time and energy wasted in recoveries from errors. It could also act as a personal tutor, allowing the user to escape needless inefficiencies.

The individuality of the user is of vital importance to the help an aid system should provide. This makes the error diagnosis more complex since both general user models and individual ones should be taken into account for a clear interpretation of the interaction leading to a good plan recognition. The latter is essential for a reasonably reliable error diagnosis. The user

modelling part should involve assumptions that the user could be imperfect in several aspects, such as having misconceptions about commands, making typing errors etc.

Interventions should be kept to a minimum. Only when it is very obvious that the user needs help, should this be provided. The only aim is to help the user, and make him/her more efficient. By no means should the opposite happen in any sense.

6. Recent Developments

In this section we describe briefly a prototype active help system called RESCUER (which stands for Reasoning System about Commands of UNIX using Evidence Reasonably) (Virvou 1992; 1998). RESCUER has been constructed to address users' problems like those described in earlier sections.

It is not within the scope of this paper to detail the modelling and reasoning issues of RESCUER. However, we will demonstrate the interventions produced by RESCUER when it was given, as input, the sample scripts presented in this paper.

A companion paper in this volume (Jones et al. 2000) describes a prototype system addressing some of the modelling issues raised here, but none of the intervention issues.

6.1. *RESCUER's overall approach*

RESCUER is a prototype active help system that monitors users interacting with UNIX and offers spontaneous help when it judges that users need it. RESCUER's design addresses three main problems that an active help system has to solve:

1. Recognition of a problematic situation.
2. Diagnosis of the cause of the problem, if there has been one.
3. Decision regarding the kind of advice and generation of response.

For the recognition of a problematic situation, RESCUER uses certain peculiarity criteria in order to evaluate every command typed. These criteria include the questions whether the command was acceptable to UNIX or not, whether it was typical of its class and so on. RESCUER uses these criteria as unconfirmed symptoms of a possible problem in a command type. The existence of such symptoms results in the further examination of the command typed. As a result of this evaluation, RESCUER attaches one of three labels to the command typed, namely "expected", "neutral", "suspect". The label

"suspect" means that RESCUER will have to generate alternative interpretations in its attempt to find one that fits the context better. The label "expected" means that the command's effects to the file store imply some continuation of a plan from previous commands; therefore RESCUER judges that the command fits the context very well. Finally, "neutral" means that RESCUER does not get alerted by this command because it considers it unharmful (e.g. "ls"), although it is not directly involved with the effects of previous commands.

The diagnosis of a problem is done by the user modelling component of RESCUER. If the user modeller can find an alternative interpretation about the user's command, such that it is similar to what was typed, but is better than that in terms of the evaluation criteria, then RESCUER may have to adopt this alternative interpretation. The process of generating an alternative interpretation contains the information needed for the construction of an explanation of what happened.

The generation of hypotheses about possible user's misconceptions is based on a cognitive theory, called Human Plausible Reasoning theory (Collins and Michalski 1989), which was originally constructed to formalise the reasoning that people use in order to make plausible guesses about questions for which they do not have full information. Here we exploit the fact that plausible guesses can be turned to plausible human errors.

Plan recognition is achieved by considering the UNIX file store as an entity that has certain properties, one of which is called "instability". The existence of instabilities imply a possible future transition of the current state of the file store to another state, which is expected to follow at some point (e.g. from a file store state that contains an empty directory to the file store state that this directory is filled with some content).

A file store is considered to be absolutely stable if it does not contain:

1. empty directories
2. directories with only one child
3. duplicate files

The reason that empty directories are considered to be adding some instability to the file store is that empty directories do not seem to have any purpose if they have no contents. One would expect them to either have contents or be removed.

Similarly, if a directory has only one child, it is considered to be borderline as to how useful it is to keep a directory with only one file. One would expect it to either have more contents, or the content file to be moved one level up and the directory to be removed.

Finally, duplicate files also imply some transition from a file store state that contains identical duplicates to a file store state where the original copies will have been removed, or the replicated files will have been changed.

Instabilities are only used to indicate that a user may have started some *plan*, but do not restrict the user's actions at all. An unstable file store does not mean that the user has made an error, but that perhaps, s/he is in a middle of a *plan*. A "stable file store" is considered to be every user's final *goal*, denoting the *completion of some plan.* In this way, RESCUER manages to evaluate users' typed actions in terms of their effects to the file store. RESCUER's expectations, in terms of effects to the file store, are used to guide the search for possible corrections of commands. A command is "expected", in terms of instabilities, if it adds and deletes instabilities at the same time (which means that there is some kind of continuity in the user's actions), or if it deletes instabilities without adding any new ones.

The generation of a response, if there is one, is done on the principle of preventing users from possible catastrophic errors, giving explanations that address the user's misconceptions and suggesting alternative correct plans. However, RESCUER mainly concentrates on errors that would be "plausible" for a human observer to identify.

6.2. *RESCUER's interventions for the sample scripts*

In this section we demonstrate RESCUER's responses when it was given, as input, the command sequences which were recorded in the sample scripts.

The poplog *example*

1. `% mkdir poplog`
2. `% cd poplog`
3. `% mkdir tf`
4. `% cd`
5. `% cp tf poplog|tf`

```
RESCUER: Did you mean to type:
          % cp -r tf poplog/tf
       or % mv tf poplog/tf?
```

RESCUER has considered the fifth command as "suspect" because it is not typical of "cp" command with respect to its purpose.

Since the command has been evaluated as "suspect", RESCUER generates the two alternatives which are similar to the command typed:

1. `% cp -r tf poplog/tf`
2. `% mv tf poplog/tf`

They are also found to be “expected”, because they assign meaningful content to the empty directory “poplog/tf”, and are considered better than the original command because they are more typical of their class.

