ELSEVIER

Interacting with Computers 11 (1999) 545–573

Interacting with Computers

Automatic reasoning and help about human errors in using an operating system

Maria Virvou

Department of Computer Science, University of Piraeus, 80 Karaoli and Dimitriou St, Piraeus 18534, Greece

Abstract

Human errors occur frequently in the interaction of a user with an operating system. However, current user interfaces of operating systems lack some reasoning ability about user's intentions and beliefs. Intelligent Help Systems (IHS) can provide additional reasoning and help. This paper presents a discussion of the features of IHSs and a review of a few IHSs for users of operating systems. Then it describes the research and results of employing a cognitive theory of Human Plausible Reasoning Theory in error diagnosis for users interacting with an operating system. This theory has formalized the reasoning based on similarities, generalizations and specializations that people use to make plausible guesses about questions. Here we exploit the fact that plausible guesses can be incorrect and thus turned into human errors. The error diagnosis is performed by the user modelling component of an IHS, called RESCUER. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: User-modelling; Error diagnosis; Human Plausible Reasoning; User interfaces; Intelligent Help Systems

1. Overview

When one user gives a command to the computer to do something that s/he does not really mean, then an error has been made. The consequences of errors vary depending on what is achieved by the command given as opposed to what was desired. Sometimes, errors which are made by the computer users themselves may be catastrophic with respect to the users' real intentions. For example, users may accidentally delete files that contain information which may be far from unwanted.

It seems that user interfaces, no matter how friendly they may be, lack an important feature of some reasoning ability about the actual way that they are being used by human users. On the other hand, human users often employ an approximate reasoning based on analogies, generalizations and specializations which is very good as a heuristic method for understanding, discovering and making guesses but is error prone, especially when interacting with a computer, which can only interpret very precise instructions.

PII: S0953-5438(98)00043-5

This belief has been formulated by analysing the results of an empirical study [1,2] made on real users' interactions with UNIX, which is an operating system having a command-driven user interface. During this empirical study, users were often observed to give UNIX commands that they did not really mean since they were incompatible with their intentions. However, problems and errors similar to the ones encountered by the UNIX users may well occur to users of other operating systems. For example, users of a GUI interface of an operating system such as Windows 95 still have to do some actions in order to carry out a plan which they think serves their intentions well. In this case, the actions may be different from actions in a command–language interface, such as selecting some files and then clicking on a certain icon for them to be deleted or placed in the clipboard. However, even in user-friendlier interfaces like this, a user still needs to conform with the user interface's formalities and may end up initiating the wrong actions with respect to her/his intentions.

Intelligent Help Systems (IHS) are pieces of software that have been specially constructed to address the lack of sophisticated reasoning ability of existing software packages. Intelligent Help Systems operate on top of the original software package and are meant to provide more help to users than that of the user interface of the original package. However, if this kind of help was provided by the user interface of the original software package then there would be no need for the construction of IHSs. Actually, all the results achieved by IHSs could be used for the design of adaptive user interfaces which would provide more flexibility to human errors and more useful help to users than what they currently offer.

Intelligent Help Systems usually consist of four components, namely the Domain Knowledge, the User Modelling component, the Advice Generation component and the User Interface of the IHS. The Domain Knowledge provides the information about the domain of the software package that the IHS has been constructed for. The User Modelling component models the user's beliefs and intentions concerning the software package that s/he is interacting with. The Advice Generation component has to decide when and how the IHS is going to provide help. Finally, the User Interface of the IHS can be in natural language or something simpler.

RESCUER, which stands for 'REasoning System about Commands Using Evidence Reasonably', is a prototype Intelligent Help System that monitors users interacting with UNIX. RESCUER reasons about the observed users' actions and their possible errors and offers spontaneous help. Its reasoning focuses on error diagnosis. Hoppe [3] describes the term of 'error diagnosis' as a procedure of inference process by which incorrect user actions are recognized and analysed. Hoppe adds that checking the correctness of given user's actions is an essential problem related to error diagnosis which is not trivial, especially in domains where there is a variety of correct actions. Other researchers also point out that there is ambiguity in interpreting a user's behaviour because there are different explanations of observed incorrect user's actions [4]. Davis [5] notes that in spite of the important progress made by the scientific community, diagnostic reasoning is still a hard task. Finally, Cerri and Loia [6] find diagnostic reasoning particularly complex and note that in the case of incorrect task performance of humans one may consider applying a diagnosis only after a model of the human reasoning is available.

In view of the generally acknowledged complexity of the problem of error diagnosis, due to the need of an automatic generation of hypotheses about the user's reasoning, we explored the utility of a cognitive theory of Human Plausible Reasoning, which was adapted and implemented in the context of RESCUER.

Human Plausible Reasoning theory (from now on referred to as HPR) was constructed by Collins and Michalski [7] in order to provide a formal model of the reasoning that people use to make plausible guesses about questions for which they do not know the immediate answer. Plausible guesses are made based on analogies, generalizations and specializations and may be correct as well as incorrect. RESCUER exploits the fact that plausible guesses may be incorrect and hence turned into plausible errors. Thus, it uses this theory to gain more insight into possible users' misconceptions or accidental slips, especially in cases where these would be easily identified by a human expert who would observe the interaction. RESCUER's user modelling component provides the automatic reasoning about possible user's errors in order to help the user rectify them as quickly as possible (sometimes, even before the user realizes that there has been an error).

In this paper we focus our discussion on the sub-domain of file manipulation of operating systems, such as UNIX. The reason for this is that file (or folder) manipulation is performed by a wide range of users varying from computer experts to novices and accounts for a significant proportion of users' time spent using an operating system. The problems that users encounter in this sub-domain is representative of the domain as a whole. Moreover, the nature of a computer operating system is such that it is feasible to create an interface which can take an action issued by a user and reason about it before it is executed. In this way catastrophic errors might be prevented.

In Section 2 of this paper, there is a description of possible errors that users of operating systems may make, as these have been observed and analysed during an empirical study. Some of these errors may be catastrophic with respect to the user's intentions. In Section 3 we present a discussion of Intelligent Help Systems and a review of a few IHSs that have been developed to offer on-line assistance to users of operating systems. This review consists of a short description of the systems that highlights both their strengths and weaknesses with particular focus on assistance to users' errors. The main body of this paper in Sections 4, 5 and 6 presents the results of the research involved in the development of RESCUER. In this paper we focus our presentation on the user modelling component of RESCUER that provides the reasoning about users' errors. Finally, we give the conclusions drawn from this work.

2. Problematic situations and errors of users of operating systems

Users of operating systems often get themselves into problematic situations which are frequently leading to their making errors with respect to their intentions. This is something that most computer-users can acknowledge and has also been shown by an empirical study undertaken on a cross-section of UNIX users at an academic site [1,2]. Volunteer subjects included faculty members, research staff and post-graduate students. Each subject interacted with the user interface of the operating system in an unobtrusive manner as s/he went

about her/his normal activities on the system. At the same time, each command issued was recorded on a log-file associated with the particular user.

The hand-analysis of the samples collected, revealed that users make errors which fall into three categories, in terms of how difficult it is for users to realize that they have made the errors and rectify them.

1. Errors that the users have not realized that they have made and thus remain unrectified.
2. Errors that the users rectify themselves soon after they have made them.
3. Errors that cannot be rectified by the users irrespective of whether the users have realized their existence.

In terms of their consequences, serious errors can be catastrophic or simply difficult to rectify. However, even errors that are rectifiable may cause the user frustration for having to repeat some actions until they can work. If an automatic help system can give advice on errors it should definitely attach more importance to catastrophic errors than harmless ones. Gertner [8] gives a useful classification of errors in terms of their consequences:

1. Tolerable, probably harmless.
2. Non-critical but potentially harmful.
3. Critical, potentially fatal.

In the following subsections we give some examples of errors and problematic situations leading to errors, which reveal a role for an IHS that provides more sophisticated reasoning about users' actions than that currently available by operating systems.

2.1. Catastrophic errors

Naturally, the worst kind of error is a catastrophic error. In terms of the interaction of a user with an operating system, an example of a catastrophic error may result from the accidental removal (complete or partial) of important files. The accidental removal of a file may occur at various situations, the simplest of which is when the user issues a 'delete' action instead of some other action that s/he may have meant. In this case, the catastrophic error may be prevented by the operating system's request for confirmation of the action of deletion of certain files. For example, operating systems may prompt the user with a message reading: 'Are you sure you want files X to be deleted? Y/N'.

