Artificial Intelligence Review **14:** 229–252, 2000.

An Assumption-based Truth Maintenance System in Active Aid for UNIX Users

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

Abstract. This paper deals with the problem of assigning meaning to the interaction of a user with a command-driven system such as UNIX. This research is part of the construction of an active intelligent help system that monitors users in order to offer spontaneous help when they are facing problems. In order to ensure this, the help system must build and maintain a model of the user. We describe a mechanism that is used by the user modelling component of such a help system. This mechanism makes explicit assumptions about the user which account for different hypotheses about what the user is actually thinking at every stage of the interaction. The consistency of these assumptions is managed by an Assumption-based Truth Maintenance System (ATMS). The selection between a number of different user models is based on the information which is extracted from the feedback that UNIX provides in response to user actions.

Keywords: Assumption-based Truth Maintenance System, advice generation, Intelligent Help Systems, UNIX commands, user's errors, user interfaces, user modelling

1. Introduction

Intelligent Help Systems for UNIX are meant to provide additional on-line help to users that interact with UNIX. Depending on the kind of response of the help system they are usually classified into passive and active systems.

Passive systems generate response on the user's initiative (i.e. the user explicitly asks for help) whereas active systems on their own initiative (i.e. they spontaneously intervene in order to offer advice) when they think that the user has problems. Passive help systems are supposed to answer user's questions in natural language such as "How can I remove a file?" or "What does the command "mv" do ?". Quite rightly the emphasis is on issues of natural language understanding. A prototypical passive help system is the UNIX consultant (UC) (Wilensky et al. 1986). Active help systems are supposed to monitor the users' actions and reason about them so that they can offer spontaneous help when they think that the user is in trouble. The model for

this form of help system is the expert who helps out after having observed "over your shoulder" that you are having problems.

Our overall goal is to construct an active help system for the UNIX operating system, in common with a number of other groups (Kemke 1986; Matthews and Pharr 1999). In a companion paper in this volume (Virvou et al. 1999) we have argued that to provide intelligent help in this domain requires considerable knowledge of the user and a thorough understanding of the interaction between the user and UNIX. The main repository for this understanding will be a user model developed and maintained by a user modelling component of the help system. Our work to date has been to identify what a properly constituted user model is.

In section 2 we describe the features and problems of UNIX as a domain for user modelling for an active help system. In section 3 we briefly describe some related work. In section 4 we give the overall approach taken for modelling UNIX users. In section 5 we present our definition of a user model and approach to constructing it. In section 6 we show the representation of UNIX that underlies our prototype implementation. In sections 7 and 8 we describe our prototype implementation of the user modelling component and in section 9 we show how the prototype works on some examples. Finally, in section 10 we give the conclusions and in section 11 we describe the recent developments in this research.

2. UNIX as a Domain for User Modelling

In this section we consider the nature of UNIX itself and its users because these have a considerable impact on user modelling for an active intelligent help system. It is the aim of our research to determine in detail what form a user model should take. However, at this stage we shall simply suppose that a user model contains whatever information is necessary in order to make the advice and help offered, individual to the user and her/his current circumstances.

UNIX is an operating system with an elegant system design which attracts a lot of users. However, the user interface is not so elegant. Norman (1981) argues that often command names bear no relationship to their functions and have inconsistent syntax. He also argues that the system's terseness or silence during interaction puts an additional burden on the user's memory.

Actually, the terseness of UNIX does in fact cause difficulties for the user; if an action executes without an error (as perceived by UNIX) the user is not explicitly informed of the actual effect. This can lead to difficulties where the action achieves an effect different to that which the user both intended and believes occurred. However, somewhat paradoxically, this terseness is

also one of the principal features of UNIX which leads us to believe that we can build the help system we envisage. This factor ensures that the user must explicitly seek information from UNIX in such a manner that an on-line help system can observe the query and thereby determine the user's focus of attention.

On the second count, the semantic representation of UNIX is tractable, a particularly simple representation being presented in section 6. There is a significant subdomain, consisting of the file store and commands to manipulate it such as *mkdir*, *rmdir*, *cd*, *cp*, *rm*, *mv*, *ls*, *lpr* and *vi*, which accounts for a significant proportion of users' time spent using UNIX and the problems they encounter and is representative of the domain as a whole. In addition, the nature of a computer operating system is such that we can easily create an interface which can take the action issued by the user and reason about it before it is executed. This, for example, gives us the scope to avoid a disaster such as the user inadvertently deleting a file.

The aim of our research is to build a system that would provide automatic help in situations where a user's utilisation of UNIX has been found to be deficient. Deficiencies may occur at a number of levels, such as planning, semantic, environmental,[1] syntactic, lexical and typographic. However, there appear to be a number of different notions of adequacy for methods of achieving goals. Users often trade off brevity and/or accuracy in favour of familiarity or (perceived) safety. A typical example of this is achieving 'move' by 'copy and remove', where the incremental nature of the latter approach allows the user to monitor the success of the individual steps in the plan.

Modelling UNIX users is particularly difficult as UNIX is used by a range of users with different levels of knowledge of the system and with different goals in mind. In addition, it offers a wide range of commands and command options as well as a rich collection of methods for combining them. This leads us to think that a simple scheme for user modelling, such as one utilising stereotypes (Rich 1979), would be inadequate. Furthermore, users typically represent a moving target to a user modelling component, in that goals and plans may change and features of the system may be learnt or forgotten.

Some aspects of the behaviour of users in the UNIX domain will present great difficulties for a user modeller in an active help system, in that they do not leave a trace in the dialogue between UNIX and the user.

