um: a toolkit for user modelling†

Judy Kay
Dept of Computer Science, University of Sydney, Australia

ABSTRACT

The um system provides an architecture for user modelling. Within that architecture is a collection of tools for acquiring and maintaining reusable user models. All the tools are implemented as rule-based programs in a shell that provides primitives for the user modelling tasks.

The underlying philosophy of um is to view the user as the owner of their user model and as a co-operative agent in the construction of their user model: accordingly, the user has access to their model and information about the processes that created it.

In keeping with the philosophy, the environment includes tools to co-operatively elicit modelling information. One such tool asks the user to externalise their propositional knowledge in an arbitrary domain and this gives models of their deeper conceptual understanding as well as more superficial and syntactic aspects of their knowledge.

This paper describes the architecture, examples of its tools and the underlying design concerns.

Keywords: shell systems for user modelling; response tailoring; acquisition of user and student models; representation of user models; user stereotypes; levels of user expertise; student modeling and tutoring strategies.


† This work was supported by grants from Telecom Australia and the Sydney County Council.
1. Introduction

A user model can be viewed as a system's beliefs about a user. It may contain various classes of information about the user: their current goals; details of interactions with them and their responses; longer term information such as their knowledge, background and interaction preferences. This paper is primarily concerned with the last of these and is motivated by the need for reusable modelling information that can be conveniently manipulated by different programs. After all, if a user has provided information about themselves (either by explicit interaction or indirectly by the system observing them) it seems reasonable that this model be applied to later interactions. Equally, that the user model should be accessible to programs other than just those involved in collecting the information. To do this, we need a convenient representation of the user modelling information and a user modelling component (Wahlster and Kobsa, 1986) to construct a user model, update it by additions and deletions of its components, maintain the consistency of the model and supply relevant parts of the model to client programs that need them. The um system provides an architecture with tools that do these functions. These are implemented using a shell for writing rule-based programs that maintain user models.

The underlying philosophy of much user modelling work is that the user model is an internal and hidden part of a program. If user models are to be reusable, they are no longer the property of one program. Moreover, if they are to be available to many programs, they should also be accessible to the user. In um, the user model belongs to the user who can access it and, if they wish, modify it. This does mean that a user can sabotage their user model but as that should only affect the quality of interaction that they get from programs, the philosophy of um is to allow them this right.

Another implication of reusability of modelling information is that it becomes important to structure the user model: modelling information that is irrelevant to a particular program needs to be kept separately from that which it needs. This is important both for efficiency and for maintaining simplicity. This structuring is a simple tree hierarchy of partial models as described in Section 2.

In the spirit of making the user model the property of the user, um includes tools that support a cooperative approach to constructing user models. This approach is particularly suited to student modelling where the user is a learner who must play an active role in the learning process. Involving the student is the construction of the system's model about them maintains the spirit of the active learner who has responsibility for and control over their own learning. Section 3 describes one tool that reflects this philosophy, cm, a program that structures the elicitation of the student's understanding of the domain knowledge.

The domain with which um has been developed is called trman, which tailors documentation about a computer operating system (Unix). Helping users in learning about such systems is a difficult and important area. The appeal of this domain is indicated by the number of similar projects: the Unix Consultant (Wilensky, Arens, Chin, 1984) which gives advice to natural language queries on how to achieve a goal; AQUA (Quili, Dyer and Flowers, 1985) which also deals with natural language queries from novices having problems with Unix; Borenstein (1985) who has developed and evaluated different forms of on-line manuals; Miller, Devillez and Vidal (1984) who provide support to a user in customising their environment; Mishra, Trojan, Burke and Douglas (1984) who developed a quasi-natural language based help system that supports casual users of Unix who were familiar with another system; O'Malley, Smolensky, Bannon, Conway, Graham, Sokolov and Monty (1983) who have studied helping users with printing tasks; Mason (1986) who developed an adaptive interface for perusing the manual system.

The tailoring done by trman is influenced by the user’s knowledge and aspects of their learning preferences. These are recorded in a user model which controls generation of manual entries. A user can be given manuals that have been minimally altered so that they should be able to understand the form presented to them. The underlying assumption is that a user will be better able to use a manual entry that excludes material they are unable to understand, a similar approach to that of Carroll and Carrithers (1984). The form of text displayed is also tailored to match some aspects of the user's preferred learning style so that a user who prefers concrete, functional descriptions gets different texts from those who like abstract descriptions that are the norm in Unix manuals. This quality of difference is similar to that offered by Paris (1986). Section 4 describes the user modelling aspects of trman, showing how cm and other tools fit into
its operation.