However, there are trickier situations where the accidental removal of some entities (directories, files, folders etc.) occurs in cases where the removal of the entities is only a part of a broader plan of the user that may have failed at a previous stage without the user having realized it. For example, a user was observed removing a directory which she probably thought she had already copied elsewhere although she had not. In that case, a simple *copy* and *remove* plan ended up in the single catastrophic action *remove* since the copy part of the plan had failed without the user having realized it.

2.2. Not realizing an error has been made

When users of operating systems give a command that does not serve their intentions well, this does not necessarily mean that an error message would be produced by the operating system to alert them. This often happens when a perfectly acceptable command

or action has been issued by the user. Operating systems do not normally provide reasoning about whole sequences of commands or actions of users in terms of their possible plans and goals. Hence, a sequence of syntactically correct actions may represent an erroneous plan in terms of a user's goal. For example, a user was observed accidentally creating a file with a very similar name to an existing file through a 'copy' command which had a typing error; the user had typed the syntactically correct command 'cp fred perquish' instead of 'cp fred perqish', where 'perqish' was mistyped as 'perquish'. This led the user to problematic situations that she would only realize after a considerable amount of time, when it was really difficult for her to figure out what had happened.

2.3. Considerable effort wasted in the rectification of errors

Naturally, there are simpler cases where errors are not catastrophic and users realize that they have made an error soon after this was made. However, even in these cases, users have to formulate and carry out a plan for the rectification of the error made. In cases like this, users have often been observed wasting considerable effort trying to recover from a small error and getting involved in new ones resulting to their frustration and inefficient use of the operating system.

2.4. Problematic situations leading to errors

Users may not know how to achieve a goal while using an operating system. For example, a user may want to copy certain files from her/his hard disk to a diskette and s/he may not know how to do it. Users have frequently been observed getting on-line help and making a mistake immediately after that. This means that users may find it difficult to retrieve the right piece of information from the on-line manuals, probably because they need advice tailored to the individual circumstances in which their question or problem has occurred.

3. A survey of the literature of intelligent help systems

In this section, we discuss a lot of the general issues concerning IHSs and then we present three prototype IHSs that have been developed to assist users.

3.1. Discussion of general issues of IHSs

Intelligent help systems are meant to operate on top of the original user interface of a software package and provide on-line help. The main reason that stimulates research on this area in general, is simply the fact that the currently available user help consisting of on-line and off-line manual information, is insufficient. A user might waste an awful lot of time trying in vain to find the piece of information that s/he needs. Advice tailored to the user's knowledge of the domain and the particular circumstances that caused a question, is bound to be appreciated.

IHSs are usually classified into two main categories, the *passive* and the *active*. Passive systems behave like the expert who has been *called to offer advice*, whereas active systems behave like the 'over the shoulder' expert who watches the user and *offers spontaneous help*. In computer terms, passive systems can answer all sorts of questions relevant to the

domain area that the user may pose, whereas active systems monitor all user actions, reason about them and decide themselves whether to interrupt or not.

Intelligent help systems have an architecture that consists of a User Modelling component, an Advice Generation component and a representation of the Domain Knowledge. A fourth component is the interface between the user and the help system and is not always mentioned, although it has been the central focus for many help systems. The boundaries of the functionality of these components are not clear cut and vary considerably in different implementations of help systems, depending on the emphasis, the needs and the design decisions of the particular help systems.

Ideally, an IHS would consist of all four main components as well as being able to operate in both passive and active mode leaving the user with a complete choice and making the help system itself much more flexible.

3.1.1. The user modeller

The user modeller is concerned with how information about users can be acquired by automated systems and with how that information can be used to improve the system's performance [9]. It tries to infer all the information needed about the user's actions and possibly store useful long-term information about him/her. For example, it tries to infer the user's goals, plans, beliefs, the misconceptions (if any) underlying these beliefs and the more long-term information such as the extent of his/her knowledge of the domain, typing skills, common errors and many more issues that would help the system form an idea of the user and his/her actions. One important task of user modelling is *plan recognition*. A plan of a user consists of several actions that the user issues in order to achieve a goal. For example, in Windows 95, when a user issues a 'cut' action and then a 'paste' action, these two actions may be considered to constitute a plan of a user that serves his/her goal to move certain items from one place to another. The plan recognition mechanism usually has to overcome the problem that actions serving the same goal may not appear one immediately after the other. For example, the user may issue the 'cut' action, then have a look at his/her e-mail and then issue the 'paste' action.

There are many types of user model depending on where the emphasis has been placed. Rich [10] makes a distinction between long-term and short-term user models. A long-term user model may consist of all past user information that has been stored by the developer of the help system or has been acquired and stored by the help system itself. For example, information about the user's level of knowledge of the domain, or style in the use of the domain can be considered to belong in the long-term user model. A short-term user model consists of the user's beliefs at a very specific time (e.g. before s/he issues command X) and is the output of some reasoning that combines pieces from several sources of information, such as the long-term user model, domain knowledge, the current user's actions and the current state of the environment (e.g. screen, file store etc.).

3.1.2. The advice generator

The user modeller is doing all the understanding about a user and the advice generator is needed for the acting. In terms of plans, it is *plan generation* this time. For example, the user is known to have a certain goal in mind; what is the best way to achieve it? The help system should be able to suggest an alternative plan to a user's inefficient one or just

suggest one to a user who does not know how to do something and is explicitly asking for help. Besides, help systems have to decide what sort of advice they are going to give and how. For example they have to decide how much information to give to the user so as to avoid presenting issues that the user already knows and say no less than the user needs. This is when the advice generator has to seek information in user models.

A classification that we have made of IHSs [11], depends on the kind of advice that they are aiming at. According to this classification we distinguish between 'trouble-shooting' and 'coaching' help systems. *Trouble-shooting* systems help users who have experienced a problem while applying previously acquired knowledge, whereas the *coaching* ones help users improve and expand their knowledge.

Trouble-shooting systems mainly try to diagnose the cause of the problem that the user encountered at a specific situation and offer remediation. The cause could be a user's misconception, a misinterpretation of an error message, a wrong belief about the current state, lack of a particular piece of domain knowledge etc. The more failure causes that the system is able to diagnose, the more powerful it becomes. Active systems seem closer to the trouble-shooting aspect because of the plan recognition and user monitoring that they involve. However, this is not always true. For example, active systems that are only concerned with the optimization of user's codes could be classified to the coaching ones because their aim is to improve and expand the user knowledge rather than help with a particular problem. Similarly, passive systems seem closer to coaching systems because of their ability to answer questions of how to do this or what is that. Again this is not always true. For example, a passive system that answers questions by giving an explanation to a user who describes an awkward situation, should be considered as a trouble-shooting system.

3.2. Help systems

In the remainder of this section, we present three help systems that have been developed to assist users of operating systems. The help systems selected to be presented here, are particularly concerned with UNIX. In this way the uniformity of the domain may help the user to gain a better insight to the issues concerning IHSs.

First is UC, which is a prototypical passive UNIX help system. Then AQUA, which is a passive and trouble-shooting system. Finally, the EUROHELP system which is both passive and active as well as both trouble-shooting and coaching.

A summary of the classification of the UNIX help systems reviewed, is illustrated in Table 1.

3.2.1. The UNIX consultant (UC)

This is a prototypical passive help system, developed at Berkeley [12–16]. UC aims at helping the user in the coaching way by answering his/her questions. It can also handle simple forms of dialogue. For example,

USER: How do I delete a file?
UC: Typing 'rm filename' will remove the file with name *filename* from your current directory.
USER: a directory?

Table 1
Classification of three existing UNIX help systems

	Active		Passive	
Systems	Coaching	Trouble-shooting	Coaching	Trouble-shooting
UC			·	
AQUA				·
EUROHELP	·	·	·	·

UC: Typing ‘rm -r directory-name’ will recursively delete the directory with name directory-name and all the files and directories contained in the directory.

Another example could be:

USER: What is the best way to create a file?
UC: The best way to create a file is to use an editor, e.g. ‘vi filename’.

A lot of research effort for this system has been put into the natural language component. Facts about UNIX, the English language and the world are represented in Knowledge Bases as frame-like structures. Its knowledge bases are extensible allowing it to learn effortlessly more about new UNIX skills and English.

The *natural language component* is split into two other components. One component is a language analyser, which analyses the user’s question and the other gives the advice in English. UC infers the user’s intentions from the question. It can also reason about the question and see whether the user meant something else or just determine what the focus of attention of the question is. All this concerns the understanding of the user’s question despite any ambiguities of the expressions that s/he used.