The range of such behaviour includes:

- Users may *arbitrarily* remake choices concerning goals and plans; these often seem to be inspired by the interaction with UNIX, but in a manner that is very difficult to relate to the task in hand. For example, users may interrupt sequences of actions aimed at achieving some goal in order to see their mail.

We distinguish these arbitrary changes from modifications of the user's plans and goals which become necessary when an error is detected through feedback from UNIX.

- Users learn by unobservable processes. For example, they seek help from other users or the (off line) manual.
- Users forget.
- Users can be selective in their attention. For example, the information gathering command 'ls' is often employed to acquire only a fragment of the information it actually presents.
- Users can be inconsistent. For example, we have observed a user[2] achieve her/his aim of moving all the proper files in a particular directory into a new sub-directory called 'ht', by creating the new subdirectory and then using the action 'cp $*$ ht'. The inconsistency will be present if s/he believes that the match of '$*$' will include 'ht' and that 'cp' cannot be used to copy directories.
- Users make mistakes.

3. Related Work

The work which most closely resembles the present proposal for a user modelling component is GUMS (Finin and Drager 1986), in which an architecture for a domain-independent user modelling component for an application program is presented. The user model forms the basis of responses by the user modeller to queries about the user from the application program.

An initial model of the user is selected from a hierarchy of stereotypes. A stereotype is a generic model of a class of users of the application program and consists of facts and rules. For example, if the application program is interacting with PC users then the stereotypical user model may assert that:

typically, the user understands the term 'rom'

typically, if the user understands the term 'rom' then he understands the term 'ram'

As the interaction of the application program with the user progresses, additional information about the user is passed to the user modeller for incorporation into the user model. The incorporation of these new facts specialises the stereotype model to the individual user. However, this specialisation raises the problem of the consistency of the resulting user model. For example, with the above stereotype model, suppose the application determines that the user does indeed understand the term 'rom' but not the term 'ram'. Finin

and Drager propose that the consistency of the user model is managed by Doyle's truth maintenance system (Doyle 1981) in the context of a forward-chaining inference mechanism. A prototype help system component, GUMS, is described which employs simple default rules and backward chaining.

Since the scheme Finin and Drager describe is domain-independent, they do not consider how relevant information may be extracted from the interaction between the user and the application program. Their proposal is merely concerned with the management issues involved once the information has been obtained.

4. Overall Approach

The development of a user model, and its utilisation, will be over a number of natural time scales. The first major time scale, which we shall refer to as an *interaction*, is that in which the user issues an action to UNIX, observes the response (if any) and formulates the next action. Thus, an interaction begins when the user presses the 'return' key to issue an action and ends when 'return' is pressed again to issue the next action. We shall reserve the word *command* for functions provided by UNIX, and *action* for a particular instance of the command's use. Thus an action will involve a command and zero or more arguments. We shall further divide an interaction into two phases: the *execution phase*, in which UNIX attempts to execute the action, and the *feedback phase* which begins when the user receives the feedback (if any) or a new shell prompt. The feedback stage is the period in which the user formulates the next action in the sequence. The second major time scale, which we shall refer to as a *session*, is between logging on to UNIX and the corresponding logging off. Thus, a session will consist of a number of interactions between the user and UNIX.

In view of the different time scales over which we shall be observing and modelling the user, we anticipate that different parts of a user model will have differing life-spans. Thus, information concerning the user's knowledge of UNIX, including any misconceptions, information concerning typical tasks attempted and the manner in which s/he attempts them, will be determined over, and persist for, a number of sessions. Other information will concern the current session only, and will be discarded as soon as it becomes redundant.

The motivation for our overall approach comes from hand-analysis of example user sessions, some of which are described in a companion paper in this volume (Virvou et al. 1999). Most sessions were understood by means of a relatively rich interaction between a number of different sources of knowledge, such as those listed above, where no one source could by itself provide a complete understanding. Thus, the overall architecture of the help system

must support diverse sources of knowledge and provide a mechanism to support their interaction in a fruitful and opportunistic manner. For these reasons, the most promising architecture is that of a blackboard system (Hayes-Roth 1983).

In an intelligent help system with the blackboard architecture, a user model will consist of a collection of hypotheses in the blackboard. While the precise form of this collection of hypotheses will be described later, we shall begin by presenting some general remarks on our approach to user modelling.

Modelling users in the context of an active help system means that the modeller has to extract from the interaction information on which to base the user model. We believe that important opportunities for determining elements of a user model are those events when the user receives explicit feedback from UNIX, such as an error message. If we suppose that generating an error message is never intended, then the user's beliefs must change during the transition from the execution to the feedback phase.

For example, consider the following sequence of actions, which is based on the perquish example discussed in (Virvou et al. 1999). The actions are issued in the user's home directory, which contains the files 'fred' and 'fredl':

```
1% mkdir perqish
2% cp fred perqish
3% cp fredl perquish
4% cd perqish
5% mv fredl drconsult.pas
   mv: Cannot access fredl
```

The most plausible explanation of the error message generated by action 5 is that the user intended action 3 to be 'cp fredl perqish'. Interestingly, the typing mistake can be explained by either of the observations that the substring 'qu' is common in English or that 'u' is next to 'i' on the keyboard. The user does not realise that s/he has made a mistake, however, until the error message is received as a result of action 5. On receiving this feedback, s/he is forced to review her understanding of the interaction. We believe that this cue can also provide valuable evidence for a user modeller.