A note on terminology

As above, I use the terms user, student or learner model. All deal with similar types of information and for the most part, the differences relate to whether the user can be viewed as a student. In this paper, this is usually the case and the terms are often used interchangeably.

The literature shows some inconsistency in the use of various terms concerned with user modelling and I have followed definitions derived from Norman (1983) where a mental model resides in a person's head, a user model resides on a computer and parts of it are intended to have some useful relationship to the mental model. A conceptual model is a representation of some information (like a map or a diagram). It is to be emphasised that this paper makes no assertions about mental models. Rather it is concerned with pragmatic user models that support improved communication between the system and the user.

2. User model

The user model is composed of a collection of partial models. The partial models are ordinary ASCII files that can be read by the user. They are stored in a directory within the user's file space and are owned by the user. These are structured using the file system hierarchy as illustrated in the example of a model in Figure 1.

There is one partial model that has general information such as preferred user interface styles, preferred learning styles, the user's characteristics such as occupation, whether they program and other general details that may be useful to many different programs.

Two partial models are used to represent the user's knowledge about Unix, with a separation of aspects relating to the file system and those for the shell. The idea is that each of these partial models should collect details of various aspects of the user's knowledge in one sub-domain.

The partial models needed by tman to tailor manual entries include a representation of the user's relevant knowledge and of their preferred interaction style for this program. This may duplicate some of the information in the partial models described above. The principle here is that one collection of partial models is kept for the best available information about the user's knowledge about Unix. Parts of these are extracted or combined to give the tman partial model. The style partial model of tman may duplicate some of the aspects in the general model. This means that the user may have a preferred interaction or learning style in general and a different one for this program. In the case of learning styles, it is common for people to adopt differing learning styles to differing tasks, depending upon factors like the importance they place on each.

![Diagram of hierarchy of partial models]

Figure 1. Example of a hierarchy of partial models

This structure of the user model is very simple and its helps preserve the simplicity of the tools that deal with the partial models. As will be discussed in Section 3, tman needs to be able to resolve conflicting
sources of information about the user. Because the partial models should each be quite small, it is easier to produce programs that can reason about the conflicts within the partial models and which can perform efficiently because they deal with small models.

A partial model also has a quite simple structure: it is a list of model components, the most primitive modelling elements. The system that constructs the user model can create very fine grained components is that what is needed. In the domain of user’s knowledge of Unix, for example, it seems that users need to be modelled at a quite fine grain level as they tend to have very idiosyncratic patterns of understanding of the system (Sutcliffe and Old, 1987).

Each component can have information about its value and the sources of knowledge for concluding its value. This is illustrated in Figure 2 which is an excerpt from a partial model of a user’s knowledge of the Unix file system.

```plaintext
... simplest_file_commands: "simplest file system commands"
support
observation: history_anal 6: Nov 11 17:00:23;
negate
rule: cm_file1 8: Nov 19 13:35:10;

dir_function: "simple commands for manipulating directories"
support
rule: cm_file1 3: Nov 19 13:35:10;

dir_structure: "structure of directories"
support
rule: cm_file1 15: Nov 19 13:35:10;
```

*Figure 2. Excerpt from student model*

Figure 2 shows the most direct display given by viewer, a program that supports users in viewing their user model (and the only difference from the actual model representation is the times shown here are in a more natural format). The left most identifiers, like dir_function, are the component names. Each is followed by a list of the information supporting the system’s belief that this component applies to this user. There is also a list of the information negating this. Supporting and negating information are provided by the modelling programs. The following sources of knowledge can be shown:

- **prerequisite** means that the model contains a component Y and because this component (X) is prerequisite knowledge for Y, the user can be assumed to know X;

- **stereotype-based** indicates that the conclusion is based on predefined stereotypes (like the starting point of Rich, 1983).

- **observation-based** applies where the user’s actions on the system are the basis of the conclusion;

- **given** information has been provided to the system by some external source and this typically includes details such as the user’s background and employment status;

- **told** means that the user has been given information that should make this component true, as for example, the user may have been told about directories and this gives some support to the belief that they know about them;

- all other conclusions are based on another rule form in the modelling system.