The *advice generator* consists of a plan generation mechanism and an expression formation mechanism. Its function is based on a plan library. Among the tasks of the plan formation is to select one goal among conflicting ones. For example, if the user asked the system to tell him/her how to get more disk space, the system could advise the deletion of all the user’s files. However, this conflicts with another long-term goal of the user preserving his/her files. The Goal Projector and the Goal Detector are meant to handle this. The Goal Projector simulates the situation that would occur, should a goal be pursued. In the above example, it shows what would happen if the goal ‘remove your files’ was adopted. The Goal Detector then is applied to the simulated environment and infers the goal of preserving the files. Finally, an expression formation mechanism is used to give the user the piece of information that s/he seeks and no more than that.

UC responds only to the user’s request, as all passive systems are meant to work. This saves a lot of computational effort, but does not save the user from catastrophic errors that s/he might accidentally get her/himself into. In addition, the information used about the user is general and does not take into account her/his actions. Therefore help is not tailored to the particular circumstances that made the user ask for help. This could be destructive sometimes as there are cases where users might be seeking information that they think they need while in reality this is absolutely against their overall plan. A very illustrative example of this situation was a session observed in a real interaction of a user with UNIX.

In that case the user thought that an old directory called 'tf' had been copied in a new directory called 'poplog/tf'. In fact, although this UNIX shell did not complain, the directory contents had not been copied. However, the user did not get any error message and therefore proceeded to the next goal of removing the 'copied' directory. The user did not know how to do this either, but was aware of that this time. If UC was available, this user might have asked how to remove the 'unwanted' directory and UC would have replied without any objection, but the user would not thank UC for that.

UC is very good at helping the user find out how to achieve certain goals and thus make fewer mistakes in the planning method that they achieve. However, the monitoring of users' actions which would be useful to users is beyond the scope of this help system.

3.2.2. AQUA

AQUA is a *trouble-shooting*, passive help system for UNIX [17,18].

It aims at handling dialogues with the user, such as:

1. *User*: I tried to remove a file with the 'rm' command. But the file was not removed and the error message was 'permission denied'. I checked and I own the file. What's wrong?
Advisor: To remove a file, you need to be able to write into the directory containing it. You do not need to own the file.
2. *User*: How do I make the directory writeable?
Advisor: To make a directory writeable, use 'chmod + w' command.
3. *User*: Wait while I try it. No, 'chmod + w' prints the error message 'permission denied'.
Advisor: To make a directory writeable, you need to own it.
4. *User*: So, to remove a file, I have to own the directory that contains it.
Advisor: No, to remove a file, you need to have write permission on the directory that contains it. You do not need to own the directory that contains it. You need to own that directory when you do not already have write permission on it.
5. *User*: So, how do I remove the file?
Advisor: Send mail to whoever has write permission on the directory, asking him to remove the file for you.

In this sort of dialogue, the user explicitly states what his/her situation was when the problem arose. That means that the user has to state his/her goal, planning method for achieving it and the UNIX response to it. In return s/he gets from the help system replies that highlight any misconceptions and gaps of knowledge that led him/her to the awkward situation. In the above example, the advisor detected two user misconceptions:

1. An incorrect belief that owning a file is a precondition to removing it.
2. A subsequent incorrect belief that the precondition is owning the directory in which it resides.

It also detects three gaps in the user's knowledge:

1. The user has no plan for making a directory writeable.
2. Does not know why the advisor's plan for doing so failed.
3. Has no plan for removing a file that is not in a writeable directory.

Actually, the focus of attention of this system is the detection of the user planning misconceptions.

The system has an *advice generation mechanism*, whose input is an initial short-term user model consisting of a set of inferred user's beliefs and output a set of advisor's beliefs that contradict those of the user. The set of advisor's beliefs is the explanation that highlights the user's planning misconception. Both its input and output could be viewed as part of the short-term user model, in the sense that they are concerned with what the user believes. However, its input is supposed to be a set of user's beliefs, whereas the output is a set of advisor beliefs about the user's beliefs and therefore the whole computation has been included in the advice generator.

The advisor expertise is a large set of advisor beliefs. A subset of the advisor's knowledge is the usual planning library. It consists of plans (sets of planning relations) associated with planning failures (goals to which a plan does not apply). Planning failures are regarded as 'potential explanations' and are classified according to the planning relation that they are attached to. A potential explanation is verified if all the planning relations it consists of are proved to hold. Actually, the overall aim of the system is to *find an explanation* to present to the user in case of a failure.

For example, an explanation for the first misconception of the user in the above example was inferred using the rule: if the user tries to verify a state *S* (this case: owning a file) and both *S* and a known precondition (this case: having write permission) are instances of a more general state (this case: having sufficient permission) then assume that *S* is a precondition of the user's goal.

Unlike UC, the *natural language* component has not been a major issue for AQUA. The focus of this system is on planning misconceptions and addresses the problem of diagnosis more deeply, than other help systems. It does not address other kinds of errors, like typographic errors.

3.2.3. The EUROHELP system

The EUROHELP system was developed in the context of a very large ESPRIT research project (about 100 man-years over 5 years), which included several researchers from several countries [19–22]. The original aim was the construction of a shell meant to be used for all help systems dealing with Information Processing Systems (IPS). However, the techniques specified were applied on UNIX mail first. It is both passive and active as well as both trouble-shooting and coaching. In principle, it addresses most of the issues discussed in the previous sections.

The passive component can answer seven general classes of enquiry in natural language, such as 'What is a folder?' or 'What do I do next?'. The answers to these questions are formed taking into account the current working context and the user's knowledge and intentions. The active component is monitoring a user performance aiming at recognizing inefficiencies in his/her plans as well as diagnosing any possible errors. In this system the *user modeller* is working for both the active and the passive mode, using different components for each of them.

For the *active mode* there is a *Performance Interpreter* consisting of a *Plan Recognizer* and a *Planner*. The performance interpreter evaluates the user's action by considering four

anomaly criteria:

1. Whether the action was legal, i.e. it could be executed.
2. Whether the action was useful, i.e. it led to some change of state.
3. Whether the latest action could be incorporated into a plan.
4. Whether that plan was optimum as far as the user was concerned.

The plan recognizer collaborates with the planner in order to guide its search for suitable plans. The plan recognizer uses a hierarchical database of known tasks and plans for them. The first user action identifies which plans it could be the first action of and one of these is selected as a hypothesis of the user's plan. If subsequent actions also fit in this plan they are considered explained. If a subsequent action does not fit in with the hypothesized plan then the plan recognition system backtracks and reconsiders alternative plans. One difficulty involved with this plan recognition strategy is the search involved.

The *Diagnoser* is activated when attempting to explain a potential problem in terms of lack of knowledge or misconception. In the *case of lack of knowledge*, it is generating hypotheses by taking some knowledge out of the domain representation. In a *perturbation case* it is perturbing concepts. The search space is controlled by the use of predefined knowledge about general misconceptions and also by the use of several heuristics such as parsimony of misconceptions in one single case, lower complexity of replacing concepts, extent of user knowledge etc. The diagnoser is also used for the *passive mode*. In that case the question is first passed to the Question Interpreter which defines the user's need within the context of the performance.

The *advice generator* is taking input from the diagnoser and tries to define a 'tactic structure' according to the way the problem was caused (question or performance) and the diagnosis of the problem itself. There is a special module that transforms the answer into *natural language* or some other formalism.

However, predefined knowledge about general misconceptions, which is the main heuristic of EUROHELP, can prove inadequate when an unknown case comes up. The ability of reasoning about misconceptions themselves is quite limited.

4. RESCUER, an IHS that reasons about plausible errors

4.1. Main focus of RESCUER

RESCUER operates in a trouble-shooting mode, offering active assistance to users who may be involved in problematic situations. Unlike UC, RESCUER does not give answers to explicit questions of users on how to achieve certain goals but rather it monitors users' actions and reasons silently about them until a problem is diagnosed. AQUA also operates in a trouble-shooting mode, like RESCUER, in the sense that it tries to give explanations to the user on what may have been wrong. However, AQUA generates advice which is based on the users' explicit description of the problem that s/he encountered. Both UC and AQUA offer help based on user's explicit questions and on the user's initiative. However, in cases where the user does not realize that s/he needs help during the interaction both

systems would not be able to offer help. In contrast, RESCUER aims at providing such context-sensitive on-line help, on its own initiative.