Such belief revision by the user must also take place when the feedback produced by UNIX in response to an action such as 'ls' or 'pwd' is inconsistent with the user's beliefs. For example, the user may believe that a particular file exists in the current working directory, but subsequently discover that 'ls' does not list it. Once again the user must revise her/his beliefs. Equally, a user modeller must revise its own user model if it is inconsistent with the feedback the user is receiving.

The user modelling component we have designed will attempt to construct a correct model of what the user believes during the execution phase of an

interaction. For the construction of the user modelling component we will make certain assumptions about the user. We shall assume that the user is attentive (to the extent that s/he notes all feedback from UNIX) and that the only learning and choice revision which takes place is driven by the feedback from UNIX. In this way we rule out most of the difficulties we identified in section 2 and we concentrate only on the possibility of users making mistakes. We assume that the user reasons consistently from a monotonically increasing collection of beliefs until the feedback from UNIX is perceived to be inconsistent with that collection of beliefs. At this point, the user is forced to reappraise the situation, and her/his beliefs must undergo a non-monotonic shift. This model will provide useful evidence for determining what is believed during the feedback phase and provide a context in which an intelligent help system can offer assistance should it be deemed necessary. In the example given above, it would be practical to indicate that the problem is with action 3 which the user "thought created ∼/perqish/fredl", if it became apparent that the cause of the error message was not understood.

Two of our criteria for an acceptable user model will be given in terms of the consistency of the hypotheses forming the user model. In view of this, we need a flexible mechanism for managing consistency. In addition, modelling the user will require belief revision on the part of the modeller and the ability to compare alternative, and incompatible user models. Thus we have chosen to manage the consistency of the blackboard with an assumption-based truth maintenance system (ATMS) (de Kleer 1984, 1986).

The monotonicity of the user's beliefs that we are assuming needs a little explication. In our user modelling, we aim to capture how the user's beliefs change as the result of errors being uncovered. However, there are other ways in which beliefs can change which we do not mean to discount by our assumption of monotonicity. For example, if a user deletes the file mbox then s/he must once have thought it existed and now no longer believes that. If we simply represented the user's beliefs as a collection of facts, this loss of belief would have to be represented as a loss of the corresponding fact. This is a non-monotonic change, which we have suggested should only occur when it is driven by the revelation of some form of error or misconception. We can retain the monotonicity of the user's beliefs under such circumstance by time-stamping assertions of user beliefs. For example, the belief in the existence of the file mbox will be of the form "up to and including the nth action mbox exists". Lack of belief in the existence of mbox after removing it with the n+1th action will be represented by the lack of an additional assertion of the form "up to and including the n+1th action mbox exists". In this way, monotonicity of the set of user beliefs is retained while allowing

the non-monotonic shifts we have identified to take place when an error or misconception is revealed.

The user modelling scheme we describe and our choice of an ATMS arose out of our experience with initial prototype user modellers constructed using a blackboard shell (Jones et al. 1988). The consistency-maintenance mechanism employed by the blackboard system proved unsatisfactory for the purposes of user modelling. The principal problem was that it could not maintain several mutually inconsistent sets of active hypotheses at the same time. As with justification-based truth maintenance systems, the blackboard system maintained a single consistent set of hypotheses. For user modelling purposes, we must be able to investigate several different and incompatible interpretations of a sequence of actions and be able to swap effortlessly between them.

5. The User Modelling Mechanism

In this section we present a mechanism on which to base the user modelling component of the proposed help system. This mechanism is applicable in the execution phase of an interaction when the currently favoured user model is no longer acceptable on the basis of the feedback that UNIX will provide to the user. We shall concentrate on the particular case in which the feedback is in the form of an error message. This mechanism makes critical use of a formal definition of what constitutes a user model.

We stated in section 3 that a user model is to be a collection of hypotheses in the blackboard. With respect to the content of the model, we consider it necessary to include a record of how the user currently views each action issued, which constitutes the current model, as well as how they were viewed in the past, should this be different, which constitutes the past model(s). The user's view of the actions will include the text of the actions, their semantics and how they relate to achieving her/his goal(s). We shall make no attempt to reproduce any deep structure of user's mental representations of goals, plans and knowledge of UNIX. In the example presented in section 4, the current user model during the execution phase for action 6 (supposing that there is such an action) should record that action 3 was 'cp fredl perquish', whereas the past model records that it was 'cp fredl perqish'. This is intended to correspond to the plausible explanation that until the error message was received, action 3 was believed to have been 'cp fredl perqish'. The current model corresponds to the moment the user presses 'return' to initiate a new execution phase, and is considered to remain fixed throughout the execution phase.

The difference between a current model and a past model will be in the number of interactions of the session that they account for. A current model must account for all the actions issued so far, past models only account for some initial subsequence. Each past model began life as a current model. The change in status from current to past occurs during the feedback phase when the user assimilates the feedback produced by UNIX as a result of the action just executed. If that feedback is inconsistent with the current model then it becomes a past model and the user must acquire a new current model. Thus, current and past models are inconsistent and are intended to model a non-monotonic shift in the user's view of the interaction.

The user modelling component of the help system must identify the correct current user model. We present four criteria which define an acceptable current user model during the execution phase. These criteria will be used by the user modeller to monitor whether what it currently hypothesises is the current model, remains viable in the light of the latest user action. Should it not, then the user modeller must revise its hypothesis. This change of hypothesis by the modeller during the execution phase does not correspond to a supposed change by the user. As described above, such a change by the user must take place during the feedback phase. The change we are defining is where the user modeller discovers that it has not properly identified the current user model.