Each justification has a time stamp showing when it was added to the user model. It also has the information needed to identify the rule within the modelling system that produced it, namely the name of the rule-set and the identifier (number) of that rule. The user can see more detailed displays from viewer to get details of the rules instead of the rather opaque references to rules shown in Figure 2. The strings that follow each component name are used by viewer to give more understandable texts of rules and displays of
A component can have some evidence supporting it and other evidence negating it as in the example of Figure 2 where the user has been observed to use several simple commands but could not demonstrate an understanding of them in a structured interview (controlled by the rule indicated). This might be the case for a user who has learnt formulaic approaches to doing tasks but does not understand much about the commands.

The model uses only qualitative justifications for conclusions about a user because this preserves the information available for resolving conflicting sources of information. Alternative representations like the numeric MYCIN-style certainty factors hide information: so, for example where there is little or no reliable information about a model component, MYCIN-style certainty factors represent this in exactly the same way as when there is strong evidence to support the conclusion as well as strong evidence negating it. In many contexts this is acceptable but in user modelling, it may be possible to make useful interpretations of these two situations. In teaching systems especially, it is often important to do just this. The approach taken in um is to retain the qualitative information in the model. It is easy to use a program to process a model, reducing all the qualitative information into numbers if that is what is required for a particular application.

The price paid for this qualitative representation is exacted in reasoning about conflicting information sources in the user model. um deals with conflicting information using two levels of partial ordering for justifications. The first level is

\[
given \succ prerequisite \succ rule \succ observation \succ told \succ stereotype
\]

which means, for example, that support due to given is treated as more significant than prerequisite, rule, observation, told or stereotype. The second level is based on time and this applies only within the first ordering. So, for example, where a component has both supporting and negating evidence at the same level, say rule, the more recent one is more significant. This ensures that newer information alters the model state, reflecting that a user has changed in some way, perhaps learning or forgetting.

Using this partial ordering um can give one of four values to a component: true, because the supporting evidence is evaluated as stronger, false, because the negating evidence is stronger, unknown, where there is no evidence about the component or conflicting which means that partial orderings cannot resolve the conflicting sources of information. Evaluating these could be quite costly if complex chains of rules need to be followed and one can construct cases where loops could prevent the evaluation. However, the small size of the partial models and rule sets makes it easier to avoid loops and, in practice, the evaluation can be done quite efficiently. Where components cannot be resolved because of loops, they are marked as unresolved. Optionally, um can output resolved user models giving the components and their resolved value.

Another qualitative approach is used in TMS (Doyle, 1987) although TMS deletes justifications to resolve conflicts. In this respect, um is more like ATMS (de Kleer, 1987). The partial ordering on the validity of different sources of justifications is closest to the entrenchments used by Gardenfors (1988).

3. A structured interactive tool for eliciting models of user knowledge - cm

Before describing how this works, it is useful to set this tool into some context. In situations where the user model needs to reflect the user's conceptual knowledge, the typical approaches involve observing the user. Although this requires little effort on the part of the user, it can require some considerable time to collect enough observations to be able to model a broad range of the user's knowledge. In the context of modelling a person's knowledge of the computer system, there are a number of other difficulties as well. For example, the person might get help at some stage (or allow others to use their account). Then the history record of that user is not really a reflection of what they know. Another problem in this approach is that there are commands that are typically used only in command files. Now if a modelling system ignores command files, then it will be very difficult to determine whether the user knows about such facilities. On the other hand, if the modeller does take account of commands used in command files, there is likely to be an overestimate of what they know as it is quite common and natural to use command files written by
others. A shortcoming of a quite different character relates to the level of knowledge that can be deduced from observing a user. Certainly, observations may give good information on the user’s knowledge of simple functional knowledge about the system and, with rather smarter analysis of user behaviour, one can detect multi-statement plans. (Just this is done by Shragg and Finlin, 1982.) But it is quite difficult to determine a user’s deeper conceptual knowledge by watching their behaviour.