This type of behaviour was also included in EUROHELP, which was meant to address most issues concerning an IHS. However, EUROHELP was a very large project where many researchers were involved for many years and yet some parts were never implemented. This means that a project like this may be very difficult to reproduce for other domains of user interfaces, although the wealth of ideas presented in this project are certainly useful for this purpose. In any case, a common criticism of the application of Artificial Intelligence to computer-aided instruction, is what McGraw notes [23] about the resulting products that they may miss the mark in terms of task reality, feasibility and effectiveness although they are usually superior to traditional standup products. This criticism may also apply to EUROHELP. However, McGraw continues saying that user interface developers may want to consider the feasibility and benefits of using AI to enhance the user interface. This is precisely one of the main aims of the design of RESCUER, namely to enhance the user interface performance in terms of the recognition of user errors without missing the mark in terms of feasibility. Therefore, RESCUER focuses on providing help to 'plausible' user errors rather than any possible error. A user error is considered 'plausible' if multiple sources of evidence show that this error may have been made. When RESCUER 'thinks' that a user has made an error with respect to his/her hypothesized intentions, it intervenes and offers advice.

For example, suppose a UNIX user types the following sequence of commands:

1. % mkdir programs
2. % cp program1.pas programs
3. % cp program2.pas program

where in command 1 a user makes a directory called 'programs', in command 2 this user copies the file 'program1.pas' into the directory 'programs' and in command 3 the user literally copies the file 'program2.pas' into a newly created file called 'program'.

Most human observers would think that command 3 would be meant to read:

% cp program2.pas programs

which would copy 'program2.pas' into the directory 'programs', just like 'program1.pas' was copied into this directory.

The reaction of a human observer would probably be to let the user know about the possible error, although the user may not have made a mistake. This is a case that is not simply the correction of a typographic error but is the result of combining evidence that shows there may have been a typographic error. The whole context suggests that there has been an error, whereas if somebody looked at command 3 alone, it would probably look correct. If a computer-advisor intervened and suggested the correction of command 3, given its particular context, this intervention would look natural to human users. This is the kind of behaviour that RESCUER aims to achieve.

The evidence that RESCUER takes into account comes from the following sources:

1. Certain peculiarity criteria that show that a given action may have been problematic (e.g. when a command has failed to execute).

2. A mechanism for generating alternative hypotheses of what the user may have meant. This mechanism applies the Human Plausible Reasoning (HPR) so that the hypothetical actions generated are similar to the action typed and also allow for 'plausible' errors to have been made.
3. A plan recognition mechanism which focuses on the sequences of the effects of commands rather than sequences of commands themselves.

The combination of these three sources of evidence agreeing that a certain action may have been erroneous makes RESCUER react to offer advice, otherwise RESCUER does not react. It may be the case that RESCUER does not react to situations where there has in fact been a problem. This is because the focus of its attention is in cases that would look 'obvious' to a human observer that there has been an error. Even in cases like this a user interface would require intelligence to spot the error. If these errors are recognized then much of the frustration of users can be alleviated and a lot of catastrophic errors can be prevented.

In the remainder of this section we introduce Human Plausible Reasoning theory so that the reader can have an idea of the background used in RESCUER and then we present the overall design of RESCUER.

4.2. Human plausible reasoning theory

Human Plausible Reasoning Theory [7,24,25], has been constructed by Collins and Michalski in an attempt to formalize several patterns that people use in order to answer questions, when they do not know the exact answer. It is assumed that people have a patchy knowledge of certain domains such as geography or biology. However, if people are asked a question about something that they do not know, they will try to conclude the answer from what they already know and consider relevant.

An example of patchy knowledge that one person may have, is illustrated in the experimental Matrix of Biologic Data in Table 2. The places with the question mark denote lack of knowledge of the person. In this case, the person does not know how a whale breathes and whether it is a fish or a mammal, but knows that it lives in the water and that it is large. The person also has a complete record of information about a shark and an elephant, i.e. that a shark is a fish, breathes through gills, lives in the water and is large and so on.

Suppose a human subject has this patchy knowledge about the animals mentioned and is asked to give some answer about something that s/he does not know. For example, how a whale breathes. The corresponding entry in the experimental matrix has a question mark,

Table 2
Experimental matrix of biologic data

Class	Isa	Breath	Lives	Size
Mammal	Animal	Lungs	Water, land	Large, medium, small
Fish	Animal	Gills	Water	Large, medium, small
Whale	?	?	Water	Large
Shark	Fish	Gills	Water	Large
Elephant	Mammal	Lungs	Land	Large

which means that the person has no direct knowledge to answer the question. In this case the person will use the knowledge which s/he considers relevant, from other places of the matrix.

For example, the person may come up with the following reasoning. The way an animal breathes is relevant to the place it lives and to the structure of its body and because a shark is similar to a whale with respect to these factors then probably the whale breathes in the same way as the shark, namely through gills.

This is a plausible guess although incorrect in this case. HPR is not concerned with misconceptions or incorrect beliefs mainly because it focuses on the inferential process that leads people to derive a belief based on another belief, irrespective of the correctness of either beliefs. The HPR theory gives a way to formalize this kind of plausible reasoning. Facts are expressed in the so-called 'statements' and inferences are drawn from the 'statement transforms' or the 'mutual implications and/or dependencies'. A brief outline of these is given in the section that follows.

4.2.1. Terminology

The matrix presented in Table 2 is an example of the knowledge of a person in a certain domain and can be formally expressed in a collection of 'statements' that represent the person's beliefs. For example a statement has the following structure:

breath(shark) = gills,

where 'breath' is called a 'descriptor'. This descriptor is applied to 'shark' which is called an 'argument'. The relation of a descriptor applied to an argument is called a 'term'. A term can take values which are called 'referents'. In this case, the referent of the term 'breath(shark)' is 'gills'. In the matrix of the example we see that the person may have gaps of knowledge, such as breath(whale) = ? which means that s/he does not know how a whale breathes.

A statement can be regarded as an object-attribute-value triple where the object is an argument, the attribute is a descriptor and the value is a referent together with a set of certainty parameters.

The simplest class of inference is called *statement transforms*. If a person believes the statement about the way a shark breathes, i.e. breath(shark) = gills, there are eight statement transforms which allow plausible conclusions to be drawn. The *argument transforms* move up, down or sideways in the argument hierarchy, using GEN, SPEC, SIM or DIS, respectively. The *referent transforms* do the same in the referent hierarchy.

For example, the person knowing the above statement can draw conclusions like the following:

Argument transforms

1. *breath(fish) = gills*, through a generalization argument transform. This is done because fish is a generalization of a shark in an animal hierarchy.
2. *breath(white-shark) = gills*, through a specialization argument transform. This is done because a white shark is a special case of a shark in an animal hierarchy.
3. *breath(whale) = gills*, through a similarity argument transform. This is done because a whale can be found similar to a shark.

4. *breath(wasp)(gills)*, through a dissimilarity argument transform. This is done because a wasp is found dissimilar to a shark.

Referent transforms

1. *breath(shark)* = *internal-organ*, through a *generalization* referent transform. This is done because an internal organ is a generalization of gills in a hierarchy about the parts of an animal body.
2. *breath(shark)* = *gills* of a big fish, through a *specialization* referent transform. This is done because the gills of a big fish is a special case of gills with respect to some features, such as size and so on.
3. *breath(shark)* = *lungs*, through a *similarity* referent transform. This is done because lungs can be found similar to gills in a hierarchy of animal organs.
4. *breath(shark)* ≠ *ear*, through a *dissimilarity* referent transform. This is done because ears are found dissimilar to gills in a hierarchy of animal organs.

4.2.2. A notation for argument and referent transforms

Argument transforms are quite straightforward in the way that they answer questions in the matrix. For example, we have 'breath-organ(whale) = ?'. An argument transform would involve changing the argument 'whale' with another argument from the matrix, such that:

1. Its breath-organ is known.
2. The argument itself is similar to a whale with respect to some descriptor that is relevant to 'breath-organ'.

For example, 'shark' could replace the argument 'whale' because:

1. 'breath-organ(shark) = gills'.
2. 'shark' is similar to 'whale' with respect to where they live (water) and their size (large) and these have some dependence on how an animal breathes.

The notation that we have used for this example is illustrated in Table 3. The two arrows are used to show how the transform works to answer a question. We start from the statement bearing the question, then apply an argument transform to find a known statement and then use the referent of the known statement to replace the question mark.