A current user model is acceptable if it satisfies the following criteria:

- Criterion 1: It is internally consistent. The user is assumed to be rational, reasoning consistently from her/his beliefs.
- Criterion 2: It is externally consistent with the feedback from UNIX. The user is assumed to be attentive. This is modelled by inserting the feedback UNIX provides into the model.
- Criterion 3: It is complete. It must address all the relevant issues, from the user's typing skills to goals.
- Criterion 4: It is parsimonious. The fewer non-default assumptions the better.

Criteria 1 and 2 result from the assumptions about the user and the interaction detailed in section 4. Criterion 3 assures that no issues are avoided, while criterion 4 is simply desirable.

The user modeller will make explicit assumptions about the user and the interaction which will be communicated to the ATMS by the blackboard system. An important function of the ATMS is to maintain a record of sets of assumptions, called nogood sets, from which the blackboard system has derived a datum equivalent to 'false'. We shall be able to ensure that criteria 1 and 2 are satisfied by the judicious definition of two kinds of nogood set:

- A particular user model cannot be based on two different versions of what the user intended to type for a particular action in the sequence.

- We assume that users do not intend to produce error messages and so the assumptions underlying a proof that the user's actions do produce one will be a nogood set.

Initially the user modeller will favour default assumptions about the user and the interaction, whereby everything is assumed to be correct. Modelling proceeds monotonically until we can derive false in the current user model as the result of some inherently false collection of assumptions being made. Criterion 1 states that the resulting collection of hypotheses is unacceptable as a current user model, and so the user modeller will attempt to select an alternative model in which false is not derivable. This alternative user model will be based on non-default assumptions.

The default current user model can only be identified as unacceptable on the basis of an error message. It cannot be discounted on the basis of an inconsistency between the user model and the feedback provided by UNIX in response to commands such as 'ls' and 'pwd', because there cannot be such an inconsistency. However, once a non-default interpretation has been adopted, it can be discounted in this way. Thus, for example, a current user model which is based on the non-default assumption that it is possible to copy a directory using the command 'cp'[3] is invalidated if the user later investigates the status of the copy of the supposed directory using 'ls' in such a manner as to discover that it is in fact a file. It is criterion 2 which suggests the modeller should detect such contradictions, and as this feedback is inserted into the user model criterion 1 suggests we have to revise the current user model.

In section 7, we shall present a preliminary implementation of a user modelling component utilising this user modelling scheme.

6. Representation of the UNIX Domain

It is sufficient to represent the file store and the effect of actions on it. For this purpose we use a simple STRIPS-like representation (Fikes and Nilsson 1971). The fundamental building block is that of a file or directory and a Prolog term of the form:

exists(Name,Type,Contents)

asserts that Name (a full UNIX path name) has type Type (file or dir; for proper file and directory respectively) and contents Contents. We shall suppose that the user's home directory is called 'home'. Full path names are represented internally using the left-associative binary operator '/', so that the path name '/home/prolog/utilities' will be rendered as 'home/prolog/utilities' without loss of generality. For the sake of simplicity, this prototype assumes that the user employs simple path names, which is quite reasonable for the examples we are considering. The user modeller, however, performs the bulk

of its reasoning using complete path names. The contents of a proper file will be a term, which is sufficient, for example, to allow us to represent the fact that the contents of a copy of a file are the same as the contents of the original. As yet, we do not use contents of directories in an important manner, so that they are left as unnamed variables (that is, _) in the example presented in this chapter.

A state of the file store is a list of files and directories, along with a record of the current working directory of the form:

```
cwd(Directory)
```

where Directory will be a full path name. A configuration is a state of the file store after N actions have been executed:

```
configuration(N,Configuration)
```

For the analysis of a sequence of actions we must define the initial state of the user's file store. For the perquish example cited earlier, the initial state is:

```
initial_state([
        exists(home,        dir, _),
        exists(home/fred, file, fred),
        exists(home/fredl, file, fredl),
        cwd(home)
            ]).
```

The semantics of a UNIX command is defined by giving the semantics of each of its legitimate uses. For example, 'mv' can be employed to move a file into a directory, to move a file to a new file in the same directory or to move a file to an existing file in the same directory. For each such use, there is a unit clause defining the preconditions required for correct execution of that instance of the command, and the effects of execution in terms of additions to, and deletions from, the current configuration of the file store. Preconditions may be both positive and negative, the latter being recorded in the list following 'without'. The instances of 'mv' which rename a file (such that the new name is not that of an existing file or directory) and move a file to an existing directory (which does not contain a file of the same name) are represented by the following unit clauses:

```
semantics(mv(A,B),
        [needs [exists(A,file,Content)]
           without
              [exists(B,_,_)],
        adds [exists(B,file,Content)],
        deletes [exists(A,file,Content)]
         ]).
```

```
semantics(mv(A/File,B),
        [needs [exists(A/File,file,Content),
              exists(B,dir,_)]
           without
              [exists(B/File,_,_)],
        adds [exists(B/File,file,Content)],
        deletes [exists(A/File,file,Content)]
        ]).
```

In these clauses, 'needs', 'adds' and 'deletes', are unary prefix operators, and 'without' is an infix binary operator.

The semantics of instances of a command that produces error messages are determined "on the fly", by examining the correct forms to see which precondition was not satisfied. In these cases, the 'adds' list will be [error(Message)], while the 'deletes' list will be empty. In previous prototype user modellers we have built, instances of commands that produced error messages were represented by unit clauses such as those given above. In these cases, the only 'add' was the appropriate error message. The present style of representation avoids the additional search problem caused by extra clauses for command semantics.