Given all these difficulties, we need to complement the existing body of work on modelling based on observations with serious exploration of the possibilities for explicit interactions with the user to co-operatively construct user models. This view has been proffered by Wenger:

In fact, given the difficulty of the diagnostic task, it is somewhat surprising that existing systems do not resort more often to simple forms of interactive diagnosis involving constrained natural-language dialogues, or even menus or graphics. This is an interesting area for further research in the near future: not only may there be feasible solutions, but generalizable results would undoubtedly have important practical applications. (Wenger, 1987:392)

The form of interactive diagnosis supported by umn is based on a technique that was originally developed by Novak (1984) for educational evaluation. The essence of the technique is that is requires the student to present the knowledge in an area in a graphical form that gives the student’s particular view of the important relationships between major concepts in that area. This technique has been used in a number of ways, including: as a learning tool, Novak and Gowin (1984); as a codification of knowledge (Stewart and Van Kirk, 1981); for exploring the differences in character of different disciplines (Donald, 1980) and as a tool for knowledge acquisition for building expert systems (McAleese, 1987). Concept mapping does appears to give some useful insights into a person’s understanding. Moreover, I have found that concept mapping tasks that are quite constrained give useful information that is relatively straightforward to interpret. Experimental work that helped give guidelines for the design of concept mapping tasks and that informed the design of the concept mapping program, cmn, are described in Kay (1989).

As a component of umn, the interesting things about cmn are concerned with the way that it creates and amends user models. To illustrate this, consider how cmn is used to construct a partial model for the user’s knowledge of basic file structure concepts. A sample screen from a user’s concept mapping task is shown in Figure 3. The user was presented with the list of concepts in the panel at the left. They placed these on the large canvas area, with more general of inclusive concepts higher on the screen and related concepts placed near each other and level. So, in this case, the user has placed the command names, ls, more, rm, cat and rm, all at the same level and below the other concepts. Once happy with the relative placement of the concepts, the user defines propositions that relate concepts. They do this by selecting the appropriate link (from the panel at the lower left) and creating a connection between the concepts that are related by that link. In the map of Figure 3, the user has shown several propositions, including cat displays contents, directory contains file and file contains contents. This part of the mapping system is really a type of drawing program.

After the user has completed their map (indicated by clicking on the ready button), the rules for interpreting that map are executed. The first to run are those that require interaction with the user. The system supports requests for the user to check their use of particular concepts, the way they have used layout or other aspects of their maps. The idea is that rules can ensure that bizarre aspects of a map can be confirmed as what the user really intended before they are used to construct the user model. Provided that the rules are well constructed, this can avoid invalid conclusions based on accidental constructions. Once the user has dealt with requests for clarification of their map, the modelling rules execute. An excerpt from the rules that interpret the mapping task of Figure 3 is given in Figure 4. The first rule, for example, tests whether the proposition “directory is a file” appears in the concept map. If this proposition is not present, the user is given the message asking them to check this aspect of their map. This proposition is at a very conceptual

† This is Wenger’s term for the task of identifying what a student knows.
level. By contrast, the other rule relates to the user’s functional knowledge. Its antecedent checks if at least three of the components are in the user model. Those components were defined by earlier rules, each of which checked for propositions that reflected knowledge of the function performed by the commands mv, ls, at least one of more and cat and rm. Other primitives available in cm allow tests for the relative vertical position of concepts (above, below, level), checking whether a concept has not been used or is on the map but is unlinked to others, whether links have been used, whether a user cannot use a link or concept even after being asked to try to do so and whether given concepts have been grouped.

In cm, the major goal is to construct a partial model of the user’s conceptual knowledge in some area. In addition, cm maintains a partial model of the user’s concept mapping skill. This can be used in rules that can interpret a map in relation to the person’s mapping skill. It can also be used in rules that give tips on how to improve mapping skill and it is used by cm to define the level of detail in the instructions given to the user of the program.

Although cm was devised for the documentation tailor, it has other potential applications. For example, system designers make many assumptions about the way that users perceive their systems. These could be tested by a concept mapping system like cm. An even more appealing use of cm would be as part of a teaching system that presented new material in terms of the user’s concept map. In addition, the rule-based shell that supports cm can be used to write rules that interpret concept maps for other domains.
The main concern in developing cm was to explore tools for co-operative elicitation of user modelling information. The strength of cm is in eliciting conceptual knowledge. It requires some effort from the user and is not appropriate for all users (as observed in Kay, 1989) or for all situations. It is intended for cases where detailed conceptual knowledge is needed as in teaching systems. Other tools are currently being developed and evaluated for applying the same co-operative approach for eliciting other types of modelling information. These include programs that ask the user to indicate whether they understand each of a list of concepts (a similar approach to UMFE by Sleeman, 1985) and other forms of co-operative construction of the user model.

```plaintext
*/
** looks for proposition showing that a directory is just a file,
** important to understanding how the file system structuring
** and various commands work
*/
if [directory is_a file]
then
    dir_structure;
else
    ask "check you have all the important links for directory and file";

/* a person who understands the file system should have demonstrated knowledge
** of the purpose of at least 3 of the four commands given
*/
if count(3, mv_function, ls_function, more_cat_function, rm_function)
then
    simplest_file_commands;
else
{
    --file_system;
    --simplest_file_commands;
}
```