Table 3
A notation for argument transforms to answer questions

Argument transform			
First statement:	$d(a)$	=	?
Arg. transform	↓		↑
Transformed statement:	$d(a')$	=	r'
An example			
First statement:	breath-organ(whale)	=	?
	Arg. transform ↓		↑
Transformed statement:	breath-organ(shark)	=	gills

Table 4
A notation for referent transforms to answer questions

Referent transform			
First statement:	$d(a)$	=	?
Ref. transform			↑
Transformed statement:	$d(a)$	=	r'
An example			
First statement:	breath-organ(whale)	=	?
	Ref. transform		↑
Transformed statement:	breath-organ(whale)	=	gills

Referent transforms are not so straightforward in the way they can answer questions in the matrix. The reason for this is that the statement which we start from is the statement with the unknown referent. However, if the referent is completely unknown to the person then it cannot be transformed. A solution to this problem would be to assume that the person whose knowledge is being modelled, has some preconception of what the unknown referent should be and uses this preconception to come to a final conclusion. For example, the person may know that a whale breathes through an internal breathing organ and tries to find what this organ is. In this case, the person would also have to know what possible breathing organs may be.

Therefore, our interpretation involves some knowledge of the person on the possible values of the referent. In the example, we may have to assume that the person knows that the known statement 'breath-organ(whale) = internal breathing organ' may lead to 'breath-organ(whale) = gills' through a referent transform.

The notation that we have used for referents (illustrated in Table 4) will only show the statement used for the replacement of the question mark by some answer.

The hierarchy that is assumed to underlie the referent transforms of the example, is illustrated in Fig. 1.

4.3. Overall design of RESCUER

The overall performance of RESCUER is outlined as follows: RESCUER's input is the command typed by the user which is evaluated against certain peculiarity criteria that are used to alert RESCUER as to whether the command needs further examination. These criteria include the questions whether the command was acceptable to UNIX or not, whether it was expected by RESCUER, in terms of the hypothesized users' intentions, whether it was a command frequently used by most users, and so on, and are only used as unconfirmed symptoms of a possible problem. Whether there has in fact been a problem has to be confirmed and explained by the diagnostic part of RESCUER.

The diagnosis of a problem is done by the User Modeller. If RESCUER can find an alternative interpretation about the user's command such that it is similar to the interpretation that UNIX has given but is better than that in terms of RESCUER's evaluation criteria, then RESCUER may have to adopt this alternative interpretation. For example, if the command typed by the user has failed to do anything at all, or if it was not expected in terms of the plan recognition scheme then the User Modeller will generate hypotheses

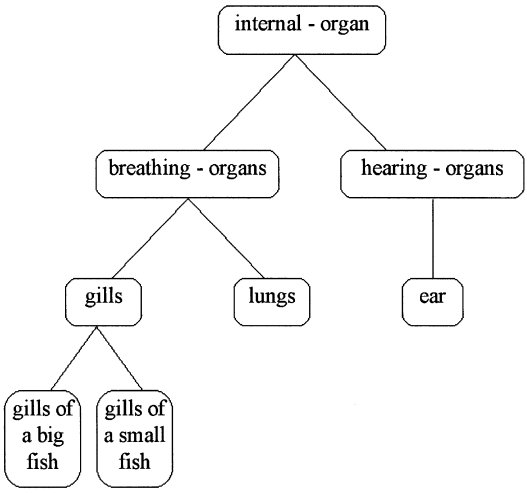

Fig. 1. A hierarchy of internal organs of an animal.

about possible interpretations of what the user's beliefs may have been. The hypotheses will be generated using HPR transforms as will be explained in detail in the following section.

Plan recognition is achieved by considering the domain (here the UNIX file store) as an entity that has certain properties, one of which is called 'instability'. The existence of instabilities implies a possible future transition of the current state of the domain to another state which is expected to follow. In this way, RESCUER overcomes the problem of having to recognize particular sequences of actions as part of a user's plan.

For example, if a user creates a new directory (or folder in other domains) then RESCUER attaches an instability to the domain. The attached instability can be removed if the user issues an action that gives contents to this directory (e.g. it copies or creates one file into the new directory). The particular actions that the user is going to issue do not play an important role. The focus of attention is on the effects of actions to the domain. Expectations in terms of effects to the domain are used to guide the search about possible corrections of commands.

The process of the generation of an alternative interpretation contains all the information needed for the construction of an explanation of what happened. The generation of response, if any, is done by the Advice Generator. If RESCUER has confirmed its initial suspicions about the existence of a problem then it will form two kinds of response:

1. A suggestion of an alternative command which would fit better in the context.
2. An explanation of what happened which includes presenting what misconception at what conceptual level was to blame for the problem observed.

Instabilities are used to indicate that a user may have started some plan. An unstable file store does not mean that the user has made an error but that perhaps s/he is in the middle of a plan. The notion of instability could be extended to other domains as well where there are sequences of actions. For example, in the domain of somebody's life, if this person

registers for a degree then s/he creates an instability in her/his life. The instability could be removed if this person graduates or if s/he drops the study. Registering for another degree (i.e. introducing a new instability while the previous one is not removed) could be considered as an action which might need attention in terms of the person's plans.

The overall architecture of RESCUER is shown in Fig. 2. An example of RESCUER's performance in monitoring a UNIX user could be the following:

Supposing the user has typed:

%cp fredd directory1

which copies 'fredd' into 'directory1'.

Then after a few commands, none of which has to do with 'fredd' or its duplicate which resides in 'directory1', the user types:

%rm fred

which aims at removing a file called 'fred'.

At this point RESCUER offers some suggestion to the user:

RESCUER: Did you mean to type:
'rm fredd' instead of 'rm fred'?

This suggestion would be offered to the user irrespective of whether 'fred' existed or not which means that the suggestion would be offered irrespective of whether there has been a UNIX error message or not.

The reason why RESCUER would be alerted would be because the command 'rm' is a destructive command and therefore it attracts RESCUER's attention. RESCUER would expect 'fredd' to be removed instead of 'fred' because first the two names are very similar (therefore there could have been a mistake) and second 'rm fredd' could be considered to

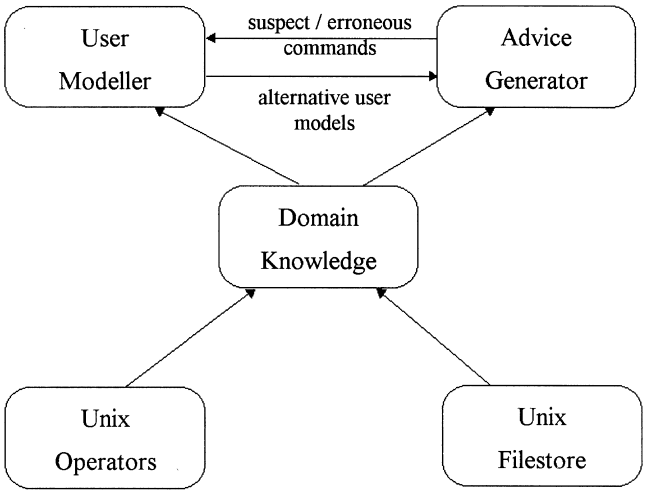

Fig. 2. Overall architecture of RESCUER.

complete a plan of ‘copy and remove’ whereas ‘rm fred’ is a destructive command that is not connected to the previous commands.

If the user had not issued the command ‘cp fredd directory1’ then RESCUER would not have responded with a suggestion to the user to remove the file ‘fredd’ unless the mistyped ‘fred’ did not exist in the file store, in which case there would have been an error message.

From the above example, it is clear that RESCUER combines evidence from more than one sources and addresses situations where there may have been a mistake irrespective of whether this has produced an error message by the operating system. It also addresses situations where an action may have been used constructively by a user although it may have produced an error message as well. This can to happen in cases where an action has more than one effect, one of which may fail and produce an error message but the rest of them may work.

5. Human plausible reasoning theory in automatic error diagnosis

The HPR theory can be directly applied to situations where a person is asked a question, such as ‘how does a whale breathe?’. However, an active help system like RESCUER, is supposed to monitor a user without asking him/her explicit questions about the domain. Therefore, the application of the theory to the user modelling component of RESCUER is made on the assumption that users ask themselves certain questions about the commands of the operating system, in their effort to form and issue a command (or action) that would be accepted by the operating system and will serve their intentions well. The chain of such questions constitutes a path of reasoning that is assumed to have led the user into concluding what command to type.