7. A Prototype Implementation of the User Modelling Component

In this section we describe a prototype implementation, in Prolog, of a user modelling component based on the mechanism we described in section 5.

It is outside the scope of this paper to describe the logic programming language Prolog in any detail. The interested reader should consult (Stirling and Shapiro 1986). However, a few introductory remarks may help to explain the fragments of code we shall be presenting.

As with Lisp, the syntax in Prolog for data and the program is the same, and it is the context in which they are used which distinguishes between them. The fundamental syntactic unit is the term. The basic terms are constants and variables. Terms with structure, which denote relationships, consist of a functor applied to terms. Those fragments of code we present which are terminated with a fullstop are unit clauses in the Prolog program that constitutes the prototype user modeller. Their utilisation in the execution of the program is such that they encode factual relationships between data items. The remainder of a Prolog program consists of non-unit clauses which define how to compute with the unit clauses and data. Computation takes place in order to solve a goal and utilises the clauses in a backward-chaining manner.

Variables in Prolog begin with an upper-case letter, the exception being the anonymous variable. The scope of a variable is the clause in which it appears.

One of the most powerful aspects of Prolog as a programming language is the pattern matching used between terms, namely unification, which can be exploited in controlling clause selection and computation equally.

The remaining element of syntax to discuss is the representation in Prolog of lists. The list with members a, b and c is denoted (a,b,c). The construction and destruction of lists is achieved by the same syntactic structures to propagate the implications of the re-derivation (under a new set of assumptions) of a previously derived datum, without the inference mechanism ensuring that this is the case. Completeness is maintained in our application by the inference mechanism.

The rules which construct user models utilise the following hypotheses, with their respective intended interpretations, in addition to those described above referring to the configuration:

- act(I,Action): Action is the actual text of the user's Ith action.
- intended_action(I,Action): Action is the full path name version of what it is believed the user intended to type as the Ith action. This need not be the full path name version of what was actually typed.
- instantiated_semantics(I,Act,Semantics): The Ith action Act has the semantics Semantics in the context of the current file store. Semantics will be a structure of the form [adds Adds, deletes Deletes], where Adds and Deletes are lists of terms of the form 'exists(A,B,C)' or 'cwd(A)' as appropriate for Act in the current configuration.
- goal(Goal,Method): Goal is achieved by Method. For example:
  goal(move(home/fredl,home/perqish/drconsult_pas),
  [cp(home/fredl,home/perqish/drconsult_pas),rm(home/f redl)])

Thus, the user modeller constructs user models as collections of assertions of the above form. In addition, the following Prolog goals are utilised in the rules that perform the analysis:

- expand(Act,Configuration,Expansion): The full path name version of Act in Configuration is Expansion.
- corrected_version(Action,Configuration,Correction): A potential correction of the full path name action Action in Configuration is Correction. Corrections are intended to make allowance for the user making typing or spelling mistakes. This predicate makes use of type information for the arguments of UNIX commands and the current configuration of the file store to propose plausible alternatives to what was actually typed. In this context, $word_2$ is a plausible correction of $word_1$ if:
  - $word_2$ has one more letter than $word_1$
  - $word_2$ has one less letter than $word_1$
  - $word_2$ is $word_1$ with two adjacent letters interchanged
  - $word_2$ is $word_1$ with one letter changed

- instantiate(Act,Configuration,Semantics): The full path name action Action has meaning Semantics in Configuration.
- effect(Semantics,Configuration,NewConfiguration): NewConfiguration is the result of effecting Semantics on Configuration.
- associated_goal(Act,Configuration,Goal): Goal is a likely goal to associate with Act in Configuration.
  The Configuration is relevant in order to be able to distinguish whether 'mv home/a home/b' achieves the goal of moving 'home/a' to 'home/b' or 'home/b/a' (that is, whether 'b' is a directory or not).

Two kinds of explicit assumption can be made in a rule:

- default(Assumption).
- non_default(Assumption).

with the obvious interpretations. The user modeller only makes assumptions concerning action(I,Action), that is what it is believed the user intended to type. Thus, the hypothesis intended-action(I,Act) is supported by the assumption default(action(I,Act)) or non-default(action(I,Act)), depending on whether Act is what was actually typed or what it is conjectured that the user intended to type. This distinction will be used to ensure that criterion 4 for a user model is satisfied. The ATMS does not itself distinguish between default and non-default assumptions, rather it is up to the programmer to supply the distinction.

To analyse a particular sequence of actions, the prototype first loads the appropriate initial state (provided by the experimenter) and then reads UNIX actions one at a time. The effect of the rules presented below is that each action read is recorded as what the user actually typed. Then the semantics of the action issued and of possible corrections, where appropriate, are determined and the current configuration of the file store updated. Finally, a number of hypotheses about possible goals and actions that achieve them are made.

The rules that perform the initial analysis are as follows:

```
  act(I,Act)
& call(J is I-1)
& configuration(J,Configuration)
& call(expand(Act,Configuration,Expansion))
& assume(default(action(I,Expansion)))
⇒ intended_action(I,Expansion).

  act(I,Act)
& call(J is I-1)
& configuration(J,Configuration)
& call((expand(Act,Configuration,Expansion),
     corrected_version(Expansion,Configuration,Correction)))
```

& assume(non-default(action(I,Correction)))
⇒ intended_action(I,Correction).

intended_action(I,Act)
& call(J is I-1)
& configuration(J,Configuration)
& call(instantiate(Act,Configuration,Semantics))
⇒ instantiated_semantics(I,Act,Semantics).

instantiated_semantics(I,Act,Semantics)
& call(J is I-1)
& configuration(J,Configuration)
& call(effect(Semantics,Configuration,NewConfiguration))
⇒ configuration(I,NewConfiguration).