Figure 4. Examples of rule forms for interpreting concept maps

4. Basic form of modelling tools

Modelling programs are implemented in two layers: the base layer is a shell which interprets rule-based modelling programs that constitute the upper layer. Each tool within um has a common set of primitive operations. These allow rules of the form:

```plaintext
if antecedent test
then
     then-part action
[else
     else-part action]
[because
     because-part]
```

where the antecedent can perform tests that are boolean combinations of simple tests on the value of model components or the results of additional primitives for the particular application. The then and else parts can perform primitive actions which amend the user model. The because-part defines the basis for resolving conflicting sources of information, as described in Section 2.
5. System overview

To illustrate how the tools operate, consider Figure 5 which gives an overview of um with examples in terms of the tools used in the tman application.

![Diagram](image)

**Figure 5. Overview of um (taking examples from tman)**

A user model can gain information from sources that require user interaction or not. Tools that elicit information from the user, like cm, need to be able to manipulate user models, deal with user interactions and, in keeping with the um philosophy, confirm important aspects of the model with the user. Other tools, that do not require user interaction are much simpler. For example, sm takes the partial models on Unix knowledge and produces the simpler model of user knowledge required by tman. Also, dm is a default modeller that operates where tman needs to operate before there partial models on the user's knowledge are available and this uses stereotypes based on other information about the user's status on the machine to make a default user model for tman. Both classes of programs can be implemented using the rule-based shells or they can be arbitrary programs that use the library of tools for dealing with user models.

The last part of the system is the programs that act as consumers of the user model. Like tman, these have read-only access to the model. In fact, tman and other consumer programs can indirectly alter the user model as they control the invocation of tools like cm or dm and they can leave "other information" (as depicted at the bottom of Figure 5) for modelling programs to use.
6. Conclusion

The main contribution of um is the provision of an architecture and tools that support the creation, maintenance, use and reuse of user modelling information. In this sense, um represents an experiment in the development of a general user modelling shell as described by Wahlster and Kobsa (1986), with a representation for beliefs about the user and a means for dealing with inconsistencies and changes in the user model.

The tools within um have consistency and flexibility as they are all implemented in a rule-based shell for writing programs that manipulate user models. Each offers the same set of user modelling primitives and it is straightforward to add new primitives for particular classes of application that demand additional tests or actions in the rules.

The um architecture supports hierarchically structured models. This means that the separate concerns of different programs can be kept separate. It also allows context dependent aspects of a user to be conveniently represented. It is also important for efficiency as the algorithms that resolve conflicts need only deal with small sets of components at one time.

One interesting class of rule programs that is supported within um has primitives for interacting with a user in eliciting their view of the knowledge in an arbitrary domain. This tool, crm, is of particular significance as it tackles the difficult problem of acquiring modelling information about the user's conceptual knowledge, as well as the more functional or syntactic knowledge. This elicitation system includes, in addition to the modelling primitives um has for combining partial models and for resolving conflicting sources of information, special primitives for reasoning about the user's responses and for verifying the system conclusions. It achieves the latter by including primitive rule actions for asking the user to check particular aspects where these should typically include the very odd or critical aspects.

An important aspect of um is the underlying philosophy. It gives the user ownership of their user model and it permits the user to play an active role in constructing their own user model. Allowing the user to be a co-operative agent is important in the way that it defines the relationship between the user, their user model and the machine. It also obviates the need for an omniscient or terribly clever system.

Acknowledgements

Thanks to my colleagues for support and especially Alan Fekete, Norman Foo, Bob Kummerfeld, Robert Rist and Richard Thomas for helpful comments on drafts of this paper.

REFERENCES