For example, in the case of the UNIX domain, the user is assumed to have assessed the command typed before typing it, in terms of whether it would be acceptable by UNIX. The user must have come to the conclusion that the command would be acceptable and then typed it. This basic assumption about the user relates to his/her intention to issue a command that would not fail to execute and therefore would be acceptable by UNIX. The assessment of the command by the user is called basic principle and it represents the main assumption of RESCUER about the user. The basic principle consists of two questions that the user is assumed to have asked her/himself in her/his effort to check whether the command s/he has selected to type is correct. The two questions are the following:

What is the semantic and syntactic structure of the command?
Is this semantic and syntactic structure acceptable to UNIX?

The user is assumed to have used the answer of the first question in the second question. The user is also assumed to have given the answer ‘yes’ to the second question before typing a command.

The two questions have been expressed as two ‘statements’ using the representation language of HPR. The fact that the answer of the first question is used in the formation of the second question leads to the use of what is called a ‘multiple descriptor statement’ in HPR terminology, which represents a chain of reasoning that a person may have used. In order to express the two questions in HPR statements we have used two descriptors,

namely ‘UNIX-acceptable’ and ‘intended-pattern’. Each descriptor is applied to an argument.

The argument of ‘intended-pattern’ is ‘action’. The descriptor ‘intended-pattern’ is used to mean a simplified form of what the user must have believed that the preconditions of the command were. The argument ‘action’ is used to mean the command line that the user has typed. For example, if there is a file called ‘fred’ in the user’s file store and the user has issued the command:

rm fred

then ‘rm fred’ is the ‘action’ and the ‘intended-pattern’ of this action is the referent ‘rm file’ which conveys the semantic and syntactic information about the action; ‘rm’ has taken one argument[1], which is a file. In the HPR representation language we have the statement:

intended-pattern(action) = rm file.

This statement corresponds to the question: ‘What is the semantic and syntactic structure of the command?’. The answer to this question can either be the *typed pattern* or a *similar pattern*.

The second statement is formed by replacing the ‘intended-pattern(rm fred)’ with the referent which in this case is ‘rm file’:

UNIX-acceptable(intended-pattern(rm fred)) = UNIX-acceptable(rm file).

The statement UNIX-acceptable(rm file) refers to the second question that the user is assumed to have asked himself or herself: is this semantic and syntactic structure acceptable to UNIX?

The answer that the user is assumed to have given to this question is ‘yes’ unless s/he is not sure about the correctness of the command that s/he issues.

The referent in the first statement refers to the instantiated semantics of the action typed, whereas argument in the second statement refers to the semantics of the command involved. In some cases the instantiated semantics may be different from the accepted semantics of the command. For example, if the user had typed the command

rmdir fred

then the instantiated semantics would be (rmdir file) whereas the semantics accepted for this command would be (rmdir directory).

The set of HPR transformations are applied to statements constituting the basic principle and generate different possible interpretations of how a user may have come to the conclusion that the command s/he typed was acceptable to UNIX. These interpretations reveal the possible misconceptions involved.

The basic principle expresses a fundamental assumption about the user, that the user believes that the semantics of the command s/he intends to type is legal as far as UNIX is concerned. The name ‘basic principle’ is justified by the fact that it is used as the main source for the generation of hypotheses about possible misconceptions.

[1] ‘Argument’ here, means the UNIX argument and not the ‘argument’ used in the HPR terminology.

Errors and misconceptions of all kinds, varying from deep conceptual confusions to accidental slips, are treated as gaps in the user's knowledge or skill that became filled with something similar to what was intended or correct. In terms of HPR, certain transformations have taken place that have led the user to the incorrect or unexpected filling of the gap. Depending on where the transformation takes place, we have the conceptual level of the error and depending on the kind of the transform we have the cause of the misconception.

5.1. The user modelling task

RESCUER's User Modeller constructs an individual, dynamic, short-term user model. A user model in this paper is an account of the user's correct or incorrect beliefs about the domain. By definition, a user model like this is impossible to construct with absolute certainty because we do not have access to the user's mind. Therefore, more accurately, a user model is the system's hypothesis about the user's beliefs. In this text we will usually omit the phrase 'the system's hypothesis about' and just refer directly to 'the user's beliefs'.

In particular, a user model just before a command has been issued, consists of the following correct or incorrect beliefs that RESCUER believes that the user has:

1. The user's beliefs about the file store.
2. The command intended by the user.
3. The user's beliefs about the semantics of the command intended.
4. The user's goal that the command intended was meant to achieve.

RESCUER has to have a hypothesis about the first three at all times when a command has been issued. If the command is not considered suspect at all then these three have the default values of the UNIX interpretation:

1. The configuration of the file store that UNIX holds in memory.
2. The command typed.
3. The standard UNIX semantics of the command typed.

All three beliefs are involved in the multiple descriptor statement that constitutes the basic principle. The command typed is the argument of the first statement. The configuration of the file store that UNIX holds in memory is used to form the instantiated semantics of the command typed which is the referent of the first statement. Finally, the standard UNIX semantics of the command typed is the argument of the second statement.

The actual compilation and execution of a user's command can be viewed as UNIX's 'interpretation' of this command. This interpretation does not necessarily account for what the user believes about the command that s/he typed and does not necessarily account for what RESCUER believes about the user's beliefs. RESCUER uses the HPR transforms to alter this default UNIX interpretation at any time that it believes that the user may have different beliefs.

For every action of the user, RESCUER instantiates the semantics of the action and computes the typed internal pattern that corresponds to the instantiated semantics. The typed internal pattern serves as the default hypothesis about what the intended internal

Table 5
No transforms in the basic principle

(1) Action having no obvious problem	
First statement	A fact: intended-pattern(action) = typed-pattern
Second statement	A fact: UNIX-acceptable(typed-pattern) = yes
Transforms	None
Meaning	Apparently the action intended was the action typed, unless there is evidence to the contrary.
Misconception	Apparently none.

pattern is. If RESCUER is alerted about a command then the default hypothesis will be questioned. For example, if a command fails to execute then RESCUER will be alerted and will question the default hypothesis by generating transformed hypotheses.

5.2. Hypotheses about user's misconceptions

The chain of reasoning that the help system uses to generate hypotheses about possible user's misconceptions which are hidden in the action typed, is contained in the statement transforms of the basic principle which was explained above.

Each of the transforms has a different meaning in terms of what the user's misconception or error may have been. The whole lot of transforms provide a satisfactory range of hypotheses about errors that the system has to examine in order to select the most appropriate to present to the user. For the purposes of the illustration of some examples we will assume that a user has typed the incorrect command: % rm tf, where tf is an empty

Table 6
Argument transform in the first statement of the basic principle

(2) Another action was intended	
First statement	intended-pattern(action) = ? Arg.transform ↓ ↑
Transformed statement	intended-pattern(action′) = close-known-pattern A fact:
Second statement	UNIX-acceptable(close-known-pattern) = yes
Meaning	The action typed was not fully intended. The action intended was different from the action typed.
Misconception	Superficial misconception involving accidental slips: Different arguments or commands were meant to be used. Any of the following could have happened: 1. Typographic errors in either the command or the arguments may have been involved. 2. The arguments or the command typed may have been used in previous actions of the session resulting in their accidental use in the present action. 3. The arguments typed may belong to a different directory from the current working directory.

Table 7
Referent transform in the first statement of the basic principle

(3) A UNIX-acceptable internal pattern was intended	
First statement	intended-pattern(action) = ?
	Ref. transform ↑
Transformed statement	intended-pattern(action) = close-known-pattern
	A fact:
Second statement	UNIX-acceptable(close-known-pattern) = yes
Meaning	*The action typed was fully intended; but the internal pattern was not.*
	The user thought that a different internal pattern corresponded to the action typed from the one that actually did.
	The pattern the user had in mind was UNIX-acceptable.
Misconception	*Rather superficial misconception involving memory slips*:
	Misconception about the UNIX file store state.
	The types of the arguments of the action issued were thought to be different from what they actually were.

directory. This means that the *typed internal pattern* is rm empty-dir. The command is incorrect because 'rm' does not remove directories but files. We will also assume that the user has a file called tff in her file store.

Since the user has typed an incorrect command, RESCUER will try to diagnose the cause of the problem by generating hypotheses about errors. The generation of hypotheses is based on statement transforms of the basic principle which are illustrated in the following five cases.