An instance of the second rule in use is as follows. Suppose that for action 2 the user types 'cp fredl perquish' and that the state after the first action is:

[exists(home/fredl,file,fredl),
exists(home/fred2,file,fred2),
exists(home/perqish,dir,_),
cwd(home)
].

The full path name version of what the user typed is 'cp home/fredl home/perquish'. Since one instance of 'cp' takes a file as first argument and a directory as second argument, there is a directory 'home/perqish' in the current state and 'perquish' is a possible mis-type of 'perqish' then this rule suggests 'cp home/fredl home/perqish' as a possible user intention, making a non-default assumption that this is in fact the case.

Finally, the current implementation has a number of rules about goals. Goals are associated with a list of actions which we shall call a method.[4] We shall assume that the user has a plan designed to achieve a goal, and a method will be our approximation to that plan. An hypothesis of the form:

goal(Goal,Method)

asserts that Method is intended to achieve Goal. The rules associate goals with methods based initially on the commands 'cp', 'mv', and 'rm' only. Goals may be associated with single actions:

goal(copy(A,B),[cp(A,B)])

or with a number of actions:

goal(move(A,B),[cp(A,B),rm(A)])

The commands 'cd' and 'mkdir' are incorporated dynamically into methods. The presence or precise position of actions involving these commands in

a plan is open to considerable variation by the user, and as such they are difficult to incorporate in explicit methods for goals, other than by enumerating all the possibilities. For example, particular instances of ‘cd’ can be altered or added to or removed from the user’s plan by utilising different path names for arguments to other actions in the plan. Actions involving ‘mkdir’ can be moved towards the beginning of an action sequence, provided that ‘mkdir Name’ does not retreat past ‘rm Name’, ‘rmdir Name’, or the creation of its parent.

We have chosen to handle this flexibility during the processing of user actions by the modeller, rather than extend the number or complexity of methods. It is achieved by means of a relationship between these particular actions and actions forming the rest of the method. We call this relationship enablement, and there are two kinds. The first type is of a syntactic nature and is exemplified by the use of ‘cd’ to change the current working directory. This kind of enabling action does not contribute to the satisfaction of the precondition of the action it enables, but simplifies the way in which that action can be expressed by the user. For example, to remove a file called ‘cv’ from the subdirectory ‘papers’ the user can do ‘cd papers; rm cv’ or ‘rm papers/cv’. The use of ‘cd’ merely alters how the file ‘cv’ is referred to. The second type of enabling action does contribute to the satisfaction of the precondition of the action it enables, but is secondary to the main goal of the method. This type of enablement is typified by the use of ‘mkdir’ to create directories. For example, directories have to exist in order to allow files to be created in them, but we see the goal as being to create the file.

Enablement is recorded in a method for a goal by an assertion of the form:

```
enables(I,J).
```

A selection of the rules in the prototype user modeller about goals are:

```
  intended_action(I,Act)
& call(J is I-1)
& instantiated_semantics(I,_,Semantics)
& call(not Semantics = [adds [error(_)],_])
& configuration(J,Configuration)
& call(associated-goal(Act,Configuration,Goal))
⇒ goal(Goal,[act(I,Act)]).

  goal(Gl,Acts1)
& goal(G2,Acts2)
& call((not G1 == G2,
    combine-goals(Gl,Actsl,G2,Acts2,Goal,Acts)))
⇒ goal(Goal,Acts).
```

```
  intended-action(I,cd(Dir))
& goal(copy(Dir/Name,Something),Method)
& call((member(act(J,cp(Dir/Name,Something)),Method),
    I < J,
    not member(enables(I,J),Method),
    sort([act(I,cd(Dir)),enables(I,J)|Method],Acts)))
⇒ goal(copy(Dir/Name,Something),Acts).
```

The first rule determines the goal associated with an act that does not produce an error message. The second combines distinct subgoals into goals. The third rule states that changing directory to the location of a file to be copied is an enabling action for the goal of copying. The Prolog goal sort/2 constructs a canonical list of actions for the hypothesised goal, including an assertion that action I enables action J.

The remainder of the rules about goals simply enumerate obvious relationships between goals such as ‘copy’, ‘remove’ and ‘move’, and further enablements.

8. User Models in the Implementation

The user modeller identifies the current user model by employing the four criteria we have given in section 5. A user model is a context in the ATMS. The first two criteria, concerning the internal and external consistency of the user model, are easily established for any user model through the underlying ATMS.

- The internal consistency of a user model is ensured by rules that produce nogood sets for the ATMS. These rules ensure that competing hypotheses about what the user intended to type for a particular action result in different contexts, and hence different user models:

```
  default(action(I,Act))
& non_default(action(I,Act1))
⇒ false.

  non_default(action(I,Act))
& non_default(action(I,Act1))
& call(not Act = Act1)
⇒ false.
```

- The external consistency of the user model is ensured by inserting the feedback each action provokes into the user model. The underlying assumption that the user does not intend to generate an error message can then be represented by a rule stating that when an action in the current configuration produces an error message, then the set of assumptions

which underly this derivation is nogood. If an error message is produced by an action then its instantiated semantics will indicate that in the "adds" list of the action there is an error message. Therefore the rule that ensures the external consistency of the user model is the following:

instantiated_semantics(I,_,[adds [error(_)],_])
⇒ false.