The ht *example*
One major point raised from this example was that the user made a constructive use of the UNIX failure to remove directories by the command “*rm* *” (command 10).

RESCUER was smart enough to understand that command 10 was useful to the user. This command was used to remove all the original files that the user had previously copied to the directory “*ht*”. Command 10 had produced an error message because “*” did not apply to directories, and therefore, “*ht*” itself was not removed, which was precisely what the user wanted.

Although command 10 was considered by RESCUER as “suspect” because it produced an error message, it was also assigned the label “expected” because of the instabilities that it removed from the file store. Therefore, RESCUER did not intervene in this case.

The perquish *example*
RESCUER intervenes at the third command as follows:

1. `% mkdir perqish`
2. `% cp fred perqish`
3. `% cp fred1 perquish`

```
RESCUER: You may have made a typing mistake. Did you mean
         to type: ``cp fred1 perqish'' instead of ``cp fred1
         perquish''? Y/N.
```

The reasons why RESCUER has generated this response are the following:

First RESCUER considers command 3 as “suspect” because it introduces new instabilities without deleting any of the previous instabilities. At this point, RESCUER starts generating similar alternatives to the command typed. In this case, it generates four alternatives:

1. `% cp fred perquish`
2. `% cp fred1 perqish`
3. `% cp -r fred1 perquish`
4. `% mv fred1 perquish`

The first two are generated by changing the arguments of the action typed to other arguments (one at a time), which in this case are lexically similar.

The last two are generated by changing the command name and leaving the arguments as they are. The most similar commands to "cp" which would work for the preconditions of the semantics of this action have been found to be the commands "cp -r" and "mv".

Next, RESCUER evaluates the hypotheses in the same way as it evaluated the action typed. Two of them are found "suspect", one of them is found "neutral", and one is found "expected".

1. `% cp fred perquish` is found "suspect" because it has a similar effect to the file store as the command originally issued.
2. `% cp fred1 perqish` is found "expected" because it deletes the instability of "perqish" having only one child as a content.
3. `% cp -r fred1 perquish` is found "suspect" because it has exactly the same effect on the file store as the command originally issued.
4. `% mv fred1 perquish` is found "neutral" because it neither deletes nor adds any instability (therefore suggests that it is a plan of its own).

RESCUER picks the "expected" command, `% cp fred1 perqish`, to show the user, because at this point, it has "enough" evidence that the user may have made a typing error. There are two reasons completely independent from each other that suggest that the user may have made an error:

1. There is a command other than the command typed that is *very similar* to the command typed, according to the knowledge representation of RESCUER.
2. This other command is *different* from the command typed in the effects that it has in the file store. The "other command" is actually *better* than the command typed with respect to their difference.

This example demonstrates the successful automatic generation of a *plausible* correction. RESCUER favours an earlier intervention, rather than a later one. Therefore, it suggests a correction if there is more than one reason for believing that there may have been a mistake. Naturally, RESCUER may be unnecessarily invoked (which is not the case in this example) but even so, the user can always ignore RESCUER's suggestion.

Acknowledgements

We thank Peter Norvig, Wolfgang Wahlster and Robert Wilensky for having made several helpful comments on an earlier version of this paper. We are also grateful to Stephen Hegner and Paul Mc Kevitt for their helpful comments on the current version.

References

Breuker, J. (1988). Coaching in Help Systems. In Self, J. (ed.) *Artificial Intelligence and Human Learning*, 310–337. London, UK: Chapman and Hall.

Collins, A. & Michalski R. (1989). The Logic of Plausible Reasoning: A Core Theory. *Cognitive Science* **13**: 1–49.

Jerrams-Smith, J. (1985). SUSI-A Smart User Interface. In Johnson, P. & Cook, S. (ed.) *Proceedings of the Conference of the British Computer Society Human Computer Interaction Specialist Group*. Cambridge University Press.

Jones, J. G., Millington, M. & Virvou, M. (2000). An Assumption-based Truth Maintenance System in Active Aid for UNIX Users. In Hegner, S., Mc Kevitt, P., Norvig, P. & Wilensky, R. (eds.) *Intelligent Help Systems for UNIX*. Dordrecht, The Netherlands: Kluwer Academic Publishers (this volume).

Kemke, C. (1986). *The SINIX Consultant: Requirements, Design, and Implementation of an Intelligent Help System for a UNIX Derivative*, Bericht Nr. 11, FR. 10.2 Informatik IV, University of Saarland, Saarbrucken, FRG.

Matthews, M., Pharr, W., Biswas, G. & Neelakandan, H. (2000). USCSH: An Active Intelligent Assistance System. In Hegner, S., Mc Kevitt, P., Norvig, P. & Wilensky, R. (eds.) *Intelligent Help Systems for UNIX*. Dordrecht, The Netherlands: Kluwer Academic Publishers (this volume).

Virvou, M. (1992). *User Modelling Using a Human Plausible Reasoning Theory*. Ph.D. thesis, CSRP 251, School of Cognitive and Computing Sciences, University of Sussex, Brighton BN19QH, UK.

Virvou, M. (1998). RESCUER: Intelligent Help for Plausible User Errors. In *Proceedings of ED-MEDIA/ED-TELECOM 98*, World Conferences on Educational Multimedia and Educational Telecommunications, Vol. 2, pp. 1413–1420.

Wilensky, R., Mayfield, J., Albert, A., Cox, C., Luria, M., Martin, J. & Wu, D. (1986). *UC – A Progress Report*, Report no. UCB/CSD 87/303, Computer Science Division (EECS), University of California at Berkeley, California 94720, USA.