1. *Action having no problem.* In this case there are no transforms involved. What the user typed was correct as far as UNIX was concerned (see Table 5). This case does not apply to our example.
2. *Another action was intended*. This case (see Table 6), is characterized by an *argument transform in the first statement*. The transform results in the replacement of the action typed by another action, which means that a similar but different action was meant to be typed. For example, the user who typed % rm tf may have meant to type % rm tff which is a different action. The action intended would produce a UNIX-acceptable internal pattern: rm file.
3. *A UNIX-acceptable internal pattern was intended.* This case (see Table 7), is characterized by a referent transform in the first statement. It is assumed that the user intended a similar pattern to the one typed. In this case the action intended was the action issued but the user thought that a different internal pattern corresponded to this action. However, that the pattern the user had in mind was UNIX-acceptable. The misconception involved here would have to do with the types of the arguments[2] of the action typed. The user must have believed that the arguments had different types from what they actually had. For example, the user may have typed % rm tf only because s/he thought that tf was a file. If this had been true the command would have worked. In this case, the intended internal pattern would be rm file which is UNIX-acceptable, unlike the typed internal pattern.

[2] 'Arguments' here, mean the UNIX arguments and not the 'arguments' used in the HPR terminology.

Table 8
Argument transform in the second statement of the basic principle

(4) The typed action and its internal pattern were intended, but there was a misconception about the semantics.			
	A fact:		
First statement	intended-pattern(action) = typed-pattern		
Second statement	UNIX-acceptable(typed-pattern)	=	?
	Arg. transform ↓		↑
Transformed statement	UNIX-acceptable(close-known-pat)	=	yes
Meaning	*The action typed was fully intended and so was the typed internal pattern.* The user falsely concluded that the typed pattern was UNIX-acceptable.		
Misconception	*Deeper misconception involving conceptual confusions about the semantics of the command.* The user may have confused the semantics of the command issued with the command suggested by the close-known-pattern in the argument transform.		

4. *The typed action and its internal pattern were intended.* This case (see Table 8), is characterized by an argument transform in the second statement. This would mean that the action intended was the action issued and the internal pattern intended was the typed pattern which the user falsely concluded to have been UNIX-acceptable. In terms of the misconception involved, this case reveals a problem with the semantics of the command. The user may have confused the semantics of the command issued with that of another command. For example, the user may have typed % rm tf because s/he thought that rm could be used for removing directories as well as files. In this case, the intended internal pattern is rm empty-dir which the user may have confused with the rmdir empty-dir, which is correct.
5. *The typed action and internal pattern were intended but the user was not sure whether they were UNIX-acceptable or not.* This case (see Table 9), has to do with a *referent transform in the second statement*, i.e. the value ‘yes’ or ‘no’ as to whether the typed pattern was UNIX-acceptable or not. The answer ‘yes’ is not similar to ‘no’ therefore

Table 9
Referent transform in the second statement of the basic principle

(5) The typed action and its internal pattern were intended but user uncertain whether they were UNIX-acceptable			
	A fact:		
First statement	intended-pattern(action) = typed-pattern		
Second statement	UNIX-acceptable(typed-pattern)	=	?
	Ref. transform		↑
Transformed statement	UNIX-acceptable(typed-pattern)	=	yes
Meaning	The action intended was the action issued and the internal pattern intended was the typed pattern. The user was probably aware that this was either UNIX-acceptable or not (the two values of the referent being ‘yes’ or ‘no’) and decided to try and see whether UNIX would accept the pattern and the action.		
Misconception	*Doubts about the semantics of the command.* The user was not sure whether semantics of the command issued was UNIX-acceptable or not and decided to have a go and find out whether UNIX would complain.		

the explanation that one can give is that the user was doubtful about the command and decided to have a go and let UNIX complain if there was an error. This case reveals a problem with the semantics of the command.

5.3. Depth of misconceptions

The basic principle is used to generate hypotheses about misconceptions. The four different statement transforms can represent a cognitive classification of possible different misconceptions involved in the same kind of user error. As Hollnagel notes [26,27] there is a difference between the underlying cause of an error and the observable manifestation of the error and points out that it is wrong to mix the classification of observable phenomena with the interpretation of their causes.

The first two cases are characterized by transforms in the first statement. This means that the misconceptions involved are more superficial as can be seen by the errors that correspond to these cases (e.g. typos, misconceptions about types of objects etc.). As we go to the second statement, misconceptions become deeper because they have to do with the understanding of UNIX static domain knowledge such as the semantics of commands. The User Modeller of RESCUER favours the most superficial errors first and drops this hypothesis only if there is evidence that this does not apply to the particular case.

6. Evaluation of RESCUER

The role of this section is to demonstrate the evaluation of RESCUER's performance in terms of its primary aim which is to reproduce some 'plausible human reasoning' from an observer's point of view as to what a user may be doing and the kind of help s/he may need.

The idea is to show how well RESCUER could have followed the reasoning of users in sequences of commands which were entirely created by real UNIX-users who pursued their own plans and goals.

For this kind of evaluation of RESCUER's performance we have used sample scripts of real UNIX–user interactions, which were collected during the empirical study described in Section 2. The sessions which were recorded, constitute a sample of everyday interactions of users with UNIX.

Evaluating RESCUER's performance on samples of real UNIX–user interactions where users pursued their own goals, has the following advantages:

1. *Evaluating RESCUER by testing the plausibility of the whole notion of filestore instabilities* which has been used as a way of the system's keeping track of the user's plans and goals. It is interesting to see whether a real user does indeed tend to remove the filestore instabilities in a single session.
2. *Evaluating RESCUER's responses in terms of their correctness.* We can verify the success or not of RESCUER's suggestions by the very actions of the user as these are issued after the command that would trigger RESCUER's response.
3. *Evaluating the utility of RESCUER's successful responses.* The fact that RESCUER has not had real-time interaction with the user who produced the sample script means that

the user has not seen RESCUER's responses. This gives us the opportunity to evaluate the utility of the responses in terms of how much effort RESCUER could have saved the user if the user had seen and followed RESCUER's advice. The effort is estimated in terms of the number of commands that the user issued only to rectify an error that could have been rectified by RESCUER. Evidently, this test applies only to the cases where RESCUER has been successful at generating the correct hypothesis about the rectification of an error.

As an example of an evaluation of RESCUER on samples completely unknown to RESCUER, we comment on RESCUER's performance in a rather long session of an interaction of a user with UNIX. This session was long enough (73 commands) to bring up a variety of issues and errors that the user had made and also had the advantage of their all being contained in one session which showed clearly how many commands RESCUER was able to follow in a single session and how well it could associate related commands which may have been issued at very different stages of the same session.

6.1. Statistics of RESCUER's performance

In the example of a very long session of a total of 73 commands, RESCUER responded to seven commands which it considered problematic and suggested alternative commands which were either verified correct by the user's later actions or at least seemed reasonable. These seven commands account for almost a 10% of the whole session.

It also had a strong expectation on about nine commands of the session. These nine commands, if added to the seven commands where it produced a response, make a total of 16 commands that RESCUER had a strong opinion about and hence managed to perform plausible plan recognition. These account for slightly more than 20% of the whole session.

RESCUER succeeded in offering plausible corrections for four commands that had failed. In three out of four commands it was verified that RESCUER managed to suggest the right correction in terms of the user's goals. One of the corrections remained unverified, but certainly looked plausible.

RESCUER did not respond to eight commands that had failed. In principle, it could have responded to more commands in a more expanded version of its prototype implementation. However, for some commands it simply may not have been plausible to offer some suggestion.

6.2. RESCUER's reasoning and success in cases where the command had failed

In the sample session, 12 commands failed completely to do anything at all except producing an error message. These were the following commands:

```
9     %cp prolog.dcg1 dcg1
12    %cp.dcg1 prolog
16    %top prolog
22    %cp prolog.is_d_f prolog
34    %greetings
36    %greetings
48    %I
49    %more uk.general
```

54 %cp dcg1 dcg2
56 %cp dcg1 dcg2
59 %isa_d_f
64 %top cp dcg1 dcg2

RESCUER found them all ‘suspect’ and managed to find plausible corrections for four of them, at commands 16, 22, 48 and 49.

RESCUER uses two degrees of expectation about commands typed, to rate the degree of certainty about the hypotheses, *neutral* and *expected.* This means that if RESCUER has generated a hypothesis with a label ‘neutral’ and a hypothesis with a label ‘expected’, only the latter will be presented to the user because RESCUER has more reasons to believe that this is a good alternative. If there are more than one hypotheses with the same label then all of them are presented.