- The third criterion for user models concerned completeness of the model. In this prototype user modeller, a complete user model is one which contains a number of assertions of goals which between them account for all the actions performed by the user. The rules are such that this definition of completeness ensures that all other appropriate issues are addressed by the user model.
- The fourth, and final, criterion was that of parsimony. This is ensured in the prototype implementation by selecting collections of hypotheses which make the least number of non-default assumptions.

9. Analysis of Some Examples

We shall now show how the user modeller handles a number of examples, including the perquish example which has been discussed in section 4. After each example has been introduced we show a trace of the user modeller's behaviour. Each component of a trace consists of a user input, given immediately after the prompt %, followed by the list of assumptions underlying the current user model.

In each of the following examples, we shall assume that the initial file store is the same as for the perquish example. Thus, the user has two files in the current working directory, called fred and fredl.

For the first example, suppose that the user mistypes the name of the command involved:

% vm fredl new_file
Current model fragment is:
[non_default(action(l,mv(home/fredl,home/new_file)))]

We can see that the user's typing error is detected, not simply because it is a typing error, but because it would be inconsistent to assume it was not a typing error. The current user model is based on a non-default assumption that the user intended to use the 'mv' command. This example exploits the knowledge the prototype has about typing errors. However, in the following example, where there is potentially a typing error, in that the user may have intended to move the file 'fredl' rather than 'fred', we see a different analysis:

% mv fred another_file
Current model fragment is:

[default (action (1,mv (home/fred, home/another_file)))]

The default user model is the preferred user model, even though fred is a plausible mis-type of fredl. A user model which suggests the user intended to type fredl is not selected at this stage because it makes more non-default assumptions.

For the 'perquish' example, the file name drconsult.pas is rendered as drconsult_pas to make it a Prolog atom and the analysis is as follows:

1. % mkdir perqish
Current model fragment is:
[default (action (1, mkdir (home/perqish)))]

2. % cp fred perqish
Current model fragment is:
[default(action(l,mkdir(home/perqish))),
default(action(2,cp(home/fred,home/perqish)))]

3. % cp fredl perquish
Current model fragment is:
[default(action(l,mkdir(home/perqish))),
default(action(2,cp(home/fred,home/perqish))),
default(action(3,cp(home/fred1,home/perquish)))]

4. % cd perqish
Current model fragment is:
[default(action(l,mkdir(home/perqish))),
default(action(2,cp(home/fred,home/perqish))),
default(action(3,cp(home/fred1,home/perquish))),
default(action(4,cd(home/perqish)))]

5. % mv fredl drconsult_pas
Current model fragment is:
[default(action(l,mkdir(home/perqish))),
default(action(2,cp(home/fred,home/perqish))),
non_default(action(3,cp(home/fredl,home/perqish))),
default(action(4,cd(home/perqish))),
default(action(5,mv(home/perqish/fredl,
home/perqish/drconsult_pas)))]

We can see that until action 5 the preferred interpretation is the default interpretation. Only when this becomes unacceptable as a user model according to the criteria we have given, is an alternative current model chosen. At this point, the acceptable current model is that which assumes the user

intended action 3 to be ‘cp fredl perqish’. A possible alternative interpretation, that actions 3 and 5 were intended to be ‘cp fred perqish’ and ‘mv fred drconsult.pas’ respectively, is inferior to the selected current model according to criterion 4, because it makes two non-default assumptions.

Thus we can see that the criteria we have given select the appropriate current user model during the execution phase based on the currently available user actions. As more becomes known of this sequence it is possible that the user modeller revises its hypotheses.

The second kind of nogood set arises when having explored some line of inference an unwanted hypothesis results. In this case, the underlying set of assumptions cannot be guaranteed to lead to a contradiction, rather it depends on the particular circumstances. An example of this form of nogood set results from a configuration which records an error message. This happens in the example given above when we make the default assumption that the user intended to type ‘cp fredl perquish’ for action 3. However, the content of the assumption is not at fault here, it is what follows from it in the particular circumstances.

An alternative way of considering the two types of nogood set we have identified is in terms of the number of hypothesis tests appearing in the preconditions of the corresponding rules. The first kind of nogood set correspond to rules with more than one hypothesis test, while the second kind correspond to rules with exactly one hypothesis test. The latter kind of rule identify inherently false hypotheses, while the former identify cases where the combination of several different hypotheses creates the contradiction.

With respect to choosing what explicit assumptions a rule should make, the following obvious comment is appropriate. A nogood set of assumptions has to have some element(s) that distinguish it from an acceptable set of assumptions. This is particularly pertinent in the case of the second kind of nogood set indicated above where a particular set of assumptions may or may not lead to a derivation of false. This observation tells us when we need additional assumptions, but does not tell us what they should be about. That is up to the rule-writer.

10. Conclusions

We have described a prototype user modelling component for an active help system for UNIX. Its underlying mechanism determines the current user model during the execution phase of an interaction. The user modeller employs four criteria to identify an acceptable user model. These criteria make essential use of the consistency of the hypotheses forming the user model. We have shown how the criteria can be utilised in a forward-chaining

rule interpreter in which consistency is managed by an assumption-based truth maintenance system. The concept of a 'nogood' set of assumptions proves very important in this implementation. In our discussion of the mechanism we have concentrated on inconsistencies caused by actions that result in error messages. The assumption-based truth maintenance system has provided a good foundation on which to base our approach to user modelling, in that it supports a precise definition of what constitutes an acceptable user model. However, in building an actual application there remains the difficulty of identifying the relevant assumptions and nogood sets. This is equivalent to having selected a generic knowledge representation scheme, such as semantic nets, and then having to use it to represent the required knowledge.

The deficiencies of the prototype user modeller that has been described fall into two categories, relating to the general approach taken and to actual details. With respect to the general approach, the modeller constructs all possible user models without its being able to direct its attention to only a few of them. For example, in the "perquish" session it would not concentrate on the third command only, which was problematic, but would have generated all possible corrections for the second command as well. After the third command the system would have already ended up maintaining six hypothetical user models about what the last two commands could have been intended to be. Thus if there were a lot of similar names in the filestore it would run the risk of a combinatorial explosion. The system would maintain *all* the hypotheses generated until eventually some hypothetical sequences would end up in some error message. In addition, there is not a mechanism for generating hypotheses which would allow the modelling of possible misconceptions of the user about the semantics of UNIX commands.

In summary the main limitations of the described prototype implementation, are the following:

1. It does not have a mechanism for generating hypotheses concerning misconceptions of users.
2. It does not have a mechanism for focusing on certain commands that may have been problematic.
3. It maintains all the hypotheses generated until the command-sequences end up in some error message, if they ever do. In addition, it does not address errors of the user with respect to her/his intentions that do not produce an error message.

11. Recent Developments

Following the prototype implementation that has been described, there has been another prototype, called RESCUER (Virvou 1992, 1998) which was

originally constructed to address the limitations mentioned above. These are addressed in the following ways:

1. RESCUER addresses errors in semantics and syntax as well as typing errors and deeper misconceptions. It has a hypotheses generation mechanism which is based on the adaptation and application of a cognitive theory about human reasoning, called "Human Plausible Reasoning theory" (Collins and Michalski 1989). This theory was originally constructed to formalise the reasoning that people use to make plausible guesses. However, RESCUER exploits the fact that plausible guesses may be incorrect and therefore it uses the theory for generating hypotheses about "plausible human errors".
2. RESCUER evaluates each command typed with respect to certain criteria as to whether this command should attract more of its attention or not. For this reason, it has a mechanism for keeping track of users' intentions.
3. RESCUER does not get alerted only in cases where there has been an error message. It may decide to respond even in cases where there has not been any error message but there has been evidence that something may have been wrong. For example, action 3 of the "perquish" session would indeed be considered problematic by RESCUER as soon as this was issued despite the fact that the action did not produce any error message. The full summary of RESCUER's behaviour in the particular example is presented in (Virvou et al. 1999).

For the purposes of addressing all types of user misconceptions and mistakes RESCUER maintains a more detailed representation of a user model than the system described. For RESCUER a user model consists of hypothesised user beliefs about:

1. The configuration of the filestore.
2. The command intended.
3. The semantics of the command intended.

In particular, the command intended is considered by RESCUER as the command that the user meant to type rather than the command that would achieve her/his goal as considered by the previous system. The reason for this is that RESCUER needs a very detailed representation of the user's beliefs. Therefore it needs to distinguish between the command that the user had in mind to type and the command actually typed because these can be different if a typing error is involved. On the other hand, RESCUER distinguishes between the command that the user had in mind to type and the command that would achieve her/his goal because again these can be different. RESCUER regards the command the user had in mind to type as the command intended although it may not have been a correct planning method with respect to

her/his goals. What RESCUER considers as the command intended bears the information about possible misconceptions involved.

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.

Notes

[1] An error at the environment level would be failure to anticipate correctly the environment in which a particular action is executed.
[2] This is discussed in the companion paper in this volume as the ht example (Virvou et al. 1999).
[3] On some versions of UNIX, such as Berkeley 4.2, using cp on a directory produces a proper file which is a byte-level copy of the directory.
[4] The term 'plan' would be inappropriate, in that methods are relatively unstructured.

References

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

de Kleer, J. (1984). Choices Without Backtracking. In Proceedings of *AAAI 84*, 79–85. Austin, TX.

de Kleer, J. (1986). An Assumption-Based TMS. *Artificial Intelligence* **28**: 127–162.

Doyle, J. (1981). A Truth Maintenance System. *Artificial Intelligence* **12**: 231–272.

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

Finin, T. & Drager, D. (1986). *GUMS: A General User Modelling System.* Report MS-CIS-86-35, Department of Computer and Information Science, University of Pennsylvannia.

Hayes-Roth, B. (1983). *The Blackboard Architecture: A General Framework for Problem-Solving.* Report no. HPY-83-30, Stanford Computer Science Department.

Jones, J., Millington, M. & Ross, P. M. (1988). A Blackboard Shell in Prolog. In Englemore, R. & Morgan, A. J. (eds.) *Blackboard Systems*, 533–542. Wokingham: Addison-Wesley.

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. (1999). Knowledge Acquisition for Active Assistance. In Hegner, S., Mc Kevitt, P., Norvig, P. & Wilensky, R. (eds.) *Intelligent Systems for UNIX*. Dordrecht, The Netherlands: Kluwer Academic Publishers (this volume).

Norman, D. A. (1981). The Trouble With Unix. *Datamation* **27**(12): 139–150.

Rich, E. (1979). User Modelling Via Stereotypes. *Cognitive Science* **3**: 319–354.

Stirling, L. & Shapiro, E. (1986). *The Art of Prolog.* Cambridge: MIT Press.

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.*

Virvou, M., Jones, J. & Millington, M. (1999). Virtues and Problems of an active help system for UNIX. In Hegner, S., Mc Kevitt, P., Norvig, P. & Wilensky, R. (eds.) *Intelligent Systems for UNIX*. Dordrecht, The Netherlands: Kluwer Academic Publishers (this volume).

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