The corrections that were suggested to the user by RESCUER for the example were the following:

At command 16 RESCUER suggested % ls prolog with a degree of certainty ‘neutral’.
At command 22 RESCUER suggested % cp prolog.isa_d_f prolog with a degree of certainty ‘expected’.
At command 48 RESCUER suggested % ls with a degree of certainty ‘neutral’.
At command 49 RESCUER suggested the following two possible replacements:
% vi uk.general with a degree of certainty ‘neutral’.
% more.uk.general with a degree of certainty ‘neutral’.

The replacements that RESCUER suggested were confirmed correct with respect to the user’s intentions as this was shown by the subsequent commands that the user issued.

6.3. The degree of utility of a successful response

We assess the degree of utility of a correct suggestion in terms of how many commands the user issued in order to recover from an error that RESCUER managed to identify. This only applies to the verified suggestions of RESCUER.

1. 16% top prolog. RESCUER neutral: ls prolog. In this case the user wanted to see what the contents of prolog were but did not know how to do it from his/her home directory. Therefore s/he went into prolog, typed an ‘ls’ and went back to his/her home directory where s/he was originally. All this took him/her three commands to do (17, 18 and 19), which s/he would not have needed to type if s/he had seen RESCUER’s response.
2. 22% cp prolog.is_d_f prolog. RESCUER expected: cp prolog.isa_d_f prolog. This time it took the user four commands to recover from his/her error (23, 24, 25 and 26) and took him/her three commands to find out what had gone wrong. RESCUER could have saved him/her all this trouble.
3. 49% more uk.general. RESCUER neutral: vi uk.general. RESCUER neutral: more.uk.-general. In this case, the right command came immediately after the wrong command. Still the user had to retype the command, with the possibility of making new typing errors.

7. Conclusions

The main point of this paper is that a kind of 'human reasoning' is crucial for the construction of user interfaces that would provide more tolerance to user errors. A user-interface like this could be feasible to construct if it focuses primarily at 'plausible' user errors. Providing assistance to users for these errors would alleviate much of the frustration of users and reduce the risk of the users being involved in catastrophic errors with respect to their intentions.

Human Plausible Reasoning theory has been examined for the above purpose and has proved successful at the generation and evaluation of hypotheses about possible user's beliefs underlying the user's observed actions.

In particular, HPR provides a framework for generating hypotheses about misconceptions and distinguishes between the conceptual levels at which the misconceptions may have occurred and their types. Competing hypotheses for possible users' errors can be reduced if commands are assigned more meaning with respect to the user's goals by some plan recognition mechanism. In our view, the command instances issued by a user do not play an important role with respect to plan recognition as long as the commands used produce (or not) some expected effects.

The approach taken in this research can be generalized to be used in domains other than command languages since the theory used is domain-independent. For example, in a GUI (Graphical User Interface) a user's action could be a mouse event. In this case again we could assume that the user intends to use the interface correctly. Errors could also occur at different cognitive levels depending on the kind of similarity of the action issued with the action that would serve the user's goals. For example, there could be a geographical similarity of the actions, meaning that the user may have clicked on a place near the one intended, or a misconception on the semantics of actions in terms of the effects that clicking on something would have and so on. HPR could be used to generate hypotheses about possible errors using the statement transforms.

The notion of instability could be used in this case as well, to give more meaning to sequences of actions. For example, selecting an object or objects could be considered as adding an instability. This means that there should be an action following this, otherwise the action of selecting does not have any sense of purpose.

Acknowledgements

I would like to thank Professor Ben du Boulay, Dr Mark Millington and Professor Roger Hartley for their constructive comments on a previous version of this work. I would also like to thank Professor Chris Johnson (editor of this volume) and the anonymous reviewers for their helpful comments on this paper.

References

[1] M. Virvou, A human plausible reasoning theory in the context of an active help system for UNIX users. Ph.D. Thesis, Dept of Cognitive and Computing Sciences, University of Sussex, 1992.

[2] M. Virvou, J. Jones, M. Millington, Virtues and problems of an active help system for UNIX, in: S. Hegner, P. Norvig, R. Wilensky (Eds.), Intelligent systems for UNIX, Dordrecht, The Netherlands, Kluwer Academic Publishers (1999).
[3] H.U. Hoppe, Deductive error diagnosis and inductive error generalization for intelligent tutoring systems, Journal of Artificial Intelligence in Education 5 (1) (1994) 27–49.
[4] A. Mitrovic, S. Djordjevic-Kajan, L. Stoimenov, INSTRUCT: modeling students by asking questions, User Modeling and User Adapted Interaction 6 (4) (1996) 273–302.
[5] R. Davis, Retrospective on diagnostic reasoning based on structure and behavior, Artificial Intelligence 59 (1993) 149–157.
[6] S.A. Cerri, V. Loia, A concurrent, distributed architecture for diagnostic reasoning, User Modeling and User-Adapted Interaction 7 (2) (1997) 69–105.
[7] A. Collins, R. Michalski, The logic of plausible reasoning: a core theory, Cognitive Science 13 (1989) 1–49.
[8] A. Gertner, Plan recognition and evaluation for on-line critiquing, User Modeling and User Adapted Interaction 7 (2) (1997) 107–140.
[9] D.W. Oard, The state of the art in text filtering, User Modeling and User-Adapted Interaction 7 (3) (1997) 141–178.
[10] E.A. Rich, User modeling via stereotypes, Cognitive Science 3 (1979) 329–354.
[11] J. Jones, M. Virvou, User modeling and advice giving in intelligent help systems for UNIX, Journal of Information and Software Technology 33 (2) (1991) 121–133.
[12] R. Wilensky, D. Chin, M. Luria, J. Martin, J. Mayfield, D. Wu, The Berkeley UNIX Consultant Project, Computational Linguistics 14 (4) (1988) 35–84.
[13] D.N. Chin, Intelligent agents as a basis for natural language interfaces. Report no. UCB/CSD 88/396, Computer Science Division (EECS), Berkeley, California, 1988.
[14] M. Luria, Knowledge intensive planning. Report no. UCB/CSd 88/433, Computer Science Division (EECS), Berkeley, California, 1988.
[15] J.H. Martin, A computational theory of metaphor. Report no. UCB/CSd 88/465, Computer Science Division (EECS), Berkeley, California, 1992.
[16] J. Mayfield, Controlling inference in plan recognition, User Modelling and User-Adapted Interaction 2 (1-2) (1992) 55–82.
[17] A.E. Quilici, M.G. Dyer, M. Flowers, AQUA and intelligent UNIX advisor, in: B. du Boulay (Ed.), Proceedings of the 7th European Conference on Artificial Intelligence, vol. II, 1986, pp. 33–38.
[18] A. Quilici, AQUA: a system that detects and responds to user misconceptions, in: A. Kobsa, A. Wahlster (Eds.), User Modelling and Dialog Systems, Springer, New York, 1988.
[19] J. Breuker, J.R. Winkels, J. Sandberg, A shell for intelligent help systems, in: Proceedings of the International Joint Conference on Artificial Intelligence, 1987, pp. 167–173.
[20] J. Breuker, Coaching in help systems, in: J. Self (Ed.), Artificial Intelligence and Human Learning, Chapman and Hall, London, 1988, pp. 310–337.
[21] J.R. Hartley, M. Smith, Question answering and explanation giving in on-line help systems, in: J. Self (Ed.), Intelligent Computer Aided Instruction, Chapman and Hall Computing, London, 1988.
[22] R. Winkels, Explorations in Intelligent Tutoring and Help. IOS Press, 1992.
[23] K.L. McGraw, Performance support systems: integrating AI, hypermedia and CBT to enhance user performance, Journal of Artificial Intelligence in Education 5 (1) (1994) 3–26.
[24] M.H. Burstein, A.M. Collins, Modelling a theory of human plausible reasoning, in: T. O'Shea, V. Sgurev (Eds.), Artificial Intelligence III: Methodology, Systems, Applications, Elsevier Science, North Holland, 1988, pp. 21–28.
[25] M.H. Burstein, A. Collins, M. Baker, Plausible generalisation: extending a model of human plausible reasoning, The Journal of the Learning Sciences 3/4 (1991) 319–359.
[26] E. Hollnagel, The phenotype of erroneous actions: implications for HCI design, in: G.R.S. Weir, J.L. Alty (Eds.), Human–Computer Interaction and Complex Systems, Academic Press, London, 1991.
[27] E. Hollnagel, The phenotype of erroneous actions, International Journal of Man–Machine Studies 39 (1993) 1–